This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Managing Trace Summaries to Minimize Stalls During Postsilicon Validation

Sandeep Chandran, Preeti Ranjan Panda, Smruti R. Sarangi, Ayan Bhattacharyya, Deepak Chauhan, and Sharad Kumar

***Abstract*— On-chip trace buffers are increasingly being used for at-speed debug during postsilicon validation. The limited size of these buffers results in their frequent overflowing. In scenarios when such overflowing is not desirable, the chip is stalled, and the state data recorded in these buffers are transferred off-chip. Such frequent stalling significantly impedes efficient debugging. We propose a novel scheme to minimize the number of such stalls using a portion of the trace buffer to also store summaries of trace messages. We describe an *overlapped* trace buffer architecture that uses a reduced number of ports to capture tapered summaries where both detailed and summary versions of traces are stored simultaneously. We propose a simple hardware structure to generate two kinds of trace summaries—*spatial* and *temporal*—as specified by the validation engineer. We introduce a *storage specification language* that allows the validation engineer to unambiguously specify the information to be captured in these summaries to the debug hardware. We demonstrate that our proposal significantly reduces the number of stalls for off-chip transfer of captured traces in four bug scenarios that are representative of different classes of bugs encountered during postsilicon validation.**

***Index Terms*— Design-for-debug architecture, postsilicon validation, trace summaries.**

I. Introduction

THE increasing complexity of processors and systems on chip (SoCs), coupled with aggressive time-to-market deadlines, have forced chip manufacturers to adopt well-planned strategies for postsilicon validation [1], [2]. At-speed debugging using on-chip trace buffers has become the *de facto* methodology for postsilicon validation because it quickly localizes the observed error in long running test cases. However, frequent overflowing of the limited sized trace buffers requires the entire chip to be halted in order to transfer its contents off-chip. Such frequent off-chip transfers are time consuming and constitute a major impediment to efficient debugging.

Manuscript received May 15, 2016; revised October 8, 2016 and December 12, 2016; accepted January 9, 2017. This work was supported by a research grant from Freescale and Semiconductor Research Corporation under Grant 2014-TJ-2528.

S. Chandran, P. R. Panda, and S. R. Sarangi are with the Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi 110016, India (e-mail: sandeep@cse.iitd.ac.in; panda@cse.iitd.ac.in; srsarangi@cse.iitd.ac.in).

A. Bhattacharyya was with the Indian Institute of Technology Delhi, New Delhi 110016, India.

D. Chauhan and S. Kumar are with NXP Semiconductors, Noida 201301, India (e-mail: d.kumar@nxp.com; sharad.kumarg@nxp.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2657604

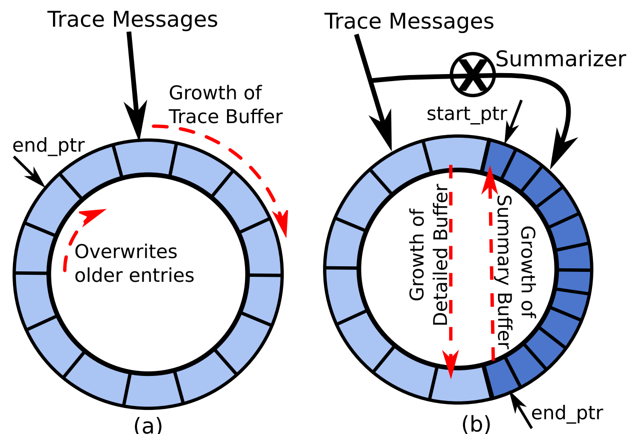

Fig. 1. High-level schematic of the proposed scheme. (a) Trace buffer organized as a circular queue. (b) Flexible runtime partition into detailed and summary buffers.

Several works have attempted to improve the efficiency of at-speed debugging by reducing the amount of redundant and irrelevant information captured in the trace buffer. These works have focused on both: 1) design changes, where redundancy in the information captured is reduced by carefully choosing signals for tracing [3], [4], and 2) runtime techniques, where only traces from relevant regions of execution are captured [5], [6] and compressed before storing into the buffer [7], [8].

We propose a novel runtime approach to improve the efficiency of at-speed debug, by storing the summaries of trace messages before overwriting them with incoming trace messages, and transfer only these summaries off-chip. The validation engineer can use the recent on-chip detailed traces to reconstruct the erroneous state and the off-chip summaries to infer the activity sequence that led to the erroneous state. Since the summaries are terser than detailed traces, the activity history for longer durations can be gathered through a reduced number of stalls.

We first propose a novel storage mechanism that allows the validation engineer to choose the amount of space for storing summaries. We achieve this by reusing the existing trace buffer through simple hardware extensions as shown in Fig. 1. Fig. 1(a) shows the default trace buffer pictured as a circular queue. In our proposed architecture pictured in Fig. 1(b), the space is partitioned at run time (by configuring the start and end pointers) into multiple circular buffers, to simultaneously capture both detailed and summary traces.

We allow the validation engineer to specify the information that is currently of interest depending on: 1) the debug

1063-8210 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

scenario and 2) the relative time of occurrence of events. These input specifications are used to generate trace summaries by discarding any information captured in the trace stream that is no longer relevant. The summary obtained by discarding irrelevant information present within a trace message is called *spatial summary*, and the summary obtained by discarding irrelevant information that is spread across several trace messages is called *temporal summary*. We propose a detailed design of a trace summarizer that can generate the spatial and temporal summaries. We also present a specification language called *storage specification language (SSL)*, which the validation engineer can use to specify the information that is of interest to the hardware unambiguously. These specifications are translated into configurable parameters of the proposed hardware and are transferred through JTAG ports to various configuration registers.

Using a set of four case studies that are representative of different kinds of bugs encountered during postsilicon validation, we show that such a tailored trace capture avoids the need to stall the chip to transfer the contents of the trace buffer off-chip when temporal summaries are captured. We observe a reduction in the number of stalls by up to 63% when only spatial summaries are captured.

The rest of this paper is organized as follows. Section II outlines the prior research. Section III introduces the specification framework of SSL and presents a detailed discussion on the information captured in spatial and temporal summaries. Section IV discusses tapered storage and trace buffer architectures to achieve it. Section V describes the hardware design of the trace summarizer that can generate spatial and temporal summaries, and Section VII explains the case studies and results. Our conclusions are given in Section VIII.

II. Related Work

Several works have attempted to reduce the number of stalls through two broad approaches: 1) increasing the utilization of the existing buffer through intelligent trace capture [4], [6] and 2) increasing the size available to capture traces by reusing architectural components such as caches [9], [10], so that the adverse effects of capturing irrelevant information are reduced. Our strategy falls under the former approach, and is complementary to the latter.

Traces with reduced information captured in them have been found useful to quickly localize bugs in the past [11]–[13]. In these proposals, traces with reduced information or execution signatures are used to broadly identify regions of execution that deviate from the expected behavior initially. Detailed traces are then captured only in the regions of execution identified above. Our work is orthogonal to this because we characterize deviations using sequences of related events instead of time intervals.

Several previous works have proposed to use traces with varying amounts of information captured in them for use in transaction-based debugging [14]–[16]. Under these approaches, the functionality of the chip is captured as a sequence of messages exchanged over the interconnect. Trace messages are generated at different granularities of the message exchange and transferred off-chip for further analysis on compliance to expected behavior. However, these proposals determine the information to be captured at each granularity during design time and hence cannot adapt to the changing requirements during debugging. In contrast to this, our proposal can generate trace messages at different granularities that are defined at runtime and store messages of different granularities in the same trace buffer. Moreover, unlike the above proposals that specifically monitor transactions over the interconnect, our technique is applicable to debug the functionality of any module.

TABLE I

Fields in Instruction Trace of LEON3 SoC

Field	Symbol	Bits	Name
1	MCI	126	Multi-cycle instruction
2	TIME	125:96	Time tag
3	LDST	95:64	Load/Store parameters
4	PC	63:34	Program counter
5	TRAP	33	Instruction trap
6	ERR	32	Processor error mode
7	OP	31:0	Opcode

Another area of research that uses a sequence of events for debugging is assertion-based verification. These techniques define invariants that must be satisfied at all times using a sequence of events, with strict timing constraints between the events. A violation of an invariant becomes the starting point for further investigation [17]–[19]. Several works have been proposed in the past under assertion-based verification to synthesize hardware from specifications of invariants [17], [18]. The hardware thus synthesized is embedded into the chip at design-time to check compliance to these invariants during regular operation. More recent works have used these invariants as trigger conditions to capture traces into the trace buffer [19]. Our work differs from these proposals because the trigger conditions are specified at runtime in our proposed design. This allows the temporal conditions for triggering trace summarization and storage to evolve, as more insights into the bug are available.

