Extending Trace History Through Tapered Summaries in Post-silicon Validation

Sandeep Chandran, Preeti Ranjan Panda, Smruti R. Sarangi

Indian Institute of Technology Delhi
New Delhi, India
{sandeep,panda,srsarangi}@cse.iitd.ac.in

Deepak Chauhan, Sharad Kumar

Freescale Semiconductors India Pvt Ltd
Noida, India
{DeepakChauhan,sharad.kumar}@freescale.com

Abstract— On-chip trace buffers are increasingly being used for at-speed debug during post-silicon validation. However, the activity history captured by these buffers is small due to their limited size. We propose a novel scheme that extends the captured trace history (by upto 162%) by 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 demonstrate the usefulness of the proposed methodology for debugging various classes of bugs encountered during post-silicon validation.

I. Introduction

The increasing complexity of processors and SoCs, coupled with aggressive time-to-market deadlines, have forced chip-manufacturers to adopt well-planned strategies for post-silicon validation [1, 2]. It is important that the on-chip Design for Debug (DfD) architecture provides necessary handles to the validation engineer so that he can root-cause the design errors quickly.

On-chip trace buffers have emerged as a promising solution to the challenge of limited observability in post-silicon validation, with many chip manufacturers adopting it into their DfD architecture. However, the limited size of these on-chip trace buffers poses a significant challenge to post-silicon debug because only a short history of the activity inside a chip is captured, and the captured traces are required to be transferred off-chip frequently. Two broad approaches have been proposed in the past to improve trace buffer utilization: (i) *Compression* – trace messages are compressed before storing them in the buffer [3, 4], and (ii) *Event Triggers* – trace messages are stored (or discarded) only on the occurrence of events of interest [5]. Both these approaches improve utilization by increasing the effective capacity of the trace buffer. Another strategy used to reduce the overhead of on-chip trace buffers is to configure a part of the cache as trace buffer [6, 7] during post-silicon validation. This way, additional area is not invested on separate trace buffers. Event Triggers and compression schemes can be applied over these proposals too in order to reduce the data stored in these buffers.

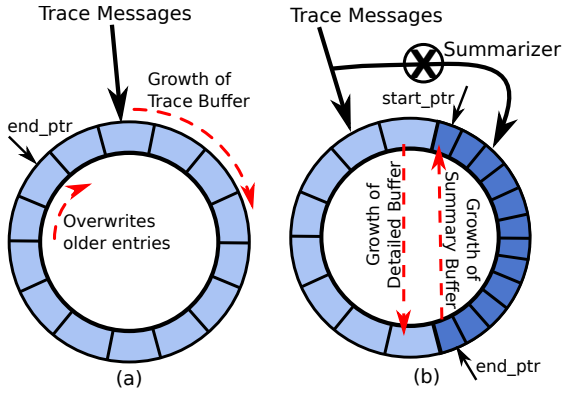

Fig. 1. High-level schematic of proposed scheme. (a) Trace buffer organized as a circular queue. (b) Flexible partition of the circular queue into Detailed and Summary Buffers. Both versions are stored simultaneously.

We propose a new approach to improve the utilization of trace buffers. We store the summaries of trace messages before over-writing 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 smaller than detailed traces, the activity window for which trace information is captured on-chip is extended. Our experiments indicate that storing summaries can extend the window by upto 162%.

We propose architectural features that allow the validation engineer to choose: (i) the amount of space for storing summaries, and (ii) the information to be captured in the summary. We achieve this by reusing the existing trace buffer through simple hardware extensions as shown in Figure 1. Figure 1(a) shows the default trace buffer pictured as a circular buffer. In

