Application Level Hardware Tracing for Scaling Post-Silicon Debug

Debjit Pal*, Abhishek Sharma§, Sandip Ray†, Flavio M. de Paula‡, Shobha Vasudevan¶

University of Illinois, Urbana-Champaign*§¶, University of Florida, Gainesville†, IBM Austin‡
{dpal2,sharma53,shobhav}@illinois.edu,sandip@ece.ufl.edu,fmdepaul@us.ibm.com

ABSTRACT

We present a method for selecting trace messages for post-silicon validation of Systems-on-a-Chips (SoCs) with diverse usage scenarios. We model specifications of interacting flows in typical applications. Our method optimizes trace buffer utilization and flow specification coverage. We present debugging and root cause analysis of subtle bugs in the industry scale OpenSPARC T2 processor. We demonstrate that this scale is beyond the capacity of current tracing approaches. We achieve trace buffer utilization of 98.96% with a flow specification coverage of 94.3% (average). We localize bugs to 21.11% (average) of the potential root causes in our large-scale debugging effort.

CCS CONCEPTS

• **Hardware → Bug detection, localization and diagnosis**;

ACM Reference Format:
Debjit Pal*, Abhishek Sharma§, Sandip Ray†, Flavio M. de Paula‡, Shobha Vasudevan¶. 2018. Application Level Hardware Tracing for Scaling Post-Silicon Debug. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3195992

1 INTRODUCTION

Post-silicon validation is a crucial component of the validation of a modern System-on-Chip (SoC) design, is performed under highly aggressive schedules, and accounts for more than 50% of the validation cost [9, 16].

An expensive component of post-silicon validation is application level *use-case validation*. In this activity, a validator exercises various target usage scenarios of the system (*e.g.*, for a smartphone, playing videos or surfing the Web, while receiving a phone call) and monitors for failures (*e.g.*, hangs, crashes, deadlocks, overflows, etc.). Use case validation forms a key part of *compatibility validation* [8], and often takes weeks to months of validation time. Consequently, it is crucial to determine techniques to streamline this activity.

Each usage scenario involves interleaved execution of several protocols among IPs in the SoC design, *e.g.*, a usage scenario that entails receiving a phone call in a smartphone when the phone is asleep may constitute protocols among the antenna, power management unit, CPU, etc. To debug such a scenario, the validator typically needs to observe and comprehend the messages being sent by the constituent IPs. An effective way to do that is to use *hardware tracing*, where a small set of signals are monitored continuously during system execution.

Unfortunately, the effectiveness of hardware tracing is limited by the signals being selected for tracing. Note that the omission of a critical signal (*e.g.*, a critical interface register) manifests only during post-silicon debug when it is too late for a new silicon spin.

In this paper, we develop a method for message selection that specifically targets use case validation. Given a collection of usage scenarios and the system-level protocols they activate (and the constituent messages), our algorithm computes the messages that are valuable for debug and error localization. We also develop heuristics for maximizing utilization of the available trace observability (trace buffer) in the context of message selection.

There has been significant research on post-silicon signal selection [2, 3, 5, 7, 10]. Most of these approaches analyze the gate level design and optimize a metric called State Restoration Ratio (SRR), that values signal reconstruction ability. However, a high restorability (SRR) of gate level signals may not correspond to crucial message buffers for the application use-cases. In our experiments on a USB controller design, we found that existing signal selection techniques could reconstruct no more than 26% of required interface messages across various design blocks. Analyzing at the application level provides our method the context to select 100% of the messages required for debug. [1] **This underlines the need for a focused approach for message selection that accounts for flows induced during use-case validation**. Further, many of the SRR-based algorithms suffer severely from scalability issues.

To show scalability and viability of our approach, we perform our experiments on a publicly available multicore SoC design OpenSPARC T2 [12]. The design contains several heterogeneous IPs and reflects many complex design features of an industrial SoC design. The scale and complexity is orders of magnitude more than traditional ISCAS89 benchmarks used to demonstrate signal selection techniques. We inject complex and subtle bugs, with each bug symptom taking several hundred observed messages (up to 457 messages) and several hundred thousands of clock cycles (up to 21290999 clock cycles) to manifest. Our analysis shows that we can achieve up to 100% trace buffer utilization (average 98.96%) and up to 99.86% flow specification coverage (average 94.3%). Our messages are able to localize each bug to no more than 6.11% of the total paths that could be explored. Our selected messages helped eliminate up to 88.89% of potential root causes (average 78.89%) and localize to a small set of root causes.

Our method needs a priori definition of system-level protocols at transaction level. Our framework uses protocol formalizations as sequences of transactions or *flows*. There is an increasing trend to generate transaction-level models specifically with formalizations like flows, to enable early validation, prototyping, and software development activities [1, 4, 11, 13]. Our work shows how to leverage this collateral for post-silicon trace selection.

[1] SRR based algorithms typically select flip-flops internal to the design for tracing whereas our method selects interface registers (either incoming or outgoing) for the relevant IPs for tracing.

