2018 IEEE Computer Society Annual Symposium on VLSI

Enhancing Observability for Post-Silicon Debug with On-Chip Communication Monitors

Yuting Cao, Hernan Palombo, Hao Zheng
CSE, U of South Florida, Tampa, FL
{cao2, hpalombo, haozheng}@mail.usf.edu

Sandip Ray
ECE, U of Florida
Gainesville, FL
sandip@ece.ufl.edu

ABSTRACT

Reconstructing system-level behavior from silicon traces is critical in post-silicon debug of System-on-Chip (SoC) designs. However, limited observability makes the reconstruction process complex and inaccurate, thus offering little help for SoC debug. This paper presents an on-chip monitoring infrastructure aiming to enhance observability by detecting communication transactions from low level signal events. The detected transactions are output on-the-fly for off-chip system-level behavior reconstruction. Experiments show that the proposed monitoring infrastructure enables accurate observations on communication transactions over long periods of time, thus leading to accurate reconstruction of system level behavior, with low area overhead.

1. INTRODUCTION

Post-silicon debug is a critical component of the design validation life-cycle for modern microprocessors and system-on-chip (SoC) designs. Unfortunately, it is also highly complex, performed under aggressive schedules and accounting for more than 50% of the overall design validation cost [1].

An SoC design is often composed of a large number of pre-designed hardware or software IP blocks that coordinate through complex protocols to implement system-level behavior [2]. As SoCs integrate more IPs, the interactions among the IPs are increasingly more complex. Moreover, modern interconnects are highly concurrent, allowing multiple system-level protocols to be processed simultaneously for scalability and performance. Many difficult errors often involve subtle and unexpected interleavings of these protocols that have a very low probability of being exercised in simulation [3] On the other hand, observability limitations allow only a small number of participating signals to be actually traced during silicon execution. It is non-trivial during post-silicon debug to identify all participating protocols and pinpoint the interleavings that result in an observed trace.

In [4], an off-chip analysis approach was proposed with an objective to reconstruct design internal behavior *wrt* given system-level flow specifications. Since *when* and *where* an error can happen are not known *a priori*, in our approach silicon traces on a small number of observable hardware signals are streamed off-chip on-the-fly in order to facilitate more efficient debug in space and time. Once a silicon trace is obtained, it is processed by the off-chip analysis in two consecutive steps: (1) trace abstraction, which maps a silicon trace into a sequence of *flow events*, abstract architectural constructs representing communication transactions,

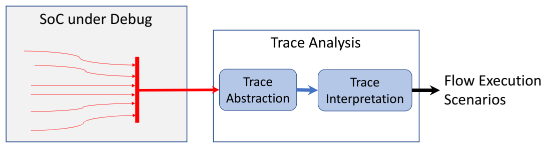

Figure 1: The system flow guided post silicon trace analysis framework for SoC debug in our previous work. The red lines across represents raw signal traces while the blue line represents the flow trace after the trace abstraction step.

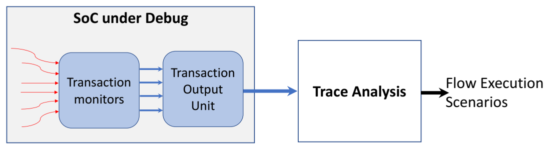

Figure 2: The new framework where the SoC under debug is instrumented with an on-chip monitoring infrastructure and the off-chip trace analysis no longer needs the trace abstraction component.

and (2) trace interpretation, which infers possible flow execution scenarios that are compliant with the abstracted event sequence. That approach is shown in Figure 1. Its main drawback is that the reconstruction process is highly complex and inaccurate due to the insufficient information that can be derived from the silicon traces observed on a very small number of trace signals. The inferred results are often ambiguous, and provide limited help for debug.