Other works have demonstrated the use of aggregates of event occurrences captured by performance counters to debug the Core 2 Duo processor [2]. Our technique does not use summaries in the form of aggregates because identifying information, which is essential to reconstruct the sequence of events that led to an erroneous state, is not retained in such summaries. However, our proposed hardware can be extended to generate aggregate summaries as well through minor modifications.

Our proposal is the first work to our knowledge that supports debugging multiple classes of bugs by offering flexibility to: 1) choose the information to be captured in each trace message and 2) capture an arbitrary mix of detailed traces and their summaries to debug observed errors.

III. Trace Summaries

A. Background

We explain different types of trace summaries generated by our proposed hardware and their benefits through the instruction trace generated by an off-the-shelf LEON3 SoC [20]. Table I shows all the information captured in the instruction

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

trace, and is representative of the state-of-the-art in at-speed debug architectures for large SoCs. Only high-level features of each module (such as `PC` or `opcode` in the case of a core) are traced in case of large SoCs so as to quickly localize the bug to a module or specific interaction between modules. Further debugging proceeds using conventional methods such as interactive run–stop debugging, logic analyzers, and emulation.

1) Sample Debug Scenario: We use the following debug scenario to discuss our proposed enhancements: the validation engineer has established through previous attempts that timing issues occur when an instruction at a particular `PC` follows a multicycle operation within two cycles and further suspects that the state of the flip-flop holding the status of `trap` is responsible for the inexplicable behavior. Therefore, at present, the primary interest is only in understanding the behavior of `trap` in relation to `PC` and `opcode` under the aforementioned timing constraint between `PC` and `opcode`. Every other information present in the trace is irrelevant at this juncture of debugging the error.

B. Storage Specification Language

In our proposed methodology, the validation engineer expresses the required information to the debug hardware using the *SSL*. The debug hardware uses the information passed on by the validation engineer in the specification to discard information from the detailed traces to generate summaries.

The grammar of the SSL is given below.

```
SSL_Statement ::= Action | Assignment
Action ::=
  store {Signals | all} Timer {Condition}
  | notify {constant} when {Condition}
Timer ::= when | from | until
Assignment ::= label := Sequence
Signals ::= trace_signal, Signals |
    trace_signal
Condition ::= Sequence | NOT(Sequence)
Sequence ::= Boolean_Exp ; Sequence
  | {Boolean_Exp} TEMPORAL_OP {Sequence}
  | {Sequence} | Boolean_Exp
Boolean_Exp ::= Simple_Exp Set_Clause
    Rep_Clause
Set_Clause ::= \set Signals\ | ϵ
Rep_Clause ::= [*Rep_Count] | ϵ
Simple_Exp ::=
  Boolean_term BINARY_OP Simple_Exp
  | Boolean_term
Boolean_term ::= true | false | Event | label
  | !Boolean_term | (Simple_Exp)
Event ::= (trace_signal COMP_OP Rvalue)
Rep_Count ::= constant | constant:constant
Rvalue ::= constant | (constant,constant)
  | trace_signal'
BINARY_OP ::= and | or
TEMPORAL_OP ::= | | && | within
COMP_OP ::= == | != | >= | <= | > | < | in
```

An SSL statement can either be an action (store/notify) statement or an assignment. The store statement indicates the debug hardware to store the list of trace signals specified by `Signals` as indicated by `Timer`. The timer allows for two modes: 1) sporadic mode and 2) continuous mode. Under the sporadic mode, the trace signals available at the inputs are written into the trace buffer only when the `Condition` holds, and the corresponding store statements use the *when* construct. The store statements for the continuous mode use the *from* and *until* constructs to mark the trace boundaries.

Similarly, an assignment permits the specification of intermediate expressions, which makes SSL store statements more readable. For every occurrence of `label`, the LHS of the operator `:=` is replaced with the RHS to get the final store statement.

The `Condition` can either be the `label` or a possibly aggregate temporal sequence, where individual sequences are boolean expressions of events, connected using a temporal operator. Each boolean expression can be associated with a repetition operator `[*low:high]`, `[*count]` that specifies the range for which the boolean expression must hold true starting from the current cycle. Four operators `;` (consecutive sequences), `|` (alternative sequences), `&&` (sequences occur and have identical start and end cycles), and `within` (sequence inclusion) are considered temporal operators in SSL (taken from SERE [21]). There are two other operators in SSL: 1) a `NOT()` operator that negates the result of the `Condition` and 2) a `set` operator that supports runtime definition of events (discussed later in this section).

An example SSL specification for the aforementioned debug scenario when only on-chip event triggers are used to filter irrelevant information is shown below. Fig. 2(a) shows the information stored into the trace buffer corresponding to the input specification.

```
store {all} from {(MCI==1)}
store {all} until {(PC==0xabcd1234)}
```

Since event triggers only use the information present in the traces to either start or stop tracing, but do not discard information at runtime, constructs other than `all` are not supported by them.

C. Spatial Summary

A spatial summary is generated by selectively filtering out irrelevant information present within a particular trace message. This feature of a debug hardware overcomes the shortcoming of standard event triggers that forced it to capture unnecessary fields along with relevant information. This is achieved by allowing the validation engineer to specify the fields of interest to the debug hardware through the SSL specification as shown below. Fig. 2(b) shows the information that will be captured as part of the spatial summary generated for the debug scenario under consideration for the input specification.

```
store {MCI,PC,TRAP} when {true[*MAX]}
```

An area-efficient hardware design to achieve such runtime filtering is discussed in Section V-A.

Filtering out irrelevant information can significantly reduce the number of stalls when used in conjunction with event triggers where only relevant fields of trace generated within the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

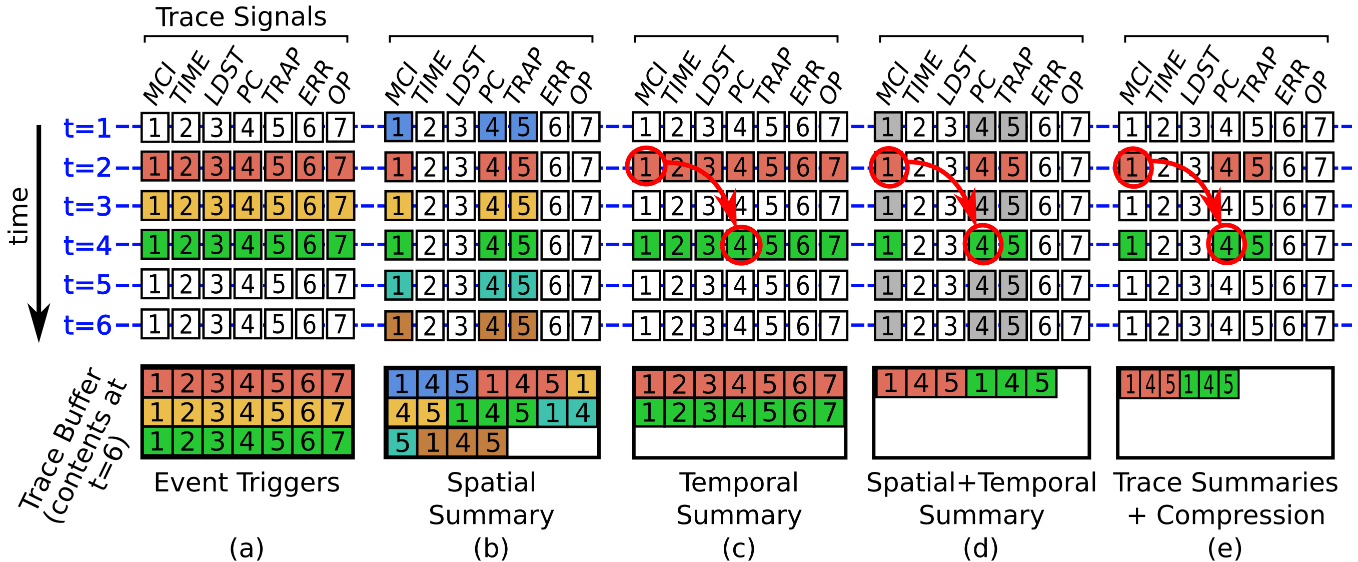

Fig. 2. (a) Data captured by event triggers. (b) Data captured by a spatial summary that discards fields other than 1, 4, and 5. (c) Data captured only when a specified temporal relationship between fields 1 and 4 is satisfied. (d) Reduction in captured information when a combination of spatial and temporal summaries is used. (e) Compressing the captured contents further.