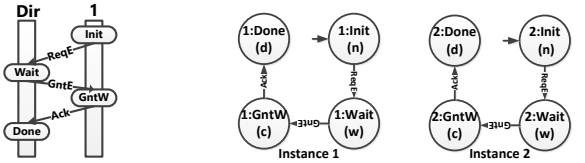
DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3195992

This paper makes three important contributions. First, we exploit available architectural collateral (*e.g.*, messages, transaction flows, etc.) to develop a targeted message selection for hardware tracing targeted towards post-silicon use-case validation. Second, we provide a technique based on mutual information gain to select messages at the application level. Third, in addition to high quality and high information content in selected messages, we make scalability an objective of the post-silicon debug solution. In doing so, we operate at a higher level of abstraction (application level), as opposed to the RTL/gate level signal tracing seen hitherto in literature. We demonstrate post-silicon debug on an industrial scale design, which is a massive engineering effort involving many man months.

2 PRELIMINARIES

Conventions. In SoC designs, a message can be viewed as an assignment of Boolean values to the interface signals of a hardware IP. In our formalization below, we leave the definition of message implicit, but we will treat it as a pair $\langle C, w\rangle$ where $w \in \mathbb{Z}^+$. Informally, C represents the content of the message and w represents the number of bits required to represent C. Given a message $m = \langle C, w\rangle$, we will refer to w as *bit-width of* m, denoted by $width(m)$ or $|m|$.



(a) (b)

Figure 1: 1a shows a *flow* for an exclusive line access request for a toy cache coherence flow [13] along with participating IPs. 1b shows two legally indexed instances of cache coherence flow.

Definition 1. *A* **flow** *is a directed acyclic graph (DAG) defined as a tuple,* $\mathcal{F} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{S}_p, \mathcal{E}, \delta_{\mathcal{F}}, Atom\rangle$ *where* $\mathcal{S}$ *is the set of flow states,* $\mathcal{S}_0 \subseteq \mathcal{S}$ *is the set of initial states,* $\mathcal{S}_p \subseteq \mathcal{S}$ *and* $\mathcal{S}_p \cap Atom = \emptyset$ *is called the set of stop states,* $\mathcal{E}$ *is a set of messages,* $\delta_{\mathcal{F}} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ *is the transition relation and* $Atom \subset \mathcal{S}$ *is the set of atomic states of the flow.*

We use $\mathcal{F}.\mathcal{S}, \mathcal{F}.\mathcal{E}$ etc. to denote the individual components of a flow $\mathcal{F}$. A *stop* state of a flow is its final state after its successful completion. *Atom* is a *mutex* set of flow states *i.e.*any two flow states in *Atom* cannot happen together. Other components of $\mathcal{F}$ are self-explanatory. In Figure 1a, we have shown a toy cache coherence flow along with the participating IPs and the messages. In Figure 1a, $\mathcal{S}$ = {Init, Wait, GntW, Done}, $\mathcal{S}_0$ = {Init}, $\mathcal{S}_p$ = {Done}, *Atom* = {GntW}. Each of the messages in the cache coherence flow is 1 bit wide, hence $\mathcal{E} = \{\langle ReqE, 1\rangle, \langle GntE, 1\rangle, \langle Ack, 1\rangle\}$.

Definition 2. *Given a flow* $\mathcal{F}$*, an* **execution** ρ *is an alternating sequence of flow states and messages ending with a stop state. For flow* $\mathcal{F}$*,* $\rho = s_0\ \alpha_1\ s_1\ \alpha_2\ s_2\ \alpha_3\ \ldots\ \alpha_n\ s_n$ *such that* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}, \forall 0 \leq i < n,\ s_i \in \mathcal{F}.\mathcal{S},\ \alpha_{i+1} \in \mathcal{F}.\mathcal{E},\ s_n \in \mathcal{F}.\mathcal{S}_p$. **Trace** *of an execution* ρ *is defined as* $trace(\rho) = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_n$.

An example of an execution of the cache coherence flow of Figure 1a would be ρ = {*n, ReqE, w, GntE, c, Ack, d*} and $trace(\rho)$ = {*ReqE, GntE, Ack*}.

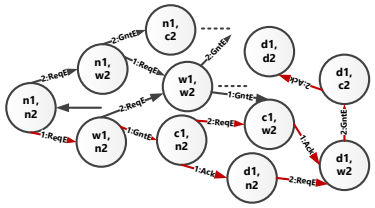


Figure 2: Two instances of cache coherence flow of Figure 1a interleaved

Intuitively, a flow provides a pattern of system execution. A flow can be invoked several times, even concurrently, during a single run of the system. To make precise the relation between an execution of the system with participating flows, we need to distinguish between these instances of the same flow. The notion of *indexing* accomplishes that by augmenting a flow with an "index".

Definition 3. *An* **indexed message** *is a pair* $\alpha = \langle m, i\rangle$ *where* m *is the message and* $i \in \mathbb{N}$*, referred to as the* **index** *of* α*. An* **indexed state** *is a pair* $\hat{s} = \langle s, j\rangle$ *where* s *is a flow state and* $j \in \mathbb{N}$*, referred as the index of* $\hat{s}$*. An* **indexed flow** $\langle f, k\rangle$ *is a flow consisting of indexed message* m *and indexed state* $\hat{s}$ *indexed by* $k \in \mathbb{N}$.