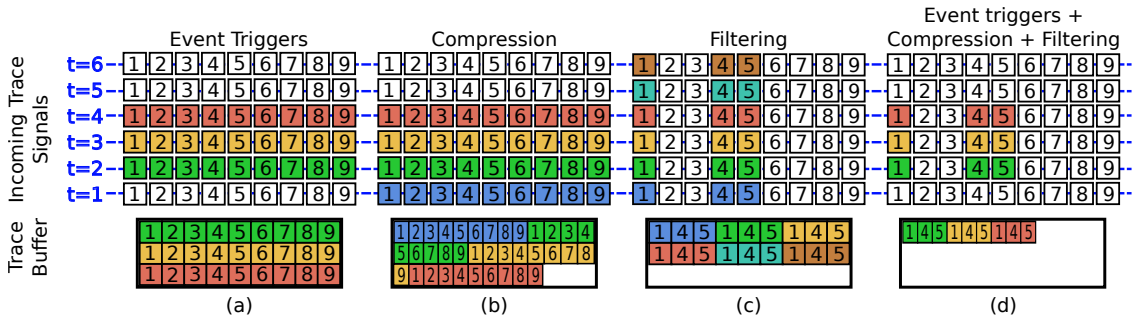

Fig. 2. Different methods to utilize trace buffers efficiently. Shaded boxes indicate the bits of the trace message that are stored in the trace buffer under each scheme

our proposed architecture pictured in Figure 1(b), the space is partitioned at run time into multiple circular buffers, to simultaneously capture both detailed and summary traces. We also demonstrate that our proposed methodology: (i) supports efficient debugging of different classes of errors encountered during post-silicon validation, and (ii) reduces the number of stalls required to transfer the captured traces off-chip by upto 63%.

II. Related work

Several researchers have proposed to significantly improve the effective capacity of the trace buffer through Event Triggers [8, 9] and Compression [3]. However, these works are targetted at errors that can be deterministically repeatable. These techniques use coarse-grained signatures (or summaries) in the initial executions to gather sufficient information on the time intervals where manifestations of the bug are captured in the trace signals. The information on the time intervals thus gathered is then passed on to subsequent executions during which more detailed traces are collected. Our proposed technique targets non-repeatable bugs as well because we do not gather information on time intervals that are suspected to contain the footprints of the bug. We only use the most recent detailed traces captured in the initial execution to identify the fields that should be captured in the summaries during subsequent executions.

Other recent works have used summaries to debug errors encountered during post-silicon validation, most remarkable of which is the use of aggregates captured by performance counters in a chip 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 erroneous state, is not retained in such summaries. Therefore, these summaries would have to be used in conjunction with other techniques. Similarly, multi-level traces (or tapered summaries) have been demonstrated to be useful for transaction-based debugging of on-chip interconnects [10]. However, the information captured in the traces at coarser granularity, is frozen at design time and cannot adapt to different requirements of different errors observed during post-silicon validation. This restricts the validation engineer to debug the observed error using only the information provided by these traces, which reduces the efficiency of debug.

Other works that use hardware support for checking temporal properties [11, 12, 13] for at-speed debug, can be extended to generate trace summaries. However, these proposals too suffer from limited flexibility because the information captured in the summary is fixed at design time. Our work provides the validation engineer the much needed flexibility to specify, at runtime, the most relevant information to be captured in the summaries for the particular execution of the failing testcase.

III. Trace Summaries

Figures 2(a) and (b) shows the bits of a trace message stored by Event Triggers and Compression respectively, and Figure 2(c) illustrates an example of a summary that captures bits 1,4, and 5 of the detailed trace. At present, such filtering of captured information is done during off-chip analysis by which time significant on-chip resources would have already been invested in storing and transferring them off-chip. This inefficiency arises because the on-chip DfD hardware does not allow the validation engineer to specify the bits (at runtime) within a trace message that are of interest to him in a particular scenario.

The subset of bits that the validation engineer is interested in can change from one debug scenario to another for multiple reasons such as: the validation engineer has previously analyzed the observed error and has deemed certain bits of the trace message as irrelevant; or that he has gained some insights into the bug based on the outputs produced prior to the erroneous behaviour, and so forth. This work seeks to address this requirement in an area-efficient manner.

Moreover, our proposed technique is complementary to event triggers and compression and can be used in conjunction with them to further improve the utilization of trace buffer as shown in Figure 2(d).

IV. Architectural considerations

We explore three architectures: (i) Split, (ii) Unified and (ii) Overlapped to store tapered summaries.