regions of activity demarcated by event triggers are captured for storage into the trace buffer. The SSL specification that leverages a combination of event triggers and online filters is shown below.

```
store {MCI,PC,TRAP} from {(MCI==1)}
store {MCI,PC,TRAP} until {(PC==0xabcd1234)}
```

D. Temporal Summary

The combination of event triggers and online filters can still capture some irrelevant information within the trace buffer. This happens because the debug hardware stores execution traces as and when they are generated. Such eager storage into the trace buffer leads to capturing irrelevant information either when only a part of an event sequence of interest occurs, but not the complete sequence, or when the entire sequence occurs, but the timing relationship between the events does not hold. An example of this is the trace sequence stored into the trace buffer when the instruction at `0xabcd1234` does not occur within two cycles of the multicycle instruction. The region of execution thus demarcated between the multicycle operation and the instruction at `0xabcd1234` could potentially span several thousand cycles, during which the SoC may stall multiple times to dump traces off-chip.

Temporal summary is generated by discarding information that is deemed irrelevant after examining compliance to the specified temporal property, instead of eagerly storing the traces into the trace buffer. All the trace messages that resulted in triggering a constituent event of the specified sequence are written into the trace buffer as part of the temporal summary. The intervening trace messages that did not trigger an event of interest are discarded as the information contained in them is irrelevant. The property of interest captures both the sequence of events and the timing relationship between the constituent events, and is represented using a temporal expression. Fig. 2(c) shows the temporal summary stored into the trace buffer for the SSL statement shown below.

```
example_1 := (MCI==1);
  {{(PC==0xabcd1234)}|{true;(PC==0xabcd1234)}}

store {all} when {example_1}
```

`example_1` represents a sequence that begins when `MCI` is 1 indicating the occurrence of a multicycle instruction. The second term captures the scenario where `PC` takes the value of `0xabcd1234` within the next two cycles. This sequence captures cases when an instruction at a particular PC follows a multicycle instruction within two cycles. The second statement indicates that the detailed trace messages must be stored as and when this sequence is observed.

Fig. 2(d) shows the benefits of using a combination of spatial and temporal summaries as specified by the SSL statement given below.

```
store {MCI, PC, TRAP} when {example_1}
```

1) Runtime Definition of Events: There may be debug scenarios where temporal relationships between the field values in the trace messages need to be checked. An example scenario may be that the validation engineer is interested only in cases where the tag of a read request issued does not match that of the response or if the response itself is delayed. In this case, the debug hardware should check for violations against all the values that the tag field takes. In this scenario, the value against which an event should be triggered changes over time as the execution proceeds. We propose two constructs that specify when to update the value against which an event is triggered and what the new value should be, unambiguously to the debug hardware.

1) `set` *Command*: This command is used to dynamically update the value against which the trigger condition

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

inside the conventional event trigger is set. The event triggers would then trigger events when the input field takes the most recently updated value. This command is always associated with the occurrence of an event and is treated as a side effect to the occurrence of the event. This command is delimited using a pair of "\" symbols. The value updated by `set` is the corresponding value of the field in the trace message that triggered the condition (MCI==1).

2) `′` *Suffix*: This symbol is suffixed to a field name to denote the value that was used to update the event trigger most recently. This is useful to statically define events in SSL statement even when the exact value of the field is not known beforehand. For example, `PC′` refers to the value that was used to configure the event trigger most recently through a `set` command.

The debug scenario considered above is captured in the SSL statement shown below, and uses the above operators.

```
example_2 := (REQ_R==1)\set REQ_TAG\[*0:25];
       (RESP_VALID==1) and (RESP_TAG==REQ_TAG′)

store {all} when {NOT(example_2)}
```

Fig. 2(e) illustrates the benefits of further compressing the captured traces. It also shows that the proposed summaries are complementary to runtime techniques such as compression [8] that aim to reduce stalls without discarding irrelevant information.

In the absence of suitable support from the debug hardware to generate trace summaries, the proposed manipulations such as filtering of information from trace messages and checking for compliance to temporal properties would be done during off-chip analysis [15], by which time significant resources would have already been invested in storing and transferring the traces off-chip. Not only would this contribute to delays in the postprocessing algorithms, but also the resulting stalls could interfere with the bug detection process.

IV. Architectural Considerations

A. Tapered Storage

When debugging an observed error, the validation engineer uses two types of information: 1) an activity log that gives a quick overview of the execution and helps assess if the execution proceeds as expected and 2) changes made to the internal state just prior to the execution deviating from the expected behavior, so as to determine the cause of such deviation. A debug methodology is defined by the mechanism it uses to capture these two kinds of information.

We observe that the activity log is a subset of the detailed state information, and therefore, capturing only the detailed state information can help create the activity log [22]. However, this may lead to a significant increase in trace volume. Therefore, we propose to use summaries of detailed trace messages corresponding to the expected execution path. Trace volume is reduced through: 1) summarizing only the activity that occurred sufficiently in the past and 2) summarizing only the trace messages that follow specified invariants.

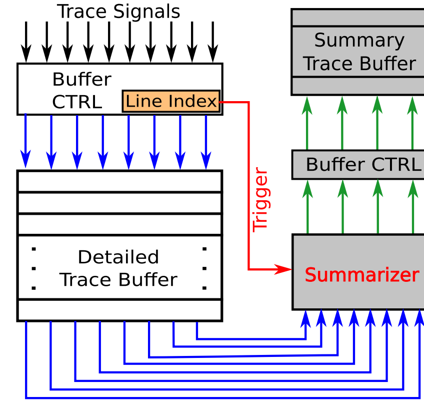

Fig. 3. Split architecture.

The first technique implicitly assumes that the operations occurring sufficiently in the past would correspond to expected executions and that the symptoms of erroneous behavior would be captured in the detailed trace messages generated most recently. Under this assumption, the former method generates trace summaries of the detailed trace messages that are further in the past, which can be used as activity logs. The second technique does not make any such assumption, and uses an invariant specified by the validation engineer using SSL to decide whether or not the execution follows a correct path.

The simultaneous storage of detailed trace messages and brief activity logs of successful operations is called *tapered storage*. We propose three different trace buffer architectures for implementing tapered storage: 1) split; 2) unified; and 3) overlapped to store tapered summaries.

B. Split Architecture

Fig. 3 shows the *split* architecture where the detailed traces and their summaries are stored in separate trace buffers. The detailed trace messages are marked in blue and the summaries are marked in green. The space in the detailed trace buffer (DTB) is regulated using two thresholds. When the number of trace messages in the DTB exceeds the higher threshold, the summarizer makes space for incoming messages by generating summaries and storing them into the summary trace buffer (STB); it stops doing so when the available space falls below the lower threshold. The sizes of the DTB and STB are frozen at design time.

C. Unified Architecture

Fig. 4 shows the *unified* architecture for storing tapered summaries. This architecture allows the validation engineer to configure the sizes of the DTB and STB as per the requirement of the debug scenario. This is achieved by merging the two trace buffers into a single physical buffer. In steady state, new trace data is written into the DTB, and the oldest detailed trace is simultaneously summarized and written into the STB in the same clock cycle. The architecture retains the two thresholds of the split architecture to maintain flow control between the DTB and STB. This architecture requires a trace buffer with three ports to support online operation of the summarizer: two write ports (one each to store incoming

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

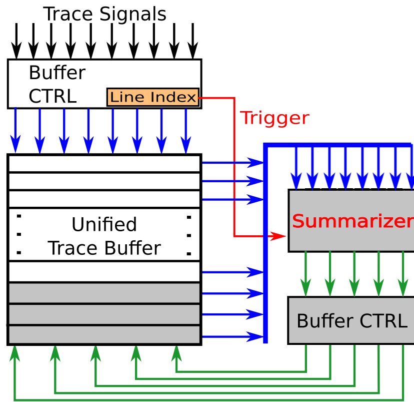

Fig. 4. Unified architecture.