Figure 1b shows two instances of the cache coherence flow of Figure 1a indexed with their respective instance number. In our modeling, we ensure by construction that two different instances of the same flow do not have same indices. Note that in practice, most SoC designs include architectural support to enable *tagging*, *i.e.*, uniquely identifying different concurrently executing instances of the same flow. Our formalization simply makes the notion of tagging explicit.

Definition 4. *Any two indexed flows* $\langle \mathcal{F}, i\rangle, \langle \mathcal{G}, j\rangle$ *are said to be* **legally indexed** *either if* $\mathcal{F} \neq \mathcal{G}$ *or if* $\mathcal{F} = \mathcal{G}$ *then* $i \neq j$.

Figure 1b shows two legally indexed instances of the cache coherence flow of Figure 1a. Indices uniquely identify each instance of the cache coherence flow.

A **usage scenario** is a pattern of frequently used applications. Each such pattern comprises multiple interleaved flows corresponding to communicating hardware IPs.

Definition 5. *Let* $\mathcal{F}, \mathcal{G}$ *be two legally indexed flows. The interleaving* $\mathcal{F} ||| \mathcal{G}$ *is a flow called* **interleaved flow** *defined as* $\mathcal{U} = \mathcal{F} ||| \mathcal{G} = \langle \mathcal{F}.\mathcal{S} \times \mathcal{G}.\mathcal{S}, \mathcal{F}.\mathcal{S}_0 \times \mathcal{G}.\mathcal{S}_0, \mathcal{F}.\mathcal{S}_p \times \mathcal{G}.\mathcal{S}_p, \mathcal{F}.\mathcal{E} \cup \mathcal{G}.\mathcal{E}, \delta_{\mathcal{U}}, \mathcal{F}.Atom \cup \mathcal{G}.Atom\rangle$ *where* $\delta_{\mathcal{U}}$ *is defined as:*

$$i)\ \frac{s_1 \xrightarrow{\alpha} s_1' \wedge s_2 \notin \mathcal{G}.Atom}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \quad and \quad ii)\ \frac{s_2 \xrightarrow{\beta} s_2' \wedge s_1 \notin \mathcal{F}.Atom}{\langle s_1, s_2\rangle \xrightarrow{\beta} \langle s_1, s_2'\rangle}$$

where $s_1, s_1' \in \mathcal{F}.\mathcal{S}, s_2, s_2' \in \mathcal{G}.\mathcal{S}, \alpha \in \mathcal{F}.\mathcal{E}, \beta \in \mathcal{G}.\mathcal{E}$*. Every path in the interleaved flow is an execution of* $\mathcal{U}$ *and represents an interleaving of the messages of the participating flows.*

Rule i) of $\delta_{\mathcal{U}}$ says that if s_1 evolves to the state s_1' when message α is performed and if g has a state s_2 which is not atomic/indivisible, then in the interleaved flow, if we have a state (s_1, s_2), it evolves to state (s_1', s_2) when message α is performed. A similar explanation holds good for Rule ii) of $\delta_{\mathcal{U}}$. For any two concurrently executing legally indexed flow $\mathcal{F}$ and $\mathcal{G}$, $J = \mathcal{F} ||| \mathcal{G}$, for any $s \in \mathcal{F}.Atom$ and for any $s' \in \mathcal{G}.Atom$, $(s, s') \notin J.\mathcal{S}$. If one flow is in one of its atomic/indivisible state, then no other concurrently executing flow can be in its atomic/indivisible state.

Figure 2 shows partial interleaving $\mathcal{U}$ of two legally indexed flow instances of Figure 1b. Since c_1 and c_2 both are atomic state,

state (c_1, c_2) is an illegal state in the interleaved flow. $\delta_{\mathcal{U}}$ and the *Atom* set make sure that such illegal states do not appear in the interleaved flows.

Trace buffer availability is measured in terms of bits thus rendering bit width of a message important. In Definition 6, we define a message combination. Different instances of the same message i.e. indexed messages are not required while computing the bit width of the message combination.

Definition 6. *A* **message combination** $\mathcal{M}$ *is an unordered set of messages. The* **total bit width** W *of a message combination* $\mathcal{M}$ *is the sum total of the bit width of the individual messages contained in* $\mathcal{M}$ *i.e.* $W(\mathcal{M}) = \sum_{i=1}^{k} width(m_i) = \sum_{i=1}^{k} w_i, m_i \in \mathcal{M}, k = |\mathcal{M}|$.

We introduce a metric called **flow specification coverage** to evaluate the quality of a *message combination.*

Definition 7. *In a flow, every transition is labeled with a message. For a given message, the* **visible state** *is defined as the set of flow states reached on the corresponding transition. The* **flow specification coverage of a message combination** *is defined as the union of the visible flow states of all the messages, expressed as a fraction of the total number of flow states.*

Mutual information gain measures the amount of information that can be obtained about one random variable by observing another. The mutual information gain of X relative to Y is given by $I(X;Y) = \sum_{x,y} p(x,y)\ log(p(x,y)/p(x)p(y))$, where $p(x)$ and $p(y)$ are the associated probability mass function for two random variables X and Y respectively.