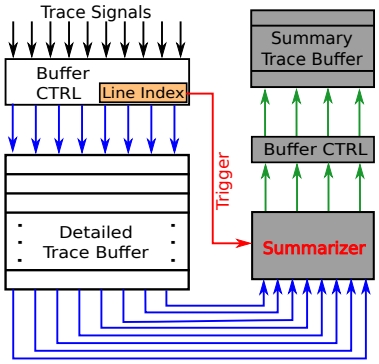

Fig. 3. Split architecture

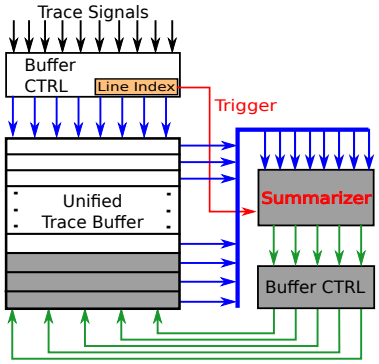

Fig. 4. Unified architecture

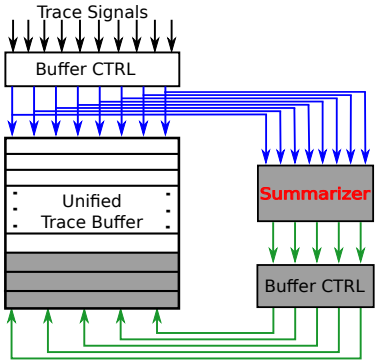

Fig. 5. Overlapped architecture

A. Split Architecture

Figure 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 is regulated using two thresholds. When the number of trace messages in the Detailed Trace Buffer exceeds the higher threshold, the Summarizer makes space for incoming messages by generating summaries and storing them into the Summary Trace Buffer; it stops doing so when the available space falls below the lower threshold. The sizes of the Detailed and Summary Trace Buffers are frozen at design time.

B. Unified Architecture

Figure 4 shows the ***Unified*** architecture for storing tapered summaries. This architecture allows the validation engineer to configure the sizes of the Detailed and Summary Trace Buffer 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 Detailed Trace Buffer, and the oldest detailed trace is simultaneously summarized and written into the Summary Trace Buffer, in the same clock cycle. The architecture retains the two thresholds of the Split architecture to maintain flow control between the Detailed and Summary Trace Buffer. This architecture requires a trace buffer with three ports to support online operation of the Summarizer: two write ports (one each to store incoming detailed traces and summaries generated by Summarizer), and one read port to read detailed trace messages into the Summarizer. This leads to an increased area overhead of the Unified architecture.

C. Overlapped Architecture

Figure 5 shows the ***Overlapped*** architecture. This architecture retains the flexibility of the Unified architecture but reduces the area overhead by discarding the dedicated read port

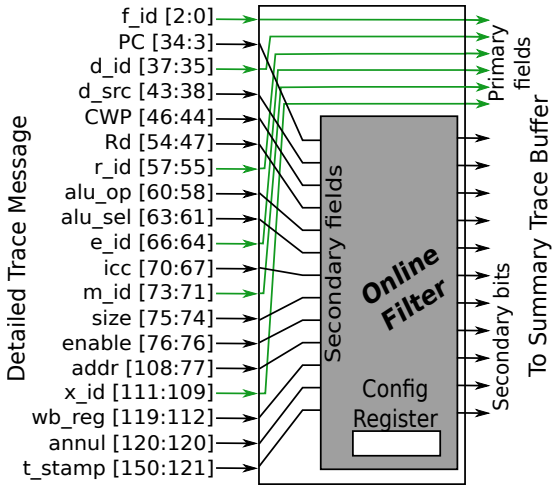

Fig. 6. Design of the Summarizer

of the Unified architecture. The Overlapped architecture generates and stores the summaries simultaneously with the detailed trace messages being written to the Detailed Trace Buffer. This avoids the need for the Summarizer to separately read the detailed trace message again later. Both the Detailed and Summary Trace Buffers 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 Summary Trace Buffer are transferred off-chip. We transfer the contents of the Detailed Trace Buffer off-chip only after the error being detected.

D. Extensions to multi-core systems

In extending the Summary Trace Buffer architecture to multi-core systems, we have considered two broad approaches: (i) Centralized and (ii) Distributed. Under the Centralized approach, a single Summary Trace Buffer is shared across all the