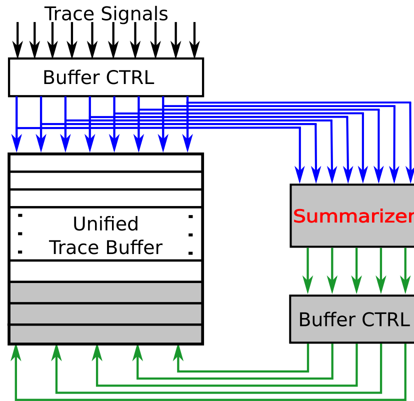

Fig. 5. Overlapped architecture.

detailed traces and summaries generated by the summarizer) and one read port to read detailed trace messages into the summarizer. This leads to an increased area overhead of the unified architecture.

D. Overlapped Architecture

Figs. 1(b) and 5 show the *overlapped* architecture. This architecture retains the flexibility of the unified architecture but reduces the area overhead by discarding the dedicated read port of the unified architecture. The overlapped architecture generates and stores the summaries simultaneously with the detailed trace messages being written to the DTB. This avoids the need for the summarizer to separately read the detailed trace message again later. Both the DTB and STB operate as regular circular buffers and the oldest message is overwritten by the incoming ones. If overwriting is not desirable, the system is stalled and the contents of only the STB are transferred off-chip. We transfer the contents of the DTB off-chip only after an error is detected.

V. Hardware Design

Fig. 6 shows the high-level design of the proposed summarizer. The detailed trace messages are input into the spatial summarizer and a staging buffer. Similarly, the output of the spatial summarizer is forwarded into the *temporal expression checker* (TEC) and a different staging buffer. The TEC checks whether the trace messages meet the specified

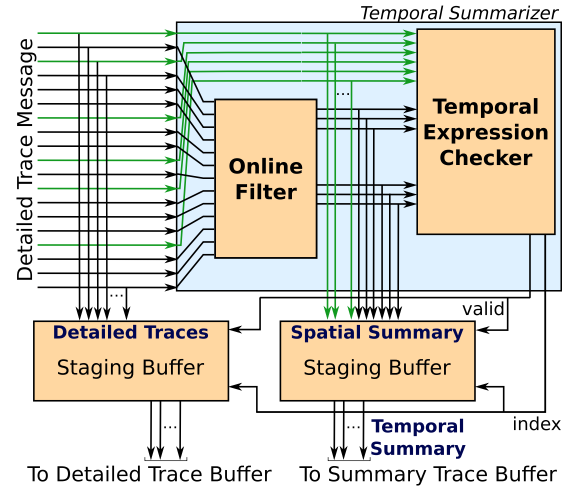

Fig. 6. High-level design of trace summarizer (for the overlapped architecture).

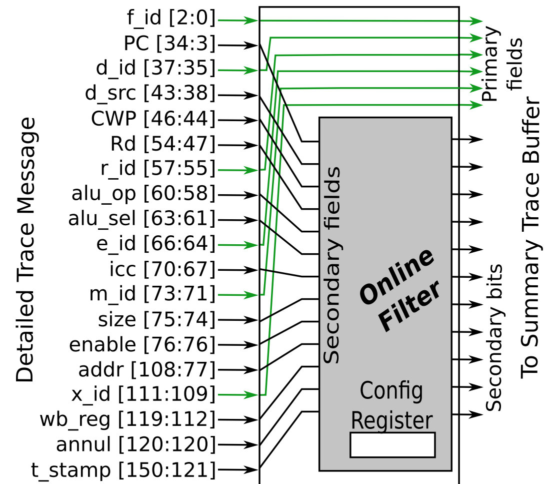

Fig. 7. High-level design of our proposed spatial summarizer on the LEON3 processor.

temporal condition or not. The two staging buffers, one for the detailed trace messages and the other for its spatial summaries, are used as temporary storage until a decision on whether or not to write them into the trace buffer is available from the TEC. The staging buffer itself is configured as a circular queue, and the oldest entries are overwritten in the case of an overflow. An entry is made into the staging buffer only when the input trace message triggers an event indicating the possibility of it being part of the specified sequence. The contents of the appropriate staging buffer are flushed into the trace buffer if the specified temporal condition is met and discarded otherwise. The following trivial modifications are made for split or unified architecture: 1) the `valid` signal to the staging buffer is ANDed with the trigger signal shown in Figs. 3 and 4 and 2) a staging buffer is not required for detailed traces.

A. Spatial Summarizer

Fig. 7 shows the hardware design of the spatial summarizer for the pipeline traces. The fields of the detailed trace messages are classified into primary and secondary fields. The primary fields contain only identifying information of the detailed trace

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

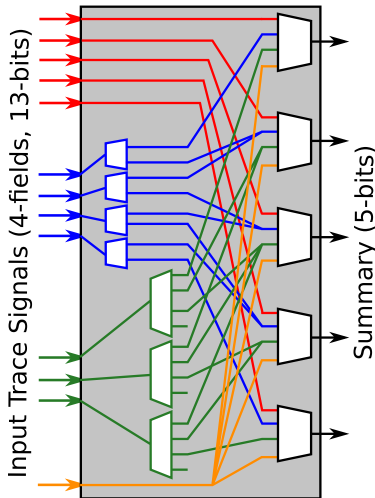

Fig. 8. Design of the online filter.

message, which is essential to reconstruct the sequence of events during off-chip analysis. Therefore, they are made part of every summary and are transferred unchanged to the output. The `ID` of the instruction that is currently being processed by a stage (fetch, decode, etc.) of the in-order pipeline is a primary field because it helps reconstructing the activity sequence in the pipeline.

A subset of the secondary fields is chosen using an *online filter* that selects any k lines out of n input lines. The straightforward technique to achieve this is to use k n-to-1 multiplexers (MUXes). This allows the validation engineer to select any bit of the input at any of the k output positions. However, this design may lead to prohibitively high area overheads when the number of input lines is large; we propose an alternative design that minimizes this overhead by imposing the constraint that the order of bits within a field remains unchanged, though the relative order of the fields could be programmed by the validation engineer. Permutation of fields is important to enable compaction before further on-chip processing such as compression [22]. Only the number of bits in the output and the trace signals appearing at the inputs are fixed at design time.

Fig. 8 shows the hardware design of the online filter. The output of each multiplexer (MUX) is a bit of the summary. The relative order of fields at the input of each MUX is the same across all MUXes, that is, any bit of input field 0 will appear only on input line 0 of all the MUXes. This gives a consistent way of specifying which field should be selected at a particular MUX. Since we allow for the relative order of the fields to change, a particular bit within a field can potentially occur at any output MUX. We use a DeMUX to select at runtime one output MUX to which a particular bit within the field should go. Fig. 9 illustrates an example of the permutations between fields 2 and 4 at the output. Such permutations allow the width of the filtered trace to be smaller than the number of output bits of the online filter. For example, if the validation engineer is only interested in two fields of 1 bit each, then these two fields can be made to appear at the first two lines of the output. The remaining lines of the output need not be written to the trace buffer.

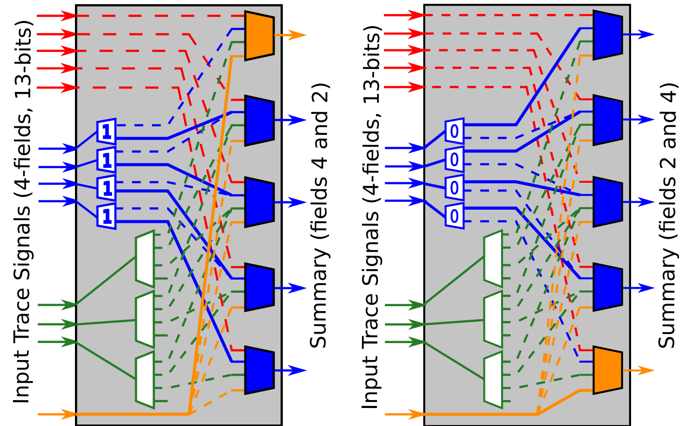

Fig. 9. Illustration of working of the online filter.

B. Temporal Summarizer

Fig. 10 shows the internal blocks of the TEC along with other modules of the summarizer. Several works in the past have synthesized Property Specification Language properties and Sequential-extended Regular Expression (SEREs) into hardware by constructing an equivalent finite automaton [18], [23]. We leverage the aforementioned works that generate an equivalent automaton for the specified temporal expression and use it to program a generic state transition table present inside the summarizer. This transition table checks for the occurrence of specified sequences of events. We create a dedicated unit for the repetition operator and capture the other temporal operators through suitable state-transition tables.