Maximizing information gain is done in order to increase flow specification coverage during post-silicon debug of usage scenarios. The message selection procedure considers the *message combination* $\mathcal{M}$ for tracing, whereas to calculate information gain over $\mathcal{U}$, it uses *indexed messages.*

Given a set of legally indexed participating flows of a usage scenario, bit widths of associated messages, and a trace buffer width constraint, **our method selects a message combination such that information gain is maximized over the interleaved flow** $\mathcal{U}$ and the **trace buffer is maximally utilized**.

3 MESSAGE SELECTION METHODOLOGY

For the cache coherence flow example of Figure 1a, we assume a trace buffer width of 2 bits and concurrent execution of two instances of the flow. *ReqE, GntE,* and *Ack* messages happen between *1-Dir, Dir-1,* and *1-Dir* IP pairs respectively. *ReqE, GntE,* and *Ack* consist of `req, gnt` and `ack` signal and each of the messages is 1-bit wide. Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values. $C(ReqE) = \mathbb{B}^{|req|}$, $C(GntE) = \mathbb{B}^{|gnt|}$, and $C(Ack) = \mathbb{B}^{|ack|}$ denote respective message contents.

3.1 Step 1: Finding message combinations

In Step 1, we identify all possible message combinations from the set of all messages of the participating flows in a usage scenario.

While we find different message combinations, we also calculate the total bit width of each such combination. Any message combination that has a total bit width less than or equal to the available trace buffer width is kept for further analysis in Step 2[2]. Each such message combination is a potential candidate for tracing.

In the example of Figure 1a, there are 3 messages and $\sum_{k=1}^{3} \binom{3}{k} = 7$ different message combinations. Of these, only one (*ReqE, GntE, Ack*) has a bit width more than trace buffer width (2). We retain the remaining six message combinations for further analysis in Step 2.

[2]For multi-cycle messages, the number of bits that can be traced in a single cycle is considered as the message bit width

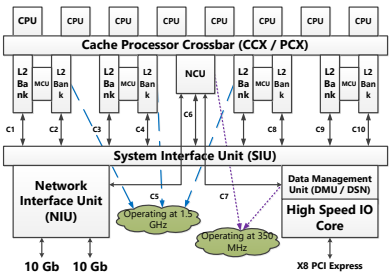


Figure 3: Block diagram of OpenSPARC T2 processor. NCU: Non-cacheable unit, MCU: Memory Controller Unit [12]

3.2 Step 2: Selecting a message combination based on mutual information gain

In this step, we compute the mutual information gain of message combinations computed in step 1 over the interleaved flow. We then select the message combination that has the **highest mutual information gain** for tracing.

We use *mutual information gain* as a metric to evaluate the quality of the selected set of messages with respect to the interleaving of a set of flows. We associate two random variables with the interleaved flow namely X and Y_i. X represents the different states in the interleaved flow i.e. it can take any value in the set $\mathcal{S}$ of the different states of the interleaved flow. Let $\mathcal{M} = \bigcup_i \mathcal{E}_i$ be the set of all possible indexed messages in the interleaved flow. Let Y_i' be a candidate message combination and Y_i be a random variable representing all indexed messages corresponding to Y_i'. All values of X are equally probable since the interleaved flow can be in any state and hence $p_X(x) = \frac{1}{|\mathcal{S}|}$. To find the marginal distribution of Y_i, we count the number of occurrences of each indexed message in the set $\mathcal{M}'$ over the entire interleaved flow. We define $p_{Y_i}(y) = \frac{\text{\# of occurrences of y in flow}}{\text{\# of occurrences of all indexed messages in flow}}$. To find the joint probability, we use the conditional probability and the marginal distribution i.e. $p(x,y) = p(x|y)p(y) = p(y|x)p(x)$. $P(x|y)$ can be calculated as the fraction of the interleaved flow states x is reached after the message $Y_i = y$ has been observed. In other words, $p(x|y)$ is the fraction of times x is reached, from the total number of occurrences of the indexed message y in the interleaved flow i.e. $p_{X|Y_i}(x|y) = \frac{\text{\# occurrence of y in flow leading to x}}{\text{total \# occurrences of y in flow}}$. Now we substitute these values in $I(X;Y)$ to calculate the mutual information gain of the state set X w.r.t Y_i.

In Figure 2, $p_X(x) = \frac{1}{15} \forall x \in \mathcal{S}$. Let $Y_1' = \{GntE, ReqE\}$ be a candidate message combination and $Y_1 = \{$*1:GntE, 2:GntE, 1:ReqE, 2:ReqE*$\}$. For $I(X;Y_1)$, we have $p(y = y_i) = \frac{3}{18}, \forall y_i \in Y_1$. Therefore, $p_{X|Y_1}(x|1:GntE) = \{1/3\ if\ x = (c1, n2), 1/3\ if\ x = (c1, w2), 1/3\ if\ x = (c1, d2)\}$ and $p_{X,Y_1}(x, 1:GntE) = \{1/18\ if\ x = (c1, n2), 1/18\ if\ x = (c1, w2), 1/18\ if\ x = (c1, d2)\}$.