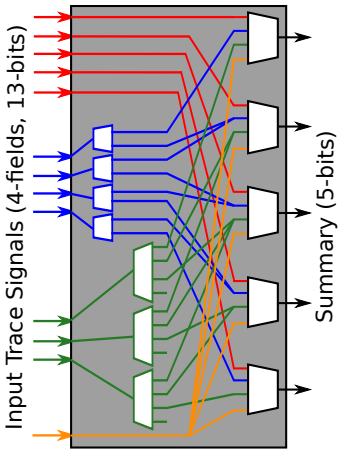

Fig. 7. Design of the Online Filter

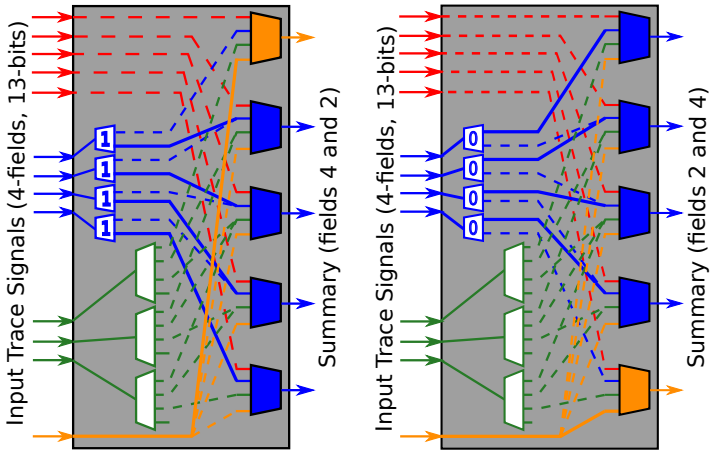

Fig. 8. Illustration of working of the Online Filter

cores to store the summaries generated from their respective detailed trace messages. This approach has a simpler design but has some disadvantages: (i) it is inflexible because the buffer sizes are frozen at design time, (ii) there is contention between summaries from multiple cores, and (iii) it is cumbersome to achieve flow-control due to communication delays between the Buffer Controllers. Under the Distributed approach, the Summary Trace Buffer is duplicated along with the Detailed Trace Buffer across multiple cores. This overcomes some of the disadvantages of the Centralized approach, and also creates additional opportunities, but at the expense of more synchronization logic.

V. Hardware Design

Figure 6 shows the hardware design of the Summarizer. 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 message which is essential to reconstruct the sequence of events during off-chip analysis. Therefore, it is made part of every summary that is generated by transferring them as-is to the output.

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 have prohibitively high area overheads when the number of input lines is large; we propose an alternative design that minimizes this overhead by imposing some constraints: (i) the order of bits within a field remains unchanged, and (ii) the relative order of the field themselves could change. The validation engineer can program the order of fields in the output. Permutation of fields is important to enable compaction before further on-chip processing such as compression. Only the number of bits in the output, and the trace signals appearing at the inputs are fixed at design time.

Figure 7 shows the hardware design of the Online Filter. The output of each multiplexer (MUX) is a bit of the summary. The 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 field themselves 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. Figure 8 illustrates an example of the permutations between field 2 and 4 at the output. Such permutations allow the width of filtered trace to be smaller than the number of output bits of the Online Filter.

VI. Experiments

A. Setup

Our experimental setup consists of a LEON3 SoC with 4-cores (SPARCv8), DDR controller and an AHB. The off-the-shelf LEON3 SoC generates a dynamic instruction trace, and a trace of transactions on the AHB which is stored into the Debug Support Unit (DSU). We enhanced this DfD hardware by tracing critical signals from the pipeline (similar to [14]), and cache controllers. Table I shows the number of bits captured by each of these detailed traces. We perform differential compression [15] to eliminate redundant temporal information before storing them into their respective trace buffers. We retain the distributed approach followed by the original LEON3 SoC, and use a 4kB trace buffer per core and AHB to store the generated traces. The information captured in the detailed trace is fixed at design time and remains the same for all the bug scenarios

TABLE I
DETAILS OF TRACES GENERATED BY THE DFD HARDWARE

Buffer Location	Trace Type	Detailed (#bits)	Primary (#bits)	Secondary (#bits)				Summary (#bits)				Area of Online Filter (mm^2)
				CCI	WIM	NAE	CSL	CCI	WIM	NAE	CSL	
Core	Instruction	128	32	-	-	-	-	114	85	114	83	0.0018
	Pipeline	151	18	32	3	32	-					0.017
	Cache	80	32	-	-	-	1					0.009
DSU	AHB	128	32	32	-	32	32	64	-	64	64	0.013

we discuss below. The bits captured in the summaries change across scenarios, as shown in Table I. The C applications 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 Cadence Encounter RTL compiler with a 90nm technology standard cell library.

