New Approaches for In-System Debug of Behaviorally-Synthesized FPGA Circuits

Joshua S. Monson and Brad Hutchings
Department of Electrical Engineering
Brigham Young University
459 Clyde Building Provo, UT 84602
Email: jsmonson@gmail.com, brad_hutchings@byu.edu

***Abstract*—This paper present new approaches for in-system, trace-based debug of High-Level Synthesis-generated hardware. These approaches include the use of Event Observability Ports (EOP) that provide observability of source-level events in the final hardware. We also propose the use of small, independent trace buffers called Event Observability Buffers (EOB) for tracing events through EOPs. EOBs include a data storage enable signal that allows cycle-by-cycle storage decisions to be made on an EOB-by-EOB basis. This approach causes the timing relationships of events captured in different trace buffers to be lost. Two methods are presented for recovering these relationships. Finally, we present a case study that demonstrates the feasibility and effectiveness of an EOB trace strategy.**

I. Introduction

Improvements in the latest generation of High-Level Synthesis (HLS) tools have led to increased interest and use of these tools in both industry and academia[1]. HLS tools have been shown to provide a 5X boost in design productivity[2]. Some users have observed that the boost in productivity comes from the ability to verify the design specification in software and avoid time-consuming simulation runs[3]. From our own experience we know that much of this productivity can be lost if the user encounters a bug in their HLS design after the design is operating at full speed in the final system. The productivity loss stems from the fact that the user is required to build an intimate understanding of the final circuit before he can (manually) instrument the circuit for debugging. This is because current HLS tools lack an effective, integrated approach to in-system, source-level debugging. To be effective and integrated, we believe a debugging approach must at least begin at the source-level; since, eventually, the source is what will be modified to fix the bug. Debugging at lower-than-source levels threatens to impact the productivity for which HLS was used in the first place.

Embedded Logic Analyzers (ELA)s are the standard method of accomplishing in-system debugging of FPGA designs. ELAs are usually configured by selecting signals from the netlist to connect to the inputs of a trace buffer. These signals are then recorded on a cycle-by-cycle basis during the active execution of the design. The amount of relevent information that can be captured by an ELA is limited by the amount of available on-chip memory. ELAs can also be used, when necessary, to perform in-system debugging of HLS designs on FPGAs. However, users of HLS are generally not familiar enough with the HLS-generated RTL code to configure the ELA to capture the correct information in a reasonable amount of time. Additionally, the cycle-by-cycle capture approach of the ELA does not provide any flexibility for the use of high-level knowledge to optimize trace buffer efficiency even when such high-level knowledge is available from the HLS tool.

This paper presents new approaches for the in-system debugging of FPGA designs specifically optimized for debugging HLS-generated FPGA circuits. Our approaches provide enough flexibility to allow a trace buffer configuration to be optimized to capture as much relevant information as possible. The instrumentation process will utilize the high-level information available within the HLS tool to both optimize the trace buffer configuration for memory efficiency and resource usage. Since our approach relies on high-level information from the HLS tool it will also (eventually) be fully automated and debugging circuitry will be compiled into the design reducing the need for the user to be familiar with the low-level organization of the FPGA circuit.

II. Background

The general goal of commercially available HLS tools is to take a sequential software input specification written in a high-level language (e.g. C, JAVA) and generate efficient hardware. In general, the sequential software specification is a series of control flow and program state update events. We debug software by analyzing the sequence of these events to determine why they went wrong. To enable the user to perform the same kind of debugging analysis on the final hardware as he would in software we need to have access to the same control flow and state update events.

In general, the design flow for a typical HLS compiler proceeds as follows. First, the sequential input specification is compiled to an intermediate representation (IR) by a standard front-end compiler. The IR is represented in an assembly language-like format and contains the necessary operations to completely implement the input specification in software. The IR is usually represented in the form of a two-level control flow diagram (CFD) consisting of *basic blocks* (higher-level) and operations (lower level) [4]. Basic blocks are straight lines of code that terminate in a control flow decision operation (such as a branch) that determines the next basic block that will be executed. Next, the IR is passed to a scheduler that assigns the operations from the CFD to the control steps of a state transition graph (STG) [4]. Once scheduling is complete, the operations in the STG are bound to allocated functional units and registers. RTL code is then generated based on the results

of scheduling and binding. The RTL code is then instanced into a larger design and passed through the vendor tool flow to create a bitstream to program the FPGA.

In general, control flow and state update events are represented in the IR by the completion of operations. The operations corresponding to control flow events are generally the branching operations found at the end of basic blocks. State update events generally correspond to the completion of operations that store data in memory or in static single assignment (SSA) registers. In order to instrument the final hardware design for debugging, the instrumentation software needs to track the correspondence between source-level events and the operations whose completion indicate they have occurred. This correspondence can be maintained using the -g flag to instruct the compiler front-end to annotate the IR representation with debugging information. Maintaining the correspondence through the rest of the HLS flow is generally straight forward as the operations from the IR are assigned to specific functional units in the RTL.