Similarly, we calculate $p_{X,Y_1}(x, 2:GntE)$, $p_{X,Y_1}(x, 1:ReqE)$ and $p_{X,Y_1}(x, 2:ReqE)$. The mutual information gain is given by: $I(X, Y_1) = \sum_{x,y} p(x,y) log p(x,y)/p(x)p(y) = 1.073$.

Similarly, we calculate the mutual information gain for the remaining five message combinations. We then select the message combination that has the highest mutual information gain, which is $I(X, Y_1) = 1.073$ thereby selecting the message combination $Y_1' = \{$ReqE, GntE$\}$ for tracing. Intuitively, in an execution of $\mathcal{U}$ of Figure 2, if the observed trace is $\{$*1:ReqE, 1:GntE, 2:ReqE*$\}$, immediately we are able to localize the execution to two paths shown in red in Figure 2 among many possible paths of $\mathcal{U}$.

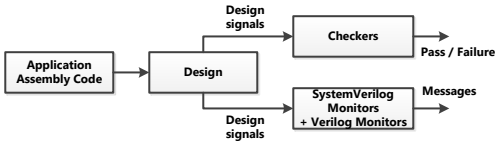


Figure 4: Experimental setup to convert design signals to flow messages

Table 1: Usage scenarios and participating flows in T2. PIOR: PIO Read, PIOW: PIO Write, NCUU: NCU Upstream, NCUD: NCU Downstream and Mon: Mondo Interrupt flow. ✓indicates Scenario i executes a flow j and ✗ indicates Scenario i does not execute a flow j. Flows are annotated with (No of flow states, No of messages)

Usage Scenario	Participating Flows					Patici-pating IPs	Potential root causes
	PIOR (6, 5)	PIOW (3, 2)	NCUU (4, 3)	NCUD (3, 2)	Mon (6, 5)		
Scenario 1	✓	✓	✗	✗	✓	NCU, DMU, SIU	9
Scenario 2	✗	✗	✓	✓	✓	NCU, MCU, CCX	8
Scenario 3	✓	✓	✓	✓	✗	NCU, MCU, DMU, SIU	9

Table 2: Representative bugs injected in IP blocks of OpenSPARC T2. Bug depth indicates the hierarchical depth of an IP block from the top. Bug type is the functional implication of a bug.

Bug ID	Bug depth	Bug category	Bug type	Buggy IP
1	4	Control	wrong command generation by data misinterpretation	DMU
2	4	Data	Data corruption by wrong address generation	DMU
3	3	Control	Wrong construction of Unit Control Block resulting in malformed request	DMU
4	4	Control	Generating wrong request due to incorrect decoding of request packet from CPU buffer	NCU

3.3 Step 3: Packing the trace buffer

Message combinations with the highest mutual information gain selected in Step 2 may not completely fill the trace buffer. To maximize trace buffer utilization, in this step we *pack* smaller message groups which are small enough to fit in the leftover trace buffer width. Usually, these smaller message groups are part of a larger message that cannot be fit into the trace buffer, *e.g.* in OpenSPARC T2, `dmusiidata` is 20 bits wide message whereas `cputhreadid` a subgroup of `dmusiidata` is 6 bits wide. We select a message group that can fit into the leftover trace buffer width, such that the information gain of the selected message combination in union with this smaller message group is maximal. We repeat this step until no more smaller message groups can be added in the leftover trace buffer. Benefits of packing are shown empirically in Section 5.1.

In our running example, the trace buffer is filled up by the set of selected message combination. The flow specification coverage achieved with Y_1' is 0.7333.

4 EXPERIMENTAL SETUP

Design testbed: We primarily use the publicly available OpenSPARC T2 SoC [12] to demonstrate our result. Figure 3 shows an IP level block diagram of T2. Three different usage scenarios considered in our debugging case studies are shown in Table 1 along with participating flows (column 2-6) and participating IPs (column 7). We also use the USB design [15] to compare with other methods that cannot scale to the T2.

Table 3: Trace buffer utilization flow specification coverage and path localization of traced messages for 3 different usage scenarios. FSP Cov: Flow specification coverage (Definition 7), WP: With packing, WoP: Without Packing. 32 bits wide trace buffer assumed.

Case study	Usage Scenario	Trace Buffer Utilization		FSP Cov		Path Localization	
		WP	WoP	WP	WoP	WP	WoP
1	Scenario 1	96.88%	84.37%	99.86%	97.22%	0.13%	3.23%
2						0.31%	6.11%
3	Scenario 2	100%	71.87%	99.69%	93.75%	0.26%	5.13%
4						0.10%	2.47%
5	Scenario 3	100%	93.75%	83.33%	77.78%	0.11%	2.65%

Table 4: Comparison of signals selected by our method with SigSeT [2] and PRNet [7] for the USB design. P: Partial bit

Signal Name	USB Module	Sig SeT	PR Net	Info Gain
rx_data	UTMI line speed	✗	✓	✓
rx_valid		✗	✓	✓
rx_data _valid	Packet decoder	✗	✗	✓
token _valid		✗	✗	✓
rx_data _done		✗	✗	✓
tx_data	Packet assembler	✗	✗	✓
tx_valid		✗	✓	✓
send_token	Protocol engine	✗	✗	✓
token_pid _sel		P	P	✓
data_pid _sel		P	✗	✓

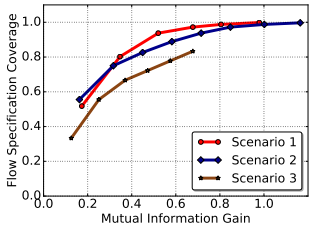


Figure 5: Correlation analysis between *mutual information gain* and *flow specification coverage* for different message combinations for three different usage scenarios.