The internal blocks of the TEC can be grouped into: 1) front-end event generation that includes the spatial summarizer, event triggers, and the boolean expression tree; 2) the repetition counters; and 3) the controller. We discuss the functioning of the internal blocks in the following paragraphs. The spatial summaries generated by the spatial summarizer are passed into the event triggers where the values of the fields are matched against preset values to detect the events of interest. The triggers convert multibit trace values into 1-bit events, which in turn helps restrict the overall area consumed by our proposed hardware. The event triggers can be reprogrammed to detect different events at runtime through the `set` command. The `set` command sets the value against which future events should be triggered to the current value of the field.

It is possible that the field selected by the spatial summarizer is wider than the input size of event triggers. We handle such cases using multiple event triggers, with each trigger tracking different portions of the field selected by the spatial summarizer. An event is said to occur only when *all* the triggers that are tasked with detecting matches of different bits of the output field fire simultaneously.

We use a *boolean expression tree* to check if all the triggers fired simultaneously, which is the logical AND of the outputs from the corresponding triggers. Fig. 11 shows a possible implementation of the boolean expression tree where the design of each node is shown in Fig. 12. Each node of this reduction tree can either perform logical operations AND/OR between the inputs or forward the inputs as is to the outputs if no such operations are required. In addition to detecting

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

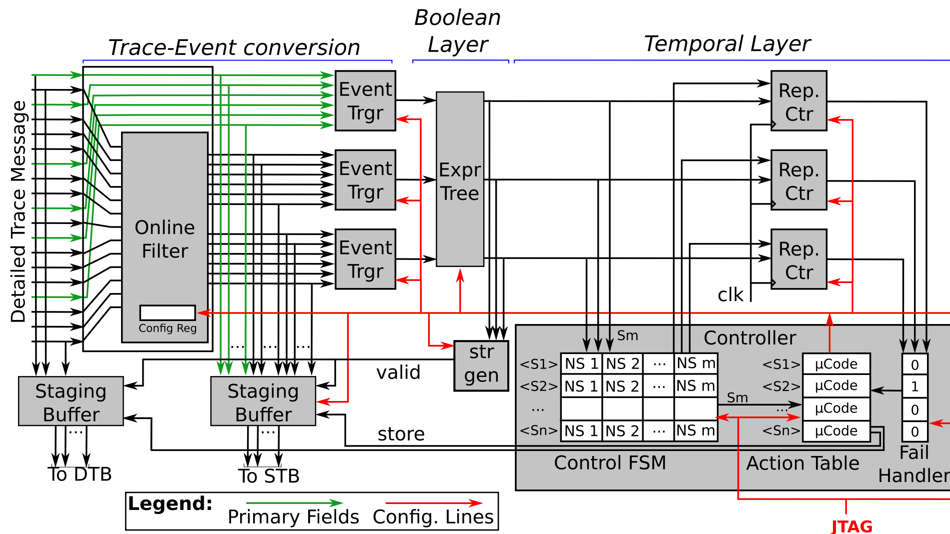

Fig. 10. Design of the temporal summarizer.

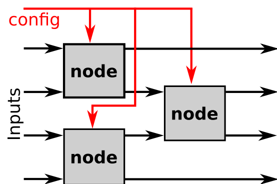

Fig. 11. Boolean expression tree.

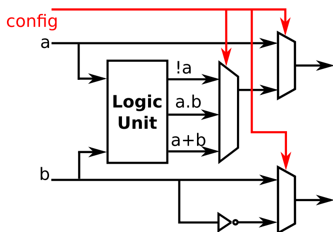

Fig. 12. Boolean expression evaluator node.

events on the fields that are wider than the input width of event triggers, the boolean expression tree is also used to compute any boolean operations that the validation engineer may have specified in the input.

The outputs of the boolean expression tree indicate the events (and the logical combinations between them) that have occurred at a particular instance of time. These are used to drive three key activities within the TEC.

1) Indicate whether or not the input trace message (and the corresponding spatial summary) should be written into the staging buffer.
2) Request the *repetition counter* to start counting so that the number of cycles that have elapsed since the occurrence of this event is known.
3) Notify the FSM so that it can request the repetition counter that is counting the cycles elapsed since the occurrence of the previous event in the sequence, to stop counting.

The outputs of the boolean expression tree are used to decide whether or not the input trace message and its spatial summary should be stored into the corresponding staging buffers. This decision is taken by the *str gen* unit and depends on the output of the boolean expression tree and whether the negation operator `NOT()` is specified against the condition. If the result of the condition is not negated, *str gen* requests the staging buffer to store the trace message and its spatial summary into the respective buffers when at least one of the outputs of the boolean expression tree is high. If the result is negated, all the messages are written to the staging buffer and the decision to write them to the trace buffer is deferred to the TEC.

We associate a repetition counter with each output of the boolean expression tree, which counts the number of clock cycles that have elapsed since the occurrence of an event, until any of the following three conditions occur: 1) the value of the field changes in a subsequent trace message; 2) it is explicitly stopped by the control Finite State Machine (FSM); or 3) the count exceeds the range of cycles specified. In all the three cases, the repetition counter asserts a `fail` signal only if the final count is not within the range specified by the validation engineer. The control FSM can notify the repetition counter associated with the most recent event to stop counting, as it has the knowledge of whether an event of interest (or a logical combination of events) is detected by the event triggers in the current cycle.

The control FSM stores the state-transition table that accepts (or rejects) an input trace sequence. Our implementation of the transition table is as follows: the index into the transition table encodes the current state and the corresponding row stores the transitions to the next state on all possible inputs. The actual inputs are used to read the appropriate bits from the row corresponding to the current state. An *action unit* stores the microcode that dictates the action to be performed on transitioning to the current state.

A *fail handler* masks the `fail` signal from being passed on to the action unit, and is used to control the behavior of the TEC when the input trace sequence does not match the specified sequence. This is required to handle negations of temporal conditions that are specified using the `NOT()` operator. The action unit checks the eventual `fail` state before triggering

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the specified action.

The input specification is used to program the summarizer by writing the transition table, the action table, and the fail handler suitably. Since the proposed unit implements a microprogrammed control logic, the action to be taken on transitions between states is changed by overwriting the entries in the transition table and the action unit across runs. We explore the use cases of dynamic reprogramming in a future work.

C. Programming the Summarizer

We briefly explain the steps to translate SSL statements into binary code using the SSL statement below.

```
store {MCI,PC,TRAP} when {(MCI==1);(PC==0xab)}
```

The order in which the secondary signals are mentioned in the input statement is the order in which they appear in the spatial summary. The primary fields such as `PC` are always available at the first few lines of the output because they bypass the online filter. The secondary fields in the example, `MCI` and `TRAP`, are available at the first two lines of the output of the online filter. The values against which triggers are defined are used to program the appropriate event triggers where the particular fields are available. In our case, the two events `(MCI==1)` and `(PC==0xab)` are available at the outputs of Event Triggers 1 and 2, say E1 and E2, respectively. The operators mentioned in the input that match either `BINARY_OP` or the `!` operator from the grammar are used to program the boolean expression tree. Similarly, the repetition counts that match `Rep_Count` in the grammar are used to program the repetition counters. These structures are bypassed in our example.

These translations convert the input `Condition` into an expression involving internal signals such as E1 and E2. A simple state-transition table is generated along the lines of [18], and the same is written into the control FSM. A state-machine with three states is derived in our example. The action table controls the commands to be issued when the FSM transitions into the accepting state. The controller can issue three different commands: 1) flush the contents of the staging buffer into trace buffer; 2) notify an ID on the network; and 3) set the value of the trace signal mentioned against `set` in the event triggers. The bit corresponding to the accepting state in the fail handler is set to 1 in the case of the `NOT()` operator.

Fig. 10 shows the paths (marked in red) along which the binary code thus generated is transferred to different modules of the summarizer.

D. Limitations

The simplicity of the current design imposes the following limitations on the capabilities of the summarizer. All of these limitations can be overcome with additional hardware.