B. Bug Scenarios

We modeled different realistic architectural bugs that could interfere with the functionality using a simple application with 4 threads: 2 writers and 2 readers. The writers increment a global counter and the readers continuously read it. The application was allowed to execute for 75000 cycles, and the bugs were introduced at random times. Such directed test applications are used extensively during post-silicon validation [2].

We considered three different configurations to capture trace messages into the on-chip trace buffers: (i) Detailed Trace Buffer (DTB) of 4kB with no Summary Trace Buffer (STB), (ii) STB of 3kB and Detailed Trace Buffer of 1kB , and (iii) Summary and Detailed Trace Buffer of 2kB each. 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.

Core-Cache interface (CCI): This bug is inspired by our experience from an actual design scenario involving adding a Victim Cache to the processor. 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. An error occurred at the interface between the core and the memory hierarchy due to which the core proceeded with its execution using the stale value present in its internal register. Previous studies have also found that a large number of functional bugs occur at the interface of the core [16].

After the initial run, we analyzed the most detailed trace messages available on-chip. This revealed that the contents of a register retained the same value across multiple increment operations. For subsequent executions, we captured a summary that included: (i) address and data values of the AHB trace, and (ii) address and data values of the pipeline trace. Only the captured summaries (a total of 114 bits) were transferred off-chip. 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. Further debugging proceeded with a similar approach until the bug was root-caused to an error in the victim cache logic.

TABLE II
NUMBER OF STALLS REQUIRED FOR DIFFERENT TRACE BUFFER CONFIGURATIONS

Bug	#Stalls			History (cycles)		
	DTB	Summary		DTB	Summary	
	4 kB	3 kB	2 kB	4 kB	3 kB	2 kB
CCI	138	51	77	1332	3491	2374
WIM	94	36	54	1109	2085	1916
NAE	138	51	77	1332	3491	2374
CSL	138	52	78	1332	3534	2307

Table II shows the number of stalls required to transfer the summaries off-chip, and the maximum duration of activity history captured in the trace buffer, for the three different configurations of the Overlapped architecture: (i) Detailed Trace Buffer (DTB) of 4kB, (ii) Summary Trace Buffer (STB) of 3kB and DTB of 1 kB, and (iii) DTB and STB of 2kB each. We observe that the number of stalls decrease by 63%, and the activity history is extended by 162% when a STB of 3kB is used. This reduction in stalls is because we dump only the contents of STB. This resulted in the total time spent on dumping the contents of trace buffer off-chip from $100.72s$ to $29.7s$, when transferred over LEON3 SoC's DSU serial link operating at 115200 bps.

Window Invalid Mask (WIM): The next bug scenario deals with the corruption of the Window Invalid Mask (WIM) of one of the SPARCv8 cores in our SoC. Such an error affects the Register Window operation, ultimately interfering with the correct return from deeply nested function calls, so the effect is manifested infrequently and after lengthy intervals. Such a bug could belong to the class of single bit-flips at flip flops [14, 17]. A corruption of the WIM during the system bootup, led to a crash after 32646 cycles.

A combination of a summary sequence that captured only the information on the Current Window Pointer (CWP), and the detailed instruction trace was extracted to debug the cause of the crash. We used event triggers to transfer the captured detailed traces off-chip only when CWP changed. The CWP was selected because the most recent update to the registers prior to the crash was by the service routine of a window underflow trap. Unlike *CCI*, this required us to capture the detailed traces and summaries simultaneously. The stall reduction is shown in

Table II.

Non-atomic execution of `ldstuba` (NAE): Determining the root-cause of race conditions introduced into applications due to hardware defects using at-speed debugging is a challenge because of their non-repeatable nature [18, 11]. We introduced a bug deep in the logic of the atomic exchange instruction (`ldstuba`) of SPARCv8 so as to break the mutual exclusion between the two writer threads.

A summary consisting of: (i) address and data values in AHB trace, and (ii) data values returned to the pipeline, sufficed to localize the bug. Trace messages of the increment operation appearing across multiple cores simultaneously, coupled with the AHB transactions helped point to a faulty implementation of `ldstuba`.