Testbenches: We used 5 different tests from `fc1_all_T2` regression environment. Each test exercises 2 or more IPs and associated flows. We monitored message communication across participating IPs during simulation and recorded the messages into an output trace file. We use System-Verilog monitors shown in Figure 4 to convert the RTL signals of OpenSPARC T2 into flow messages during execution for our large scale debugging effort.

Bug injection: We created 5 different buggy versions of T2, that we analyze as five different case studies. Each case study comprises 5 different IPs. We injected a total of 14 different bugs across the 5 IPs in each case. The injected bugs follow two sources, i) sanitized examples of communication bugs received from our industrial partners, ii) "bug model" developed at Stanford University in the QED [6] project capturing commonly occurring bugs in an SoC design. A few representative injected bugs are detailed in Table 2. Table 2 shows that the set of injected bugs are complex, subtle and realistic. It took upto **457 observed messages** and upto **21290999 clock cycles** for each bug symptom to manifest. These demonstrate complexity and subtlety of the injected bugs. Following [12] and Table 2, we have identified several potential architectural causes that can cause an execution of a usage scenario to fail. Column 8 of Table 1 shows number of potential root causes per usage scenario.

5 EXPERIMENTAL RESULTS

In this section, we provide insights into our large scale debugging effort of five different (buggy) case studies across 3 usage scenarios of the T2.

Table 5: Selection of important messages by our method

Message	Affecting Bug IDs	Bug coverage	Message importance	Selected	
				Y / N	Usage scenario
m1	8, 33, 36	0.21	4.76	Y	1, 2
m2	8, 33, 34, 36	0.28	3.57	Y	1, 2
m3	33, 36	0.14	7.14	Y	1, 2
m4	8, 29, 33	0.21	4.76	Y	1, 3
m5	18, 33	0.14	7.14	Y	1, 2
m6	-	-		N	-
m7	-	-		Y	1, 3
m8	33	0.07	14.28	Y	2
m9	1, 33	0.14	7.14	N	-
m10	24	0.07	14.28	Y	2
m11	1, 24	0.14	7.14	Y	2
m12	24	0.07	14.28	Y	2
m13	8	0.07	14.28	Y	2
m14	1, 17, 33	0.21	4.76	Y	2
m15	1, 17, 18, 33	0.28	3.57	N	-
m16	1, 17, 18, 33	0.28	3.57	Y	2, 3

5.1 Flow specification coverage and trace buffer utilization

Table 3 demonstrates the value of the traced messages with respect to flow specification coverage (Definition 7) and trace buffer utilization. These are the two objectives for which our message selection is optimized. Messages selected **without packing** achieve **up to 93.75% of trace buffer utilization** with **up to 97.22% flow specification coverage**. **With packing**, message selection achieves **up to 100% of trace buffer utilization** and **up to 99.86% flow specification coverage**. This shows that we can cover most of the desired functionality while utilizing the trace buffer maximally.

5.2 Path localization during debug of traced messages

In this experiment, we use buggy executions and traced messages to show the extent of path localization per bug. Localization is calculated as the fraction of total paths of the interleaved flow. In Table 3, columns 7 and 8 show the extent of path localization. We needed to explore **no more than 6.11% of interleaved flow paths** using our selected messages. With packing, we needed to explore **no more than 0.31% of the total interleaved flow paths** during debugging. Even with packing, subtle bugs like NCU bug of buggy design 3 and buggy design 2 needed more paths to explore.

5.3 Validity of information gain as message selection metric

We select messages per usage scenario. In Figure 5 we analyze the correlation between flow specification coverage and the mutual information gain of the selected messages. Flow specification coverage (Definition 7) **increases monotonically with the mutual information gain** over the interleaved flow of the corresponding usage scenario. This establishes that **increase in mutual information gain corresponds to higher coverage of flow specification**, indicating that mutual information gain is a good metric for message selection.