1) Only four events are detected by this design in order to limit the size of the event trigger datapath and control FSM. This is often sufficient because the temporal summaries are used quite deep into the debug process when the region and conditions of interest have been considerably narrowed down by the validation engineer.
2) An event cannot be part of multiple boolean expressions because the inputs to the boolean tree are mutually exclusive.
3) If occurrences of event sequences are overlapped, where the first event from a subsequent occurrence of the specified sequence is triggered before the last event of the current occurrence, then the TEC detects only the most recent occurrence.
4) The number of store conditions that can be issued to the summarizer at a time depends on the total number of simultaneous events that being tracked. If each store condition involves only one event, then four different store conditions can be issued and the state-transition table of the control FSM will be an aggregrate of four individual control FSMs.

VI. Extensions to Distributed and Multicore Systems

The amount of execution trace to collect and transfer off-chip is higher in distributed or multicore systems, where debug information may be collected from multiple sources. A single large trace buffer to capture execution traces from multiple sources is not scalable due to contention on shared resources. State-of-the-art designs distribute the on-chip trace buffers and associate them with individual components such as cores and accelerators, as well as shared resources such as the interconnect and memory controller [20]. However, this naive scheme does not reduce the number of stalls because the size of each trace buffer continues to be small.

We propose to use the aforementioned summarizer to capture trace summaries into the trace buffer associated with different components and the platform. However, a simple extension that does not support cross triggering (triggering on events that occur on a different core) can capture irrelevant information into the trace buffers. We exploit the opportunities for reducing the number of stalls significantly by extending our proposed hardware to support efficient cross triggering. An efficient cross-triggering mechanism is known to yield significant benefits when debugging hard to repeat errors [24].

We build on existing proposals and therefore assume the presence of a reliable interconnect that can carry messages related to debug events between different debug controllers. Previous works have proposed two broad approaches for transferring these messages: 1) use a dedicated lightweight interconnect [25] and 2) insert markers into payloads being exchanged over the existing interconnect [26]. Our proposal is agnostic to the internal details of the communication infrastructure available and is compatible with both these approaches. We broadly refer to the underlying communication infrastructure as *debug interconnect*.

A. Specification

We use the SSL statement template below to allow a summarizer to notify its peers about the occurrence of an event

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
DETAILS OF TRACES GENERATED BY THE DfD HARDWARE

Buffer Location	Trace Type	Detailed (#bits)	Primary (#bits)	Secondary (#bits)				Summary (#bits)			
				CCI	WIM	NAE	CSL	CCI	WIM	NAE	CSL
Core	Instruction	128	32	-	-	-	-				
	Pipeline	151	18	32	-	32	-	114	85	114	83
	Cache	80	32	-	-	-	1				
DSU	AHB	128	32	32	-	32	32	64	-	64	64

of their interest, instead of notifying its local trace to store a trace summary.

```
notify {NID} when {CONDITION}
```

The notification ID is an encoding of a unique event identifier and the destination. In case the debug interconnect is a simple broadcast medium, the destination is not specified. The receiver can set an explicit trigger on the notifications it receives over the debug interconnect (similar to the ones set on input trace messages) in order to take suitable action as and when an event of interest occurs on the remote core.

There are debug scenarios where the condition specified on the remote core holds true for several cycles. An example of such a specification is shown below.

```
notify {0x01} when
  {(PC>=0xabcd1234) and (MCI==0)}
```

In this case, sending a new notification for every cycle the condition holds true can potentially congest the interconnect by flooding it with redundant messages. Similarly, constantly asserting a signal between the sender and the receiver until the specified condition turns false has high area overhead. We address this issue by sending a separate *clear* message to the receiver when the condition that resulted in sending the first notification message is no longer valid.

B. Storage Architecture

We use a distributed architecture to store trace summaries and propose to use an STB with each detailed trace that is associated with a core and the platform. This architecture overcomes the three disadvantages of a centralized STB as follows.

1) Runtime partitioning of the available size to capture detailed and summary traces based on the debug scenario is now possible because the DTB size is not frozen at design.
2) Trace loss is minimized by reducing contention between summaries from multiple cores.
3) Space is utilized efficiently by tagging the core ID only when transferring the contents off-chip rather than on every trace summary. The distributed architecture requires a small additional synchronization hardware to access the off-chip interconnect when transferring the contents.

C. Hardware Extensions

The following minor modifications are made to the summarizer to support cross triggering: 1) new output lines connecting the summarizer to the interconnect are used for cross triggering; 2) new microinstructions are added to send `notify` and clear messages over the debug interconnect; and 3) the inputs from the debug interconnect are connected to an existing event trigger.

VII. EXPERIMENTS

A. Setup

Our experimental setup consists of a LEON3 SoC with 4 cores (SPARCv8), a memory controller, and an advanced high-performance bus (AHB). The off-the-shelf LEON3 SoC has the following debug features: 1) generate instruction traces and store them locally in an instruction trace buffer and 2) debug support unit (DSU), which stores AHB traces into a separate trace buffer in addition to supporting activities like diagnostic read (or write) from (or to) system registers, accesses to the internal state through JTAG and UART ports, and so forth. We enhanced this DfD hardware by tracing critical signals from the pipeline (similar to [27]), and cache controllers. Table II shows the number of bits captured by each of these detailed traces. We perform differential compression [28] before storing them into their respective trace buffers. We retain the distributed approach followed by the original LEON3 SoC and use a 4-kbyte trace buffer per core and the AHB to store the generated traces. The information captured in the detailed trace is fixed at design time and remains the same in all the scenarios we consider in the following section. However, the bits captured in the summaries change across scenarios, as shown in Table II, as per the specifications from the validation engineer. The input applications are written in C and are compiled using the *Bare-C* cross compiler, which allows them to be executed off the bare metal. We used CACTI 5.3 to estimate the area of the trace buffers. The proposed design was implemented in VHDL and synthesized using a Cadence Encounter RTL compiler with a 90-nm technology standard cell library.

B. Bug Scenarios

We modeled different realistic architectural bugs that could interfere with the functionality using a simple application with four threads: two writers and two readers. The writers increment a global counter and the readers continuously read it. The application was allowed to execute for 75 000 cycles and the bugs were introduced randomly. Such directed test applications are used extensively during postsilicon validation [2].

We considered three different configurations to capture trace messages into the on-chip trace buffers: 1) base case: the DTB of 4 kbytes with no STB; 2) the STB of 3 kbytes and the DTB of 1 kbyte; and 3) the STB and DTB of 2 kbytes each.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

We studied the number of stalls required to transfer the contents of the trace buffer off-chip in each of these configurations. Under our proposed methodology, only the contents of the STB are transferred off-chip when it is full and the older messages in the DTB are overwritten by newer detailed trace messages. If only detailed trace messages are of interest, then the trace buffer is configured to store only the detailed trace messages, which are transferred off-chip as and when the trace buffer overflows.

1) Core–Cache Interface: This bug is inspired by our experience from an actual design scenario involving adding a victim cache to the processor. Previous studies have also found that a large number of functional bugs occur at the interface of the core [29]. The implementation of the victim cache sent the data to the core as well as to the L1 cache as an optimization if the request to it resulted in a hit. This required changes to the interface between the core and the memory hierarchy, such that the core can accept data either from the victim cache or from the regular memory hierarchy through the data cache. The interface between the core and the memory hierarchy is governed by two key control signals: 1) MDS and 2) HOLDN. The former is a strobe signal that indicates whether valid data are available for the LEON3 processor to store into its internal registers, and the latter is used to stall the pipeline until the memory system returns valid data to the pipeline. The protocol to be followed at the interface mandates that MDS should change one cycle before HOLDN changes. When adding a victim cache into the base system, the timing of these signals was violated, due to which the core proceeded with its execution using the stale value present in its internal register.

After the initial run, we analyzed the most recent detailed trace messages available on-chip, which revealed that the contents of a register retained the same value across multiple increment operations. In the subsequent run, we captured a spatial summary that included: 1) address and data values of the AHB trace and 2) address and data values of the pipeline trace. Only the spatial summaries (a total of 114 bits) were transferred off-chip. This was achieved using the SSL statements below.

```
store {MADDR, MDATA} when {true[*MAX]}
store {MADDR, CDATA} when {true[*MAX]}
```

The two SSL statements are used to program the online filters associated with the AHB trace and the pipeline trace, respectively. These two SSL statements capture a narrow but lengthy trace of data accesses made by the pipeline and the responses from the memory. The data returned to the pipeline are correlated with the data returned by the memory system using the addresses captured in the traces. A mismatch in the data returned to the pipeline and the data visible over the AHB helped us localize the bug to the core–cache interface. This is because the LEON3 SoC uses a write-through cache with no-allocate policy, and therefore, dirty data are never present in the caches. If dirty data are allowed in the system, then each mismatch would have to be analyzed during postprocessing to see if intervening stores were issued. This would not affect the number of stalls because this information is also present in the traces captured using the above-mentioned SSL.