Contributions This paper addresses the above drawback by proposing an on-chip communication monitoring infrastructure to enhance debug observability. It consists of transaction monitors attached to the communication links of a SoC design, and a transaction output unit that manages transporting detected transactions by the monitors on-the-fly for off-chip analysis. The new trace analysis approach is shown in Figure 2. Instead of streaming observed signals off-chip every clock cycle, the monitors offload transactions only when they happen. Since transactions typically happen sporadically over time, it allows transactions from different communication links to be interleaved while being outputted for off-chip analysis. On the same limited number of trace signals, every event now can capture much more accurate

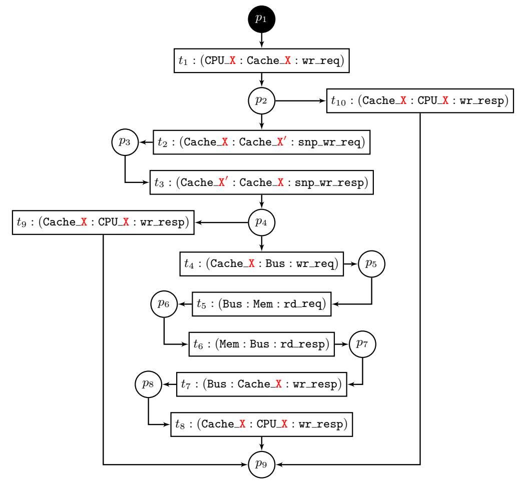
p_1
t_1 : (CPU_X : Cache_X : wr_req)
p_2
t_{10} : (Cache_X : CPU_X : wr_resp)
p_3
t_2 : (Cache_X : Cache_X′ : snp_wr_req)
t_3 : (Cache_X′ : Cache_X : snp_wr_resp)
t_9 : (Cache_X : CPU_X : wr_resp)
p_4
t_4 : (Cache_X : Bus : wr_req)
p_5
p_6
t_5 : (Bus : Mem : rd_req)
t_6 : (Mem : Bus : rd_resp)
p_7
p_8
t_7 : (Bus : Cache_X : wr_resp)
t_8 : (Cache_X : CPU_X : wr_resp)
p_9

Figure 3: LPN formalization of a CPU write protocol.

information on internal communications during system execution, it allows the trace analysis method to reconstruct system-level behavior more accurately and efficiently.

2. PREVIOUS WORK

This section briefly reviews our previous work on the system flow guided post-silicon trace analysis for SoC debug [4].

In our trace analysis method, system-level protocols or *system flows* are formalized using Labeled Petri-nets (LPNs). Figure 3 shows a memory write protocol initiated from a CPU CPU_X in LPN where $\mathtt{X} \in \{0, 1\}$ and $\mathtt{X}' = 1 - \mathtt{X}$. An LPN is a tuple (P, T, E, L, s_0) where P is a finite set of *places*, T is a finite set of *transitions*, 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 $e \in E$. In a system flow specification, each LPN transition is labeled with an event (src, dest, cmd) where cmd is a command sent from a source component src to a destination component dest. 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 state $s \subseteq P$ of a LPN is a subset of places marked with tokens. There are two special states associated with each LPN; $s_0 \subseteq P$ which is the set of initially marked places, also referred to as the *initial state*, and the end state s_{end} which is the set of places not going to any transitions.

A transition t can be executed in a state s if $\bullet t \subseteq s$. Executing t causes the labeled event to be emitted, and leads to a new state $s' = (s - \bullet t) \cup t\bullet$. Therefore, executing an LPN leads to a sequence of events. Execution of a LPN completes if its s_{end} is reached. For example, in Figure 3, t_1 can be executed in $s_0 = \{p_1\}$. Event (CPU_X : Cache_X : wr_req) is emitted after t_1 is executed, and the LPN state becomes $\{p_2\}$. The end state is $s_{end} = \{p_9\}$. A flow specification may also contain multiple branches describing different ways a system can execute such flow. For example, the flow shown in Figure 3 has three branches covering the cases where the cache (snoop) operation is hit or miss.

The objective of our silicon trace analysis is to infer possible compliant *flow execution scenarios* from a partially observed trace *wrt* given system-level flow specifications $\mathbf{F}$. A flow execution scenario can be viewed as *a state of system execution abstracted wrt system flows*, and it is defined as

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

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

To illustrate the basic idea, consider the system flow in Figure 3, which we call F_1. Suppose that the following flow event trace is abstracted from an observed silicon trace by executing a design that implements F_1.

1 $\langle$(CPU_0 : Cache_0 : wr_req)$\rangle$
2 $\langle$(CPU_0 : Cache_0 : wr_req), (CPU_1 : Cache_1 : wr_req)$\rangle$
3 $\langle$(Cache_0 : CPU_0 : wr_resp),
(Cache_1 : Cache_0 : snp_wr_req)$\rangle$
...