Cache snooping logic (CSL): We introduced a bug into the cache coherence logic, which in turn, led to a race condition. The objective was to test the robustness of our methodology against different bugs that result in similar manifestations. The race condition was introduced here due to data duplication, instead of mutual exclusion violations. This is possible because the semaphore is maintained in an alternate address space of SPARCv8 that bypasses the caches. The shared data is maintained in a cachable address space for performance reasons. A summary that captures (i) address and data values in AHB trace, and (ii) line numbers of cache lines invalidated due to snoop hits, was used to determine the root-cause.

None of the previous works on multi-level tracing [18, 13, 12] 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 Online Filter that generates upto 32 secondary bits for each type of trace. The last column of Table I shows the area overhead of each Online Filter for generating secondary bits of upto 32-bits. The total area consumed by all the Summarizers is 0.0408 mm^2. The Split architecture, with DTB and STB of 2kB each, occupies 0.216 mm^2 of which 0.139 mm^2 and 0.077 mm^2 are occupied by the DTB and STB respectively. The Overlapped architecture occupies 0.186 mm^2, which is less than that of Split architecture by 13.9% due to the reduced ports. The Unified architecture has the highest area overhead of 0.399 mm^2. The total area occupied by the trace buffers and the Summarizers is 0.785 mm^2, which is only 2.27% of the area occupied by a 64kB cache.

VII. Conclusion

We proposed a new debug methodology that uses a combination of detailed trace messages, and their summaries to maintain an extended trace history. This combination of traces is stored in the existing trace buffer using simple hardware extensions that occupy less than 2.27% of the area of 64kB cache, but extend the activity window for which trace history is available by upto 162%. 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 non-repeatable errors, and errors with long suspect windows that are encountered during post-silicon 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 *DAC 2011*.

[2] T. Bojan, M. Arreola, E. Shlomo, and T. Shachar, "Functional coverage measurements and results in post-silicon validation of core2 duo family," in *HLVDT 2007*.

[3] E. Anis Daoud and N. Nicolici, "On using lossy compression for repeatable experiments during silicon debug," *IEEE TC*, vol. 60, no. 7, 2011.

[4] K. Basu and P. Mishra, "Efficient trace data compression using statically selected dictionary," in *VTS 2011*.

[5] H. F. Ko and N. Nicolici, "Mapping trigger conditions onto trigger units during post-silicon validation and debugging," *IEEE TC*, vol. 61, no. 11, 2012.

[6] C.-H. Lai, Y.-C. Yang, and I.-J. Huang, "A versatile data cache for trace buffer support," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 61, no. 11, pp. 3145–3154, Nov 2014.

[7] R. Abdel-Khalek and V. Bertacco, "Functional post-silicon diagnosis and debug for networks-on-chip," in *ICCAD 2012*.

[8] J.-S. Yang and N. Touba, "Improved trace buffer observation via selective data capture using 2-d compaction for post-silicon debug," *IEEE TVLSI*, vol. 21, no. 2, Feb 2013.

[9] X. Liu and Q. Xu, "On multiplexed signal tracing for post-silicon validation," *IEEE TCAD*, vol. 32, no. 5, May 2013.

[10] B. Vermeulen, K. Goossens, and S. Umrani, "Debugging distributed-shared-memory communication at multiple granularities in networks on chip," in *NoCS 2008*.

[11] A. Gharehbaghi and M. Fujita, "Transaction-based debugging of system-on-chips with patterns," in *ICCD 2009*.

[12] M. Neishaburi and Z. Zilic, "On a new mechanism of trigger generation for post-silicon debugging," *IEEE TC*, vol. 63, no. 9, Sept 2014.

[13] C.-T. Huang, K.-C. Tasi, J.-S. Lin, and H.-W. Chien, "Application-level embedded communication tracer for many-core systems," in *ASP-DAC 2015*.

[14] S.-B. Park and S. Mitra, "Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *DAC 2008*.

[15] A. Hopkins and K. McDonald-Maier, "Debug support for complex systems on-chip: a review," *Computers and Digital Techniques, IEE Proceedings* -, vol. 153, no. 4, pp. 197–207, July 2006.

[16] S. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *MICRO 2006*.

[17] A. DeOrio, D. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *ICCAD 2011*.

[18] E. Larsson, B. Vermeulen, and K. Goossens, "A distributed architecture to check global properties for post-silicon debug," in *ETS 2010*.