TABLE III
NUMBER OF STALLS REQUIRED FOR DIFFERENT TRACE BUFFER CONFIGURATIONS WHEN STORING SPATIAL SUMMARIES

Bug	Simple Trace Buffer 4 KB		Overlapped architecture (DTB/STB) 2/2 KB		Overlapped architecture (DTB/STB) 3/1 KB	
	#Stalls	History	#Stalls	History	#Stalls	History
CCI	138	1332	51	3491	77	2374
WIM	94	1109	36	2085	54	1916
NAE	138	1332	51	3491	77	2374
CSL	138	1332	52	3534	78	2307

To pinpoint the cause of the erroneous behavior, we programmed the temporal summarizer to store a spatial summary that includes the data returned to the pipeline (CDATA, MDS, and HOLDN) only when the invariant to be followed by the control signals at the interface was violated. This was achieved using the SSL given below.

```
interface_seq := (CDATA!=CDATA') and
    (MDS==0)\set CDATA\;(HOLDN==1)

store {CDATA, MDS, HOLDN} when
    {NOT(interface_seq)}
```

Table III shows the number of stalls required to transfer the spatial summaries off-chip and the maximum duration of activity history captured in the trace buffer, for the three different configurations of the overlapped architecture: 1) the DTB of 4 kbytes; 2) the STB of 3 kbytes and the DTB of 1 kbyte; and 3) the DTB and STB of 2 kbytes each. We observe that the number of stalls decreases by 63% and the activity history is extended by 162% when an STB of 3 kbytes is used. This reduction in stalls occurs because we dump only the STB contents. This resulted in the total time spent on dumping the trace buffer contents off-chip reducing from 100.72 to 29.7 s, when transferred over LEON3 SoC's DSU serial link operating at 115 200 bits/s.

2) Window Invalid Mask: The next bug scenario deals with the corruption of the window invalid mask (WIM) of one of the SPARCv8 cores in our SoC. The WIM is used by the SPARCv8 core to detect register window overflow when executing the SAVE instruction and register window underflow during the execution of the RESTORE instruction. When the current window pointer (CWP) points to the WIM, the execution of the current instruction traps, thereby invoking the suitable handler to either save a register window to create space or restore previously saved contents into a register window. We investigated a bug in the pipeline that corrupted the WIM in a way such that the execution of the trap handler itself traps (double trap), which forces the processor to reboot. Such a bug falls into the class of single-bit upsets at flip-flops [27], [30]. A bug due to corruption of WIM has long suspect windows when one or more functions are called within a loop. The length of the suspect window increases as the number of iterations increase, because the bug manifests into a failure only when the call stack shrinks eventually. For this case study, a corruption of the WIM during the system bootup led to a crash after 32 646 cycles.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Initially, a simple instruction trace, which is the default spatial summary (primary fields), was captured to gain insights into the possible reasons for the observed erroneous behavior. Since the execution was within the trap handler for window underflow, the bug was localized to the register window logic.

For further debugging, a combination of instruction trace and detailed pipeline trace only when the CWP changed was stored into the trace buffer as per the SSL shown below.

```
store {all} when
  {(CWP==CWP') ; (CWP!=CWP') \set CWP\}
```

The detailed pipeline trace (of 151 bits) is stored instead of just 21 bits (of spatial summary) every time the CWP changes. This shows an example scenario where spatial summaries and detailed trace messages are required to be stored simultaneously. The stall reduction is shown in Table III.

3) Nonatomic Execution of `ldstuba`*:* Determining the root cause of race conditions introduced into applications due to hardware defects using at-speed debugging is a challenge because of their nonrepeatable nature [15], [24]. The nonrepeatable nature of the bug requires the test case to be executed multiple times for long durations, continuously collecting detailed traces, and analyzing them for the presence of bugs.

A bug deep in the logic of the atomic exchange instruction (`ldstuba`) of SPARCv8 caused the master issuing the `ldstuba` instruction to not lock the AHB until the semaphore was acquired by writing the value `0xff` at address 0x400126f0. This violated the atomic operation of `ldstuba` and hence introduced a data race in our application.

A spatial summary consisting of address and data values in AHB trace and data values returned to the pipeline helped eliminate possible sources of errors such as errors on the interface and unintentional caching of values. This led to the reduction in stalls obtained through the use of spatial summaries, which is shown in Table III.

We then coded the invariant for mutual exclusion on the AHB trace, as shown in the SSL below.

```
acquire_r := (MADDR==0x400126f0) and (OP==r)
    and (MDATA==0x00)

store {all} when
  { {(MSRC!=MSRC') and acquire_r} within
    {acquire_r\set MSRC\[*0:40];} }
```

This gave insights into the detailed state of the AHB as and when mutual exclusion was violated. On examining the detailed state, we observed that the AHB master released the lock before the slave responded. This helped to pinpoint the cause of the bug to the implementation of `ldstuba`. Since the detailed traces were captured only on violation of mutual exclusion, the trace buffer never overflowed for the sample application under consideration.

4) Cache Snooping Logic: Another bug in the cache coherence logic also led to a race condition. This tested the robustness of our methodology to different bugs that result in similar manifestations. The reason for the observed race condition was data duplication, instead of mutual exclusion violation. This scenario arose because the semaphore is maintained in an alternate address space of SPARCv8 that bypasses the caches, whereas the shared data is maintained in a cacheable address space for performance reasons. Therefore, two writers were not incrementing the counter simultaneously, but each writer was incrementing a stale value of the counter because of the faulty cache invalidation logic.

A spatial summary that captures address and data values in the AHB trace and line numbers of cache lines invalidated due to snoop hits was used to narrow down the root cause to faulty cache behavior. To determine the exact cause for the observed error, we used the SSL shown below for cross triggering.

```
notify {0x1} when
  {(MADDR==0x40013214) and (OP==w)}

store {CDATA,SNHIT,SNADDR} when {(NID==0x1)}
```

The above statements are configured on all the summarizers in order to track all the updates to the shared variable and the invalidates that occur on other caches as a result of it. The first statement is responsible for notifying the peers every time the shared variable is updated. The latter detects the updates to the shared variable and captures the snoop hit status and the corresponding address to give specific visibility into the invalidation logic.

None of the previous works on multilevel tracing [16], [19], [24] would be able to detect all the bugs studied above, because each bug required the capture of a different set of signals, and a different combination of detailed traces and summaries.

C. Area Overhead

We synthesized the proposed summarizer that generates secondary fields that are up to 32 bits at the output of the online filter for each trace type. The inputs to each event trigger are 8-bit wide. Therefore, the 32 bits of the secondary fields are partitioned statically into four groups and each one is assigned an event trigger and a repetition counter. The last column of Table IV shows the area overhead and the delay of the critical path of the summarizer for each trace type. The total area occupied by all the summarizers associated with a core is 0.187 mm^2. The summarizer associated with processing AHB traces occupies 0.066 mm^2.

The split architecture, with the DTB and the STB of 2 kilobytes each, occupies 0.216 mm^2, of which 0.139 mm^2

TABLE IV

AREA OVERHEAD OF THE PROPOSED DfD HARDWARE

Trace Type	Online Filter		Summarizer	
	Area (mm^2)	Delay (ps)	Area (mm^2)	Delay (ps)
Instruction	0.0018	469	0.0508	638
Pipeline	0.017	793	0.073	984
Cache	0.009	659	0.063	842
AHB	0.013	523	0.066	702

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

and 0.077 mm^2 are occupied by the DTB and STB, respectively. Although the overall sizes of the DTB and STB are the same, the widths of the entries in these two trace buffers are different. The unified architecture has the highest area overhead of 0.399 mm^2, due to the increased number of ports. The overlapped architecture occupies 0.186 mm^2, which is less than that of the split architecture by 13.9%. This is due to the reduction in the number of ports compared with the unified architecture and also due to the sharing of the SRAM read/write controller across both the DTB and STB compared with the split architecture.

The total area occupied by the trace buffers and the summarizers (associated with one core, including the AHB) is 0.625 mm^2 $(= (0.187 + 0.066 + 0.186 \times 2))$, which is only 1.8% of the area occupied by a 64-kilobyte cache.