The numbers on the left in the above trace represent the relative time when the corresponding events are observed. Possible flow execution scenarios that are derived by our approach are shown below.

$$\{(F_{0,1}, \{p_9\}, 1, 3),\ (F_{0,2}, \{p_2\}, 2, -),\ (F_{1,1}, \{p_3\}, 2, -)\},$$
$$\{(F_{0,1}, \{p_2\}, 1, -),\ (F_{0,2}, \{p_9\}, 2, 3),\ (F_{1,1}, \{p_3\}, 2, -)\}$$

The above flow execution scenarios indicate two possible system states defined over the execution states of flow F_1 after observing the first three steps of events as shown above. This ambiguity is mainly due to lack of necessary information in the observed events due to the limited observability in post-silicon debug.

The limited observability problem leads to two unpleasant consequences. The large number of execution scenarios typically derived during the trace analysis would take longer runtime and large amount of memory to process and to store, thus making it less efficient. This is referred to as the *complexity* problem of the trace analysis. After the analysis is done, a large number of derived execution scenarios make it difficult to understand the analysis results, thus being less helpful for debugging. Obviously, a single flow execution scenario derived at the end of the trace analysis provides much more precise information for debug than ten candidate flow execution scenarios. This is referred to as the *accuracy* problem of the trace analysis. Those problems are addressed in the next section that describes an on-chip monitoring infrastructure.

3. MONITORING INFRASTRUCTURE

In order to enhance observability and facilitate the trace analysis, this section describes a communication monitoring infrastructure. It consists of monitors attached to communication links to detect communication transactions, and a

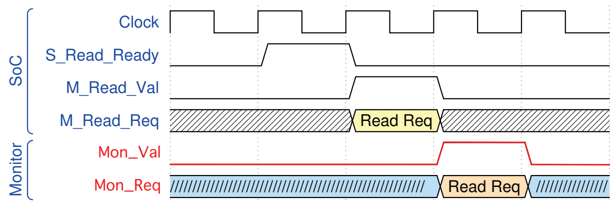

Figure 4: An example of the AXI read transaction on a communication link, and the output of a monitor attached to that link.

transaction output component to transport detected transactions off-chip for the trace analysis.

3.1 Communication Transaction Monitoring

A communication link consists of signals that can transfer some transactions, one at a time. Instead of routing a limited number of design signals to the chip interface, signals of a communication link are connected to the inputs of its attached monitor. During system execution, a monitor reads signal events occurring on its inputs, and outputs an encoding if a transaction is detected. A *signal event* denotes an assignment to a set of design signals, while a communication transaction is a transfer of a body of information from a source to a destination following a communication protocol. *Flow events* are an abstract construct used in flow specifications, and are typically implemented by transactions.

Figure 4 shows an example of a master and a slave communicating with the AXI read protocol [5]. The slave needs to assert `S_Read_Ready` before the master asserts `M_Read_Val` to initiate a transaction. A transaction of a AXI read request is detected by its monitor with a pulse on `Mon_Val` when `M_Read_Val` is asserted. The monitor can also selectively encode some information transferred as part of the detected transaction. The basic idea of the above monitor can be naturally extended to different protocols such as the AXI write request and response.

The biggest benefit from those monitors is the compression of a potentially long sequence of signal events into a single cycle communication transaction. Obviously, transporting this single cycle transaction demands much less bandwidth of the trace port than transporting low level signal events implementing such transaction. Another advantage is that these monitors can implement protocol checking capability so that low level protocol errors can be detected timely and right on the spot.

3.2 Transaction Output

The detected transactions can be stored in the on-chip trace buffer, and offloaded from the chip at the end of system execution. However, the on-chip trace buffers can only store limited transactions due to the restriction on their capacities. As explained in Section 1, when and where an error can happen are not know a priori, therefore, these limited transactions stored in the trace buffer may offer only limited debugability. This section describes a transaction output design that can output transactions via trace port on-the-fly, thus enabling system internal execution over an much extended period to be observed for off-chip analysis.

Parallel Output The first approach is parallel where multiple links are traced simultaneously. Since the number of available trace signals are fixed, there is a trade-off between the number of links that can be traced simultaneously and the amount of information encoded for detected transactions on each link. More information encoded for transactions demands more trace signals, thus reducing the number of links that can be traced simultaneously. For example, suppose that a total of 100 trace signals are available, and there are 20 communication links to observe. If each transaction generated by the monitors for those links is encoded with 30 bits on average, then only 3 links can be traced simultaneously. On the other hand, if we wish to observe more communication links simultaneously, the number of bits for encoding transactions must be reduced, thus limiting the amount of information represented by transactions. More discussion on transaction encodings is given in Section 3.3.