5.4 Comparison of our method to existing signal selection methods

To demonstrate that existing Register Transfer Level signal selection methods cannot select messages in system level flows, we compare our approach with an SRR-based method [2] and a PageRank based method [7]. **We could not apply existing SRR based methods on the OpenSPARC T2, since these methods are unable to scale. We use a smaller USB design for comparison with our method.** In the USB [15] design we consider a usage scenario consisting of two flows. Table 4 shows that our (mutual information gain based) method selects all of `token_pid_sel`, `data_pid_sel` and other important interface signals for system level debugging. SigSeT, on the other hand selects signals which are not useful for

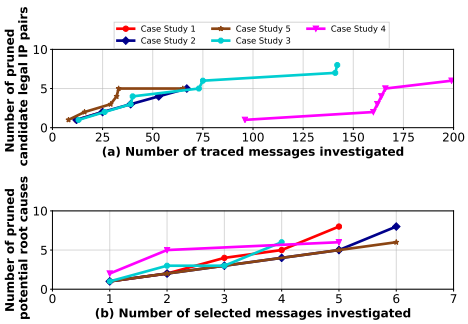


Figure 6: Root causing buggy IP

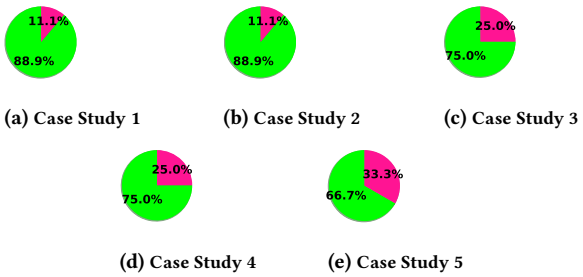


Figure 7: Selected messages-cause pruning distribution for diagnosis. Plausible Cause, Pruned Cause

system level debugging. Our messages are composed of interface signals, and achieve a flow specification coverage of **93.65%**, whereas messages composed of interface signals selected by SigSeT and PRNet have a low flow specification coverage of **9%** and **23.80%** respectively.

5.5 Selection of important messages by our method

For evaluation purposes, we use *bug coverage* as a metric, to determine which messages are important. A message is said to be *affected* by a bug if its value in an execution of the buggy design differs from its value in an execution of the bug free design. Intuitively, if multiple bugs are affecting a message, it is highly likely that message is a part of multiple design paths. The *bug coverage* of a message is defined as the total number of bugs that affects a message, expressed as a fraction of the total number of injected bugs. From debugging perspective, a message is *important* if it is affected by very few bugs implying that the message symptomizes subtle bugs. Table 5 confirms that post-Silicon bugs are subtle and tend to affect no more than 4 messages each. Column 4, 5 and 6 of Table 5 show that our method was able to select important messages from the interleaved flow to debug subtle bugs.

Table 5 shows that message *m15* is affected by four bugs and message *m9* is affected by two bugs, but due to their size being wider than 32 bits trace buffer, our method does not select them.

5.6 Effectiveness of selected messages in debugging usage scenarios

Every message is sourced by an IP and reaches a destination IP. Bugs are injected into specific IPs (Table 2). During debug, sequences of IPs are explored from the point a bug symptom is observed, to find the buggy IP. An IP pair (<source IP, destination IP>) is *legal* if a message is passed between them. We use the number of legal IP pairs investigated during debug as a metric for selected messages. Table 6 shows that we investigated **an average of 54.67%** of the

Table 6: Diagnosed root causes and debugging statistics for our case studies on OpenSPARC T2.

Case Study	No of. Flows	Legal IP Pairs	Legal IP pairs investigated	Messages investigated	Root caused architecture level function
1	3	12	5	25	An interrupt was never generated by DMU due to wrong interrupt generation logic
2	3		6	67	Wrong interrupt decoding logic in NCU / Corrupted interrupt handling table in NCU
3	3	10	8	142	Malformed CPU request from Cache Crossbar to NCU / Erroneous CPU request decoding logic of NCU
4	3		6	199	Erroneous interrupt deque logic after interrupt is serviced
5	4	12	5	65	Erroneous decoding logic of CPU requests in memory controller

Table 7: Representative potential root causes for one case study. Rest of the root causes are omitted due to lack of space. Remaining case studies are available in [14]

Selected Messages	Potential Causes	Potential Implication
`reqtot,grant, mondoacknack, siincu, piowcrd dmusiidata. cputhreadid siincu,`	**1.** Mondo request forwarded from DMU to SIU's bypass queue instead of ordered queue	**1.** Mondo interrupt not serviced
	2. Invalid Mondo payload forwarded to NCU from DMU via SIU	**2.** Interrupt assigned to wrong CPU ID and Thread ID
	3. Non-generation of Mondo interrupt by DMU	**3.** Computing thread fetches operand from wrong memory location

total legal IP pairs, implying that our selected messages help us focus on a fraction of the legal IP pairs.

To debug a buggy execution, we start with the traced message in which a bug symptom is observed and backtrack to other traced messages. The choice of which traced message to investigate is pseudo-random and guided by the participating flows.

Figure 6(a) plots the number of such investigated traced messages and the corresponding candidate legal IP pairs that are eliminated with each traced message. Figure 6(b) shows a similar relationship between the traced messages and the candidate root causes, *i.e.* the architecture level functions that might have caused the bug to manifest in the traced messages. Both graphs show that with more traced messages, more candidate legal IP pairs as well as candidate root causes are progressively eliminated. This implies that every one of our traced messages contributes to the debug process.

Figure 7 shows that traced messages were able to prune out a large number of potential root causes in all five case studies. Our traced messages pruned out an **average of 78.89% (max. 88.89%)** of candidate root causes.

5.7 Debugging case study

It is illuminating to understand the debugging process for one case study to appreciate the role of the selected messages.

