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.

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.

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ’18, June 24–29, 2018, San Francisco, CA, USA © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-5700-5/18/06... $15.00 https://doi.org/10.1145/3195578.3195992

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.
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 \( (c, w) \) 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 \( m = (c, w) \), we will refer to \( w \) as bit-width of \( m \), denoted by \( \text{width}(m) \) or \( |m| \).

Definition 1. A flow is a directed acyclic graph (DAG) defined as a tuple, \( F = (S, S_0, S_p, E, \delta_F, \text{Atom}) \) where \( S \) is the set of flow states, \( S_0 \subseteq S \) is the set of initial states, \( S_p \subseteq S \) and \( S \cap \text{Atom} = \emptyset \) is called the set of stop states, \( E \) is a set of messages, \( \delta_F \subseteq S \times E \times S \) is the transition relation and \( \text{Atom} \subseteq S \) is the set of atomic states of the flow.

We use \( \mathcal{F}, \mathcal{S}, \mathcal{E} \) etc. to denote the individual components of a flow \( F \). A stop state of a flow is its final state after its successful completion. \( \text{Atom} \) is a mutex set of flow states i.e.any two flow states in \( \text{Atom} \) cannot happen together. Other components of \( F \) 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, \( \mathcal{S} = \{\text{Init}, \text{Wait}, \text{GntW}, \text{Done}\} \), \( S_0 = \{\text{Init}\} \), \( S_p = \{\text{Done}\} \), \( \text{Atom} = \{\text{GntW}\} \). Each of the messages in the cache flow is 1 bit wide, hence \( E = \{\text{ReqE}, \text{GntE}, \text{Ack}\} \).

Definition 2. Given a flow \( F \), an execution \( \rho \) is an alternating sequence of flow states and messages ending with a stop state. For flow \( F \), \( \rho = s_0 a_1 s_1 a_2 s_2 a_3 \ldots a_n s_n \) such that \( s_i = \frac{a_i s_{i-1}}{a_{i+1}} s_{i+1}, i = 0 \implies n \in \mathbb{F} S, a_{i+1} \in \mathcal{E}, s_{n} \in \mathcal{S}_p \). Trace of an execution \( \rho \) is defined as \( \text{trace}(\rho) = a_1 a_2 a_3 \ldots a_n \).

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

Figure 2 shows partial interleaving \( U \) 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. \(\delta_{ij}\) 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 \(M\) is an unordered set of messages. The total bit width \(W\) of a message combination \(M\) is the sum total of the bit width of the individual messages contained in \(M\) i.e. \(W(M) = \sum_{i=1}^{k} \text{width}(m_i) = \sum_{i=1}^{k} w, m_i \in M, k = |M|\).

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 \(I(X; Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}\), 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 \(M\) for tracing, whereas to calculate information gain over \(U\), 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 \(U\) 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: \(\text{ReqE, GntE}\), and \(\text{Ack}\) messages happen between 1-Dir, Dir-1, and 1-Dir IP pairs respectively. \(\text{ReqE, GntE, and Ack}\) consist of req, gnt and ack signal and each of the messages is 1 bit wide. Let \(B = \{0, 1\}\) be the set of Boolean values. \(C(\text{ReqE}) = B^{[\text{req}]}, C(\text{GntE}) = B^{[\text{gnt}]}, \) and \(C(\text{Ack}) = B^{[\text{ack}]}\) 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. Each such message combination is a potential candidate for tracing.

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

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

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 \(Y\). \(X\) represents the different states in the interleaved flow i.e. it can take any value in the set \(S\) of the different states of the interleaved flow. Let \(M = \bigcup_{i} E_i\) be the set of all possible indexed messages in the interleaved flow. Let \(Y_i\) be a candidate message combination and \(Y\) 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 \(p(x) = \frac{1}{18}\). To find the marginal distribution of \(Y_i\), we count the number of occurrences of each indexed message in the set \(M^i\) over the entire interleaved flow. We define \(p_X(y) = \frac{\text{# occurrences of all indexed messages in flow}}{\text{total number of occurrences of all indexed messages in flow}}\). 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 \(x\) that \(y\) has been observed. In other words, \(p(x|y)\) is the fraction of \(x\) that \(y\) has been observed.

Similarly, we calculate \(p_X(y|x) = \frac{\text{# occurrence of } y \text{ in flow leading to } x}{\text{total # occurrences of } y \text{ in flow}}\). To calculate the mutual information gain of the state set \(X\) w.r.t. \(Y_i\),

In Figure 2, \(p_X(y) = \frac{1}{18} \forall x \in S\). Let \(Y_i = \{\text{GntE, ReqE}\}\) be a candidate message combination and \(Y_1 = \{\text{GntE, 2GntE, 1ReqE, 2ReqE}\}\). For \(X, Y_1\), we have \(p(y = y_1) = \frac{1}{18}, y_1 \in Y_i\). Therefore, \(p_X|Y_1(y|x) = \{\text{GntE} = (1/3 \text{ if } x = (c_1, n_2), 1/3 \text{ if } x = (c_1, w_2), 1/3 \text{ if } x = (c_1, d_2)\}\) and \(p_X|Y_1(y|x) = \{\text{GntE} = (1/18 \text{ if } x = (c_1, n_2), 1/18 \text{ if } x = (c_1, w_2)\}\). Similarly, we calculate \(p_X|Y_1(y|x) = \{\text{GntE, 2GntE, 1ReqE, 2ReqE}\}\). The mutual information gain is given by:

\[
I(X; Y_i) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} = 1.073
\]

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 \(Y_1 = \{\text{GntE, ReqE}\}\) for tracing. Intuitively, in an execution of \(U\) of Figure 2, if the observed trace is \(\{\text{1ReqE, 1GntE, 2ReqE}\}\), immediately we are able to localize the execution to two paths shown in red in Figure 2 among many possible paths of \(U\).
Figure 4: Experimental setup to convert design signals to flow messages

Table 1: Usage scenarios and participating flows in T2. PIOR: P1O Read, PIOW: P1O Write, NCUU: NCU Upstream, NCU: NCU Downstream and Mon: Mondo Interrupt flow. ✓ indicates Scenario i executes a flow j and X indicates Scenario i does not execute a flow j. Flows are annotated with (No of flow states, No of messages)

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.

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 is a subgroup of dmusiidata with 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.

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.

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

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_a111T2 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 up to 457 observed messages and up to 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 | Affected Bug IDs | Bug coverage | Message importance | Selected Usage scenario | Usage | Messages | Message symptomizes subtle bugs
|---------|------------------|--------------|--------------------|------------------------|-------|----------|--------------------------------|
| m1 | 8, 35, 36 | 0.21 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m2 | 8, 35, 36, 39 | 0.20 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m3 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m4 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m5 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m6 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m7 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m8 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m9 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m10 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m11 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m12 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m13 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m14 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m15 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes
| m16 | 8, 29, 35 | 0.24 | Y | 93.75% of trace buffer utilization | **Y** | 1 | Yes

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 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
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. 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.

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