Interleaved Output Since detected transactions by monitors are distributed over time relatively sparsely as illustrated in the last sub-section, an alternative approach is to interleave transactions detected on different links, and transport them off-chip serially. In this approach, monitors are connected to a transaction output unit like the one used in ARM CoreSight [6], which is shown in Figure 5. The transactions from monitors are routed through this output unit, merged into a sequence, and eventually output through the trace port. The biggest advantage of this approach is the very high observability in terms of the larger number of communication links to be traced and the higher amount of detailed information that can be encoded for transactions.

On the other hand, an issue with the interleaved approach is that the rate of transactions detected by monitors can exceed the peak bandwidth of the trace port from time to time. If that happens, some detected transactions cannot be transported off-chip. It can be viewed as another form of limited observability. Therefore, it would be desirable to reduce the number of transactions that have to be discarded.

The above issue can be addressed by using FIFOs as shown in Figure 5. Those FIFOs can buffer detected transactions temporarily if they cannot be outputted right away. One FIFO is connected to the output of each monitor. On every cycle, the outputs of all monitors carrying detected transactions are stored into the corresponding FIFOs. At the same time, the transaction validity information of all monitors is collected into `Tr_Val`, and stored into a special FIFO `Tr_Val_fifo`. This information is used to control how to output buffered transactions. The width of `Tr_Val` is equal to the number of monitors. `Tr_Val[i]=1` indicates that transaction output from monitor M_i is valid. Otherwise, no valid output is from M_i. All the transaction FIFOs are connected to a N-to-1 selector where one transaction FIFO is routed to the trace port.

The control logic generates values for `sel` to control the selector based on the information stored in `Tr_Val_fifo`. In the initial state, it asserts `Read_Tr_Val` to read the head of `Tr_Val_fifo` into `Tr_Status`. If it contains some bits of 1, the control unit first determines the smallest index `i` such that `Tr_Status[i]=1`. This can be done by a priority encoder. Next, the transaction FIFO for monitor M_i is connected to the trace port, and the transaction at its head is outputted. Then, `Tr_Status[i]` is reset to 0, and the control logic repeats the above step if there is a larger index `i` such that `Tr_Status[i]=1`. Otherwise, it returns to the ini-

Cycle	M_2	M_1	M_0	`Tr_Val`	`Read_Tr_Val`	`Tr_Status`	`Sel`	Selector Output
1	✓	✘	✓	101	1	000	X	–
2	✘	✓	✘	010	0	101	0	M_0
3	✘	✘	✘	not stored	0	100	2	M_2
4	✘	✘	✘	not stored	1	010	1	M_1
5	✘	✘	✘	not stored	1	000	X	–

Table 1: Operations of the transaction output unit for 3 monitors over 5 cycles.

tial state and read the next data from `Tr_Val_fifo`. When `Tr_Status` is 0, `sel` is set to a special value `X` to disable the selector. The control flow diagram for the control unit is at the bottom right corner in Figure. 5.

Illustration We illustrate the operations of the transaction output unit with a simple example. Suppose that there are three monitors M_0, M_1 and M_2 connected to the transaction output unit. Their validity information is collected as a 3 bit `Tr_Val` and stored at the tail of `Tr_Status_fifo` on each cycle. Table 1 shows how the output unit outputs the transactions from these monitors off-chip. In the columns under M_0, M_1 and M_2, a ✓ means its generated transaction is valid, while ✘ indicates it is not valid.

In cycle 1, outputs from M_0 and M_2 are valid. They and `Tr_Val=101` are stored into their corresponding FIFOs, respectively. There is nothing to output, therefore `sel` is set to `X`. `Read_Tr_Val` is set to 1 to read `Tr_Val_fifo`. In cycle 2, similarly, outputs of all monitors and their validity are stored in the FIFOs. `Tr_Status` is used to compute `sel`, which is 0 in this case. Therefore, the selector is directed to output the transaction from M_0 captured in cycle 1, as indicated in column of Selector Output. `Tr_Status[0]` is reset to 0 before the next cycle. Signal `Read_Tr_Val` is reset to 0 when `Tr_Status` contains more than one bit of 1. In cycle 3, the transaction from M_2 captured in cycle 1 is outputted by the selector, and `Read_Tr_Val` is asserted to read next validity data from `Tr_Val_fifo` to prepare for the next cycle.