Symptom: In this experiment we used traced messages from Table 7. The simulation failed with an error message *FAIL: Bad Trap*.

Debug with selected messages: We consider bug symptom causes of Table 7 to debug this case. From the observed trace messages, `siincu` and `piowcrd`, we identify NCU got back correct credit ID at the end of the PIO read and PIO write operation respectively. This rules out two causes out of 9. However, we cannot rule out causes related to PIO payload since a wrong payload may cause computing thread to catch *BAD Trap* by requesting operand from wrong memory location. Absence of trace messages `mondoacknack` and `reqtot` implies that NCU did not service any Mondo interrupt request and SIU did not request a Mondo payload transfer to NCU respectively. Further, there is no message corresponding to `dmusiidata.cputhreadid` in the trace file, implying that DMU was never able to generate a Mondo interrupt request for NCU to process. This rules out all causes except cause **3** (**1 cause out of 9, pruning of 88.89% of possible causes**) to explore further to find the root cause.

Root Cause: From [12], we note that an interrupt is generated only when DMU has credit and all previous DMA reads are done. We found no prior DMA read messages and DMU had all its credit available. Absence of `dmusiidata` message correct CPUID and ThreadID implies that DMU never generated a Mondo interrupt request. This makes DMU a plausible location of the root cause of the bug.

6 CONCLUSIONS

We have demonstrated the scalability and effectiveness of our trace message selection approach on the OpenSPARC T2 processor for root causing bugs in system-level usage scenarios. This is the most large-scale application of a hardware signal tracing approach in published literature.

REFERENCES

[1] Yael Abarbanel, Eli Singerman, and Moshe Y. Vardi. 2014. Validation of SoC Firmware-Hardware Flows: Challenges and Solution Directions. In *The 51st Annual DAC '14, San Francisco, CA, USA, June 1-5, 2014*. 2:1–2:4. https://doi.org/10.1145/2593069.2596692

[2] Kanad Basu and Prabhat Mishra. 2011. Efficient trace signal selection for post silicon validation and debug. In *VLSI Design (VLSI Design), 2011 24th International Conference on*. IEEE, 352–357.

[3] Debapriya Chatterjee, Calvin McCarter, and Valeria Bertacco. 2011. Simulation-based signal selection for state restoration in silicon debug. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 595–601.

[4] Ranan Fraer, Doron Keren, Zurab Khasidashvili, Alexander Novakovsky, Avi Puder, Eli Singerman, Eran Talmor, Moshe Y. Vardi, and Jin Yang. 2014. From visual to logical formalisms for SoC validation. In *Twelfth ACM/IEEE MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*. 165–174. https://doi.org/10.1109/MEMCOD.2014.6961855

[5] Ho Fai Ko and Nicola Nicolici. 2009. Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug. *IEEE Trans. on CAD of Integrated Circuits and Systems* 28, 2 (2009), 285–297. https://doi.org/10.1109/TCAD.2008.2009158

[6] D Lin, T Hong, Y Li, S Kumar, F Fallah, N Hakim, DS Gardner, S Mitra, et al. 2014. Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 33, 10 (2014), 1573–1590.

[7] Sai Ma, Debjit Pal, Rui Jiang, Sandip Ray, and Shobha Vasudevan. 2015. Can't See the Forest for the Trees: State Restoration's Limitations in Post-silicon Trace Signal Selection. In *Proceedings of ICCAD 2015, Austin, TX, USA, November 2-6, 2015*. 1–8. https://doi.org/10.1109/ICCAD.2015.7372542

[8] Prabhat Mishra, Ronny Morad, Avi Ziv, and Sandip Ray. 2017. Post-Silicon Validation in the SoC Era: A Tutorial Introduction. *IEEE Design & Test* 34, 3 (2017), 68–92. https://doi.org/10.1109/MDAT.2017.2691348

[9] P. Patra. 2007. On the Cusp of a Validation Wall. *IEEE Design and. Test of Computers* 24, 2 (2007), 193–196.

[10] Kamran Rahmani, Prabhat Mishra, and Sandip Ray. 2014. Efficient trace signal selection using augmentation and ILP techniques. In *Fifteenth ISQED 2014, Santa Clara, CA, USA, March 3-5, 2014*. 148–155. https://doi.org/10.1109/ISQED.2014.6783318

[11] Eli Singerman, Yael Abarbanel, and Sean Baartmans. 2011. Transaction based pre-to-post silicon validation. In *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*. 564–568. https://doi.org/10.1145/2024724.2024855

[12] SPARCT2 2008. OpenSPARC T2. (2008). http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html.

[13] Murali Talupur, Sandip Ray, and John Erickson. 2015. Transaction Flows and Executable Models: Formalization and Analysis of Message passing Protocols. In *FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. 168–175.

[14] Tech Report 2017. Zoom Out and See Better: Scalable Message Tracing for Post-Silicon SoC Debug. (2017). http://hdl.handle.net/2142/98857.

[15] USB 2008. USB 2.0. (2008). http://opencores.org/project,usb.

[16] S. Yerramilli. 2006. Addressing Post-Silicon Validation Challenge: Leverage Validation and Test Synergy. In *Keynote, Intl. Test Conf.*