VIII. Conclusion

In this paper, we proposed to generate and store trace summaries before overwriting the captured trace messages during postsilicon debug. Specifically, we presented an overlapped debug architecture where the trace summaries are stored alongside the recently captured execution traces in a flexible manner. The trace summaries are generated by retaining only the information of interest to the validation engineer and discarding the rest. To this end, we proposed an SSL that the validation engineer can use to specify the information to be retained in the summaries. Moreover, we extended the base design to support cross triggering in the case of distributed and multicore systems. We showed how the proposed enhancements exploit the increasing amounts of information gathered about the bug over successive debug attempts to progressively reduce the number of stalls until they are avoided, in all of the four bug scenarios that we considered. The proposed methodology does not make any assumptions about the type of errors and supports at-speed debugging of different classes of bugs such as nonrepeatable errors and errors with long suspect windows that are encountered during postsilicon validation.

References

[1] A. Adir, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor," in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2011, pp. 569–574.

[2] T. Bojan, M. A. Arreola, E. Shlomo, and T. Shachar, "Functional coverage measurements and results in post-silicon validation of Core 2 duo family," in *Proc. IEEE Int. High Level Design Validation Test Workshop (HLVDT)*, Nov. 2007, pp. 145–150.

[3] Q. Xu and X. Liu, "On signal tracing in post-silicon validation," in *Proc. 15th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2010, pp. 262–267.

[4] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan, "Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2015, pp. 1–8.

[5] M. Neishaburi and Z. Zilic, "Hierarchical embedded logic analyzer for accurate root-cause analysis," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2011, pp. 120–128.

[6] H. F. Ko and N. Nicolici, "Mapping trigger conditions onto trigger units during post-silicon validation and debugging," *IEEE Trans. Comput.*, vol. 61, no. 11, pp. 1563–1575, Nov. 2012.

[7] E. Anis and N. Nicolici, "On using lossless compression of debug data in embedded logic analysis," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2007, pp. 1–10.

[8] K. Basu and P. Mishra, "Efficient trace data compression using statically selected dictionary," in *Proc. IEEE 29th VLSI Test Symp. (VTS)*, May 2011, pp. 14–19.

[9] C. H. Lai, Y. C. Yang, and I. J. Huang, "A versatile data cache for trace buffer support," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 11, pp. 3145–3154, Nov. 2014.

[10] R. Abdel-Khalek and V. Bertacco, "Functional post-silicon diagnosis and debug for networks-on-chip," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2012, pp. 557–563.

[11] J.-S. Yang and N. A. Touba, "Improved trace buffer observation via selective data capture using 2-D compaction for post-silicon debug," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 2, pp. 320–328, Feb. 2013.

[12] X. Liu and Q. Xu, "On multiplexed signal tracing for post-silicon validation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 5, pp. 748–759, May 2013.

[13] E. A. Daoud and N. Nicolici, "On using lossy compression for repeatable experiments during silicon debug," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 937–950, Jul. 2011.

[14] B. Vermeulen, K. Goossens, and S. Umrani, "Debugging distributed-shared-memory communication at multiple granularities in networks on chip," in *Proc. 2nd ACM/IEEE Int. Symp. Netw.-Chip (NoCS)*, Apr. 2008, pp. 3–12.

[15] A. M. Gharehbaghi and M. Fujita, "Transaction-based debugging of system-on-chips with patterns," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Oct. 2009, pp. 186–192.

[16] C. T. Huang, K.-C. Tasi, J.-S. Lin, and H.-W. Chien, "Application-level embedded communication tracer for many-core systems," in *Proc. 20th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2015, pp. 803–808.

[17] M. Boulé, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proc. 8th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2007, pp. 613–620.

[18] M. Boulé and Z. Zilic, "Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2007, pp. 324–329.

[19] M. H. Neishaburi and Z. Zilic, "On a new mechanism of trigger generation for post-silicon debugging," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2330–2342, Sep. 2014.

[20] Cobham Gaisler AB. *LEON3 Processor*, accessed on Feb. 1, 2017. [Online]. Available: http://www.gaisler.com

[21] *IEEE Standard for Property Specification Language (PSL)*, IEEE Standard 1850-2010, IEC 62531:2012(E), Jun. 2012, pp. 1–184.

[22] P. R. Panda, M. Balakrishnan, and A. Vishnoi, "Compressing cache state for postsilicon processor debug," *IEEE Trans. Comput.*, vol. 60, no. 4, pp. 484–497, Apr. 2011.

[23] M. Boulé and Z. Zilic, "Efficient automata-based assertion-checker synthesis of PSL properties," in *Proc. 11th Annu. IEEE Int. High-Level Design Validation Test Workshop*, Nov. 2006, pp. 69–76.

[24] E. Larsson, B. Vermeulen, and K. Goossens, "A distributed architecture to check global properties for post-silicon debug," in *Proc. 15th IEEE Eur. Test Symp. (ETS)*, May 2010, pp. 182–187.

[25] A. Azevedo, B. Vermeulen, and K. Goossens, "Architecture and design flow for a debug event distribution interconnect," in *Proc. IEEE 30th Int. Conf. Comput. Design (ICCD)*, Sep. 2012, pp. 439–444.

[26] S. Tang and Q. Xu, "In-band cross-trigger event transmission for transaction-based debug," in *Proc. Conf. Design, Autom. Test Eur.*, Mar. 2008, pp. 414–419.

[27] S.-B. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proc. 45th Annu. Design Autom. Conf.*, Jun. 2008, pp. 373–378.

[28] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug support for complex systems on-chip: A review," *IEE Proc.-Comput. Digit. Techn.*, vol. 153, no. 4, pp. 197–207, Jul. 2006.

[29] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Archit. (MICRO)*, Dec. 2006, pp. 26–37.

[30] A. DeOrio, D. S. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2011, pp. 755–761.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Sandeep Chandran received the bachelor's degree in computer science and engineering from Visveswaraya Technological University, Belgaum, India.

He is currently a Research Scholar with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India. His current research interests include postsilicon validation methodologies and design-space exploration.

Preeti Ranjan Panda received the B.Tech. degree in computer science and engineering from IIT Madras, Chennai, India, and the M.S. and Ph.D. degrees from the University of California at Irvine, Irvine, CA, USA.

He was with Texas Instruments and Synopsys, and has been a Visiting Scholar at Stanford University, Stanford, CA, USA. He is currently a Professor with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India. He has authored two books: *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* and *Power-Efficient System Design*. His current research interests include embedded systems and design automation.

Prof. Panda was a recipient of the IBM Faculty Award and the Department of Science and Technology Young Scientist Award. He served as the Technical Program Co-Chair of CODES+ISSS and VLSI Design, and on the Technical Program Committees and chaired sessions at several conferences including DAC, ICCAD, DATE, CODES+ISSS, ISLPED, and EMSOFT. He also served on the editorial boards of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, *ACM Transactions on Design Automation of Electronic Systems*, and *International Journal of Parallel Programming* (IJPP).

Smruti R. Sarangi received the B.Tech. degree in computer science from IIT Kharagpur, Kharagpur, India, in 2002, and the M.S. and Ph.D. degrees in computer architecture from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2007.

He was involved in the design of computer architecture, and parallel and distributed systems. He is currently an Assistant Professor with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India.

Prof. Sarangi is a member of ACM.

Ayan Bhattacharyya received the M.Tech. degree in integrated electronics and circuits from IIT Delhi, New Delhi, India, in 2014.

He is currently an IP Verification Engineer with Samsung Electronics, Bengaluru, India. His current research interests include digital design and verification of IP blocks.

Deepak Chauhan received the engineering degree in electronics and communication.

He was a Scientist with the Indian Space Research Organization Satellite Center, Bengaluru, India. He is having more than 15-years of experience in the design and validation area in renounced companies such as Freescale Semiconductor (NXP), Nvidia Graphics, and STMicro. He has authored many papers in the IEEE conferences on silicon validation and debug techniques.

Sharad Kumar received the B.E. degree from the Netaji Subhas Institute of Technology, New Delhi, India, in 1997, and the M.S. degree in computer engineering from Michigan State University, East Lansing, MI, USA, in 2000.

He has experience and interest in silicon validation methodologies and regularly collaborates with Academic Groups in this area. He is currently a Senior Manager with NXP Semiconductors, Bengaluru, India. He is responsible for the silicon validation of chips that come out of the Networking Group.

Mr. Kumar was a recipient of the SRC Mahboob Khan Mentor Award in 2013.