While the interleaved approach reduces the number of trace signals needed to output detected transactions, the transaction output unit can potentially introduce large area overhead. The area overhead are mainly due to the use of FIFOs to buffer transactions. Larger FIFOs can reduce the chance of transactions to be discarded, but increases area overhead. The optimal FIFO sizes are typically determined by the design of the interconnect network and the rates at which various blocks initiate system flows. However, detailed discussion of this topic is beyond the scope of this paper. In this paper, when overflow happens to any FIFOs, new incoming transactions are discarded as a simplification.

3.3 Transaction Encoding

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

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

$$\langle \texttt{Val},\ \texttt{Cmd},\ \texttt{Tag},\ \texttt{Sid},\ \texttt{Addr} \rangle$$

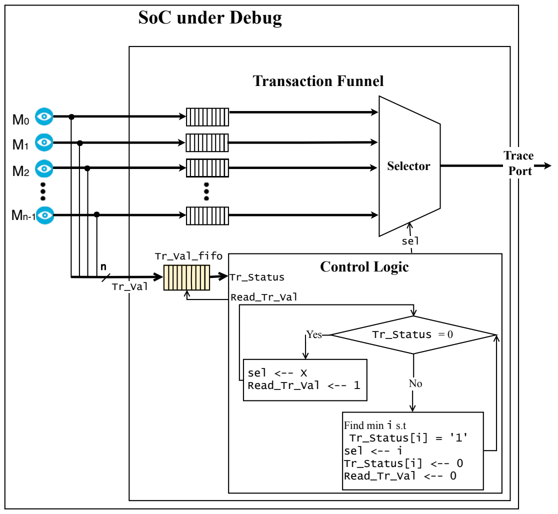

Figure 5: Transaction output unit.

The meanings of the message fields are defined below.

`Val` indicates the validity of a detected transaction.

`Cmd` carries operations to be performed by the target block. For the AXI protocol, there are separate links to support read/write request and response operations, this field is not needed.

`Tag` is used to identify the original sources of transactions from different blocks that go to the same destination, For example, in Figure 6, `Tag` is needed for transaction `wr_req` from `Bus` to `Memory` in response to `wr_req` from either CPU.

`Sid` is a unique number representing sequencing information associated with transactions initiated by a component that supports out-of-order execution.

`Addr` carries the memory address at the target block where `Cmd` is applied. If the observability limitation does not allow full address information to be encoded, it can be abstracted with two bits to represent three states: *same as previous one*, *sequential*, and *others*, as described in [7].

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

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

$$\langle \texttt{Val}, \texttt{MasterID},\ \texttt{SlaveID},\ \texttt{Cmd},\ \texttt{Tag},\ \texttt{Sid},\ \texttt{Addr},\ \texttt{Step} \rangle \tag{1}$$

where

- `MasterID` and `SlaveID` encode IDs of the sender and receiver of a transaction. The number of bits required for these two fields are determined by the number of masters and slaves in an SoC design.
- `Cmd` has a fixed number of bits for all transactions. Its width is determined by the largest number of transactions that any link can transfer.

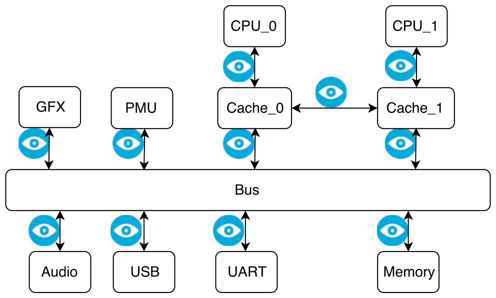

Figure 6: A SoC prototype where each communication link is attached with a monitor.

- `Val`, `Tag`, `Sid`,and `Addr` are the same.
- `Step`, which is only 1-bit, indicates the ordering of a transaction relative to its immediate predecessor. If this field is asserted, it indicates that the current transaction being outputted is detected after its immediate predecessor. Otherwise, the current transaction and its immediate predecessor are detected at the same time.

In the transaction output unit as shown in Figure 5, when a transaction is pulled out of its FIFO, it is converted to the above standard format before it enters the selector. Similarly, the fields and their sizes in the above standard format can be reduced to meet the observability constraint.

4. EXPERIMENTAL RESULTS

4.1 The Model

To evaluate the ideas and techniques presented in this paper, a non-trivial SoC design that implements sophisticated system flows is desired. However, to the best of our knowledge, we cannot find an open-source design that meets the above requirements. Therefore, we developed a multi-core SoC prototype, as shown in Figure 6, which implements a total of 16 system-level protocols including cache coherence, power management, downstream read/write protocols for CPUs, upstream read/write for the peripheral blocks, *etc.* All of them are abstracted from real industrial protocols. More details on some of those protocols can be found in [8].

This prototype is a cycle- and pin-accurate RTL model. The above system-level protocols are supported by inter-block communication protocols based on the ARM AXI4-lite [5]. A total of 32 monitors are inserted into this model, one for each link between a device and the interconnect,

Since the proposed monitoring infrastructure is to support communication centric trace analysis, the focus of this model is the implementation of system flows for on-chip interconnect. The CPUs are treated as a test environment where software programs are simulated in VHDL to trigger various protocols. Therefore, there is no instruction cache as no instructions are involved when the CPUs are simulated. The peripheral blocks, GFX, PMU, Audio, *etc*, are also described as abstract models that generate events to initiate flows or to respond incoming requests.

4.2 Experiment Setup and Result Analysis

The model is simulated where five components, including CPUs, GFX, and three other peripheral blocks are programmed to randomly select a flow to initiate in every five clock cycles. The contents of `Cmd`, `Addr`, and `Data` in each activated flow are set randomly. Additionally, CPUs can activate power management protocols randomly in time. Each of those five blocks activates a total of 100 flow instances during entire simulation. After simulation traces are obtained, the trace analysis approach in [4] is applied to extract the flow execution scenarios.

	Full	Parallel	Interleaved	SS1	SS2
# Bits	870	720	36	36	36
# scen (Max)	1	1	1	1	282k
# scen (Final)	1	1	1	1	-
#flows	500	500	500	100	200
Time	1.391	1.237	1.218	0.714	600
Mem	1.068	1.017	1.028	0.608	>5GB

Table 2: Runtime Results from analyzing traces obtained in different approaches. Runtime is in seconds and memory usage is in MB.

In the experiment, simulation is run five times with different tracing configurations. In the first run, "Full" observability is assumed. In the second run, the monitoring infrastructure is configured in the "Parallel" output mode where all links are assumed to be observable. In the third run, the monitoring infrastructure is configured in the "Interleaved" output mode. In this run where the transaction output unit along with monitors are used, all fields in (1) except `Addr` are used for transactions through the trace port. In our SoC model, `Val` and `Time` are 1 bit, `MasterID` and `SlaveID` are 5 bits, and `Cmd`, `Tag`, and `Sid` all are 8 bits. Therefore, encoding of transactions in the standard format as in (1) requires 36 bits. The depth of all FIFOs in the output unit is set to 16. The last two runs generate results from using tracing without the communication monitors.

The results from all runs are shown in Table 2. In the table, row 2 and 3 show the *peak* count of flow execution scenarios encountered during the reconstruction process that is used to measure the complexity and the *final* count of flow execution scenarios derived at the end of the reconstruction process that is used to measure the accuracy, respectively. Row 5 shows the maximal number of flow instances activated by various components and identified by the trace analysis. The last two rows show the runtime and memory usages by the trace analysis for different runs. The results from analyzing the simulation traces of those five runs are shown in column $2-6$, respectively, in Table 2.

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

The last two columns show the trace analysis results without using the monitoring infrastructure. We assume that 36 signals are available for tracing as in the third run. Due to this restriction, we can select only a small number of links where the signal events can be observed accurately. The re-

	Cells	LUTs	FFs	Muxs	BRAM
Original	59154	24395	25962	3125	1
Parallel	+1283	+8	+1251	+15	+0
Interleaved	+232	+92	+126	-6	+32

Table 3: Area overhead of the monitoring infrastructure.

sults from this run are shown in Column 5. Even though only one flow execution scenario is derived at the end, the limited number of signal events selected for observation allow much less number of related flow instances to be derived than what can be derived when the monitoring infrastructure is used. When we try to observe more links, we are forced to allocate less trace signals for each event on those links. This causes ambiguity to the interpretations of the observed signal events. As a result, an excessively large number of potential flow execution scenarios are derived as shown in Column 6. After 10 minutes, the trace analysis has to be terminated due to memory usage explosion. These results show that under the limited observability the complexity and accuracy of the trace analysis would suffer significantly if the monitoring infrastructure is not used.

4.3 Area overhead

We measure the hardware area overhead of the monitoring infrastructure by synthesizing the SoC model to the Xilinx Zynq FPGA xc7z020ckg484-1 using Vivado 2017.2. The synthesis results are shown in Table 3. The area overhead is measured by the FPGA resources used including LUTs, FFs, block RAMs (BRAMs), etc.

The rows for Parallel and Interleaved show the additional resources required to implement monitors with or without transaction output unit to the resources used on the previous row. For example, the parallel approach uses additional 1283 cells to implement all 32 monitors, and the interleaved approach requires extra 283 cells to implement the transaction output unit. From the table, other than the big jump in BRAM usage, demand on logic resource is small to implement the monitoring infrastructure. The BRAMs are used to implement the FIFOs in Figure 5. Since the size of the BRAMs is fixed, each FIFO only uses a small capacity of a BRAM. In practice, SoCs are often embedded trace buffers, which can be used for those FIFOs.

5. RELATED WORK

Ciordas and e.t. in [9] proposed the first monitoring service to provide run-time observability of NoC behavior and supporting system-level debugging. However, it offers no explanation on how the detected events are outputted for off-chip analysis. Gharehbaghi and Fujita in [7] [10] [11] introduce an on-chip instrumentation that allows transaction level message abstraction using formal specifications of the bus communication protocols. The authors propose an innovative encoding technique that can reduce the address bits to two bits containing three different states to indicate relationships between two consecutive transactions. Despite its low area overhead, this methods suffers from inability of detecting implementation errors that are are not observed. Moreover, this method lacks the ability to check the overall system communication protocols as it only focus on communication interfaces' protocol. [12] proposes another on-chip instrumentation BiPeD that learns communication interface' protocol during pre-silicon stage, and reconfigure its detection hardware to check the learned protocols during the post-silicon validation. While BiPed is effective towards detecting and locating many hardware bugs, the circular buffer implemented for each communication interface introduces large area overhead. Another transaction monitor is proposed by [13] where a generic template for bug and router monitors are presented. However, it mainly targets run-stop debug control instead of real-time tracing.

6. CONCLUSION

This paper describes an on-chip monitoring infrastructure that detects and outputs communication transactions on-the-fly for off-chip inferences of system-level behavior for efficient post-silicon debug. Initial experiments show some promising results. In the future, we plan to perform in-depth study on using the described infrastructure on SoC designs with diverse interconnects, and explore optimizations for the infrastructure to offer higher observability with reduced hardware overhead.

Acknowledgement The research presented in this paper was partially supported by a gift from the Intel Cooperation.

7. REFERENCES

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

[2] Harry D. Foster. Trends in functional verification: A 2014 industry study. In *DAC*, pages 48:1–48:6, 2015.

[3] M. Talupur, S. Ray, and J. Erickson. Transaction flows and executable models: Formalization and analysis of message-passing protocols. In *FMCAD*, 2015.

[4] Y. Cao, H. Zheng, H. Palombo, S. Ray, and J. Yang. A post-silicon trace analysis approach for system-on-chip protocol debug. In *ICCD*, 2017.

[5] Amba axi and ace protocol specification. http://www.arm.com.

[6] ARM. Coresight architecture specification v2.0, 2013.

[7] A. M. Gharehbaghi and M. Fujita. On-chip transaction level debug support for system-on-chips. In *ISOCC*, pages 124–127, Nov 2009.

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

[9] Calin Ciordas, Twan Basten, Andrei Rădulescu, Kees Goossens, and Jef Van Meerbergen. An event-based monitoring service for networks on chip. *ACM TODAES*, 10(4):702–723, October 2005.

[10] A. M. Gharehbaghi and M. Fujita. Transaction-based debugging of system-on-chips with patterns. In *ICCD'09*, pages 186–192, Oct 2009.

[11] A. M. Gharehbaghi and M. Fujita. Transaction-based post-silicon debug of many-core system-on-chips. In *ISQED*, pages 702–708, March 2012.

[12] A. DeOrio, J. Li, and V. Bertacco. Bridging pre- and post-silicon debugging with biped. In *ICCAD*, 2012.

[13] B. Vermeulen and K. Goossens. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs. In *VLSI-DAT'09*, pages 183–186, April 2009.