Maintaining correspondence is complicated by the introduction of compiler and HLS optimizations to the design flow. Scott Hemmert previously addressed many aspects of HLS correspondence problem[5][6]. Others, have addressed this problem specifically for software debugging. However, a thorough review of the work of Hemmert and others as well as further investigation into the correspondence problem is beyond the scope of this work. In this paper, we move forward with the assumption that the appropriate correspondences have been maintained and leave further investigation into the correspondence problem to be addressed as future work.

III. Event Observability Ports

To provide visibility of the state update and control flow events we propose to instrument HLS designs with *Event Observability Ports* (EOP). EOPs consist of an *event signal* and a *data signal*. The event signal is a one-bit signal that is asserted when the operation corresponding to the event has completed. The event signal also validates the data signal which provides the result of the corresponding operation. In this way, EOPs provide an abstraction that links source-level correspondence information to the final hardware implementation.

The benefit of the EOP approach is that the ports are added to the design after it has been scheduled and bound. The effect of adding the EOP after scheduling and binding is that it will not modify the results of these processes and potentially obscure the bug the user is trying the find. Users are often reluctant to recompile and re-synthesize a debug-enabled version of their design, especially after place and route. Hung and Wilton[7] and Keeley and Hutchings[8] have shown that incremental debug insertion (i.e. insertion after place and route) provides both better performance and higher productivity compared to recompiling and inserting instrumentation into the netlist. Delaying EOP insertion until after RTL-code generation permits the possibility that EOPs could be incrementally inserted at these late points in the design flow.

To instrument a design with an EOP we need to identify or create the relevant event and data signals. An important source of information to complete this task is the STG. Using the definition of STG provided by Cong and Zhang [4], an STG is a directed graph with a set of *control states* with a set of *transitions* between them. Each transition is associated with a *transition condition*. Each control state also contains a set of *operations* (from the IR). Each operation is associated with a *guard condition* to control its execution. In general, the RTL code generation process implements the STG according to some structured circuit model (e.g. Finite State Machine with Data Path or Data Flow Models).

The guard conditions from the STG can be used to identify the completion of the operations that correspond to the control flow and state update events. One benefit of using the guarding conditions is that they are often implemented as register clock enables for the hardware registers that store the result of the operation in the final circuit. This is beneficial because registers survive logic synthesis more often than combinational logic while preserving name correspondence to the RTL. In these cases, the register inputs can be used as the data signal and the register clock enable can be used as the event signal. This approach is beneficial because it utilizes circuitry that already exists in the final circuit. In cases where the operation result signal and guard signals have been optimized out they must be re-created. This can be done by using the HLS-tools own library to independently generate RTL code for the operation and guard signal using the register signals still in existence as input. Then the additional generated circuitry can be advanced through the vendor tool flow to the point where the circuit will be instrumented. In general, this may not always be possible. But, we believe that it will cover a majority of the necessary signals.

An important property of EOPs is the *relative assertion rate* (RAR) of the event signal. The assertion rate of an event signal is calculated by dividing the number of occurrences of the event signal by the execution latency of the hardware. The RAR of an event signal A is then calculated as the ratio of the assertion rate of A to the lowest assertion rate in the design. In the general case, assertion rates are not always statically determinable. In these cases, a user-guided approach may be useful. For example, when a for-loop has a variable loop bound, Vivado HLS allows the user to specify the expected number of loop iterations using the *TRIPCOUNT* directive [9]. The TRIPCOUNT directive provides a number to Vivado HLS so the tool can calculate an estimate of the latency of the design. A similar, directive-based approach, could be used to assist instrumentation software in estimating RAR. The RAR data would then be used to efficiently instrument the EOPs for trace.

IV. Tracing Event Observability Ports

The primary purpose of developing EOPs is to enable the efficient trace of source-level control-flow and state update events. We propose the use of Event Observability Buffers (EOB) for tracing EOPs. An EOB is a small, independent trace buffer with a data input and data storage enable input. There are advantages of using EOBs over the monolithic trace buffer of an ELA. The small size of EOBs often allows an EOB to be allocated on an EOP-by-EOP basis. This allows the storage depth of the EOB to be configured according to the RAR of the EOP's event signal. For example, an EOB could be configured with a greater depth when the RAR of an EOP's event signal

is high. Additionally, by connecting the EOPs event signal to the data storage enable input of the EOB the event data signal is only stored when an event has actually occurred. Small, independent trace buffers also have advantages for incremental insertion after place and route[7][8]. On the other hand, the centralized, monolithic nature of the ELA prevents it from being able to make storage decisions on an EOP-by-EOP basis. This means that the ELA must trace both the event and data signals and will likely store event data signals when they are invalid thereby wasting the limited trace buffer memory. Additionally, the ELA model provides equal storage depth for all trace buffer inputs. Therefore, ELAs cannot adjust storage depths on a per input basis.

EOBs are most useful when the RAR of different EOPs is vastly different. For example, consider an HLS design that computes an 8-bit *unsigned char* value to determine the loop-bound of a for-loop that computes a 16-bit *short int* on each iteration. To enable the trace of these events EOPs are added for each event. The maximum RAR of the 16-bit event (compared to the 8-bit event) is 256. This means that the 16-bit event will occur a maximum of 256 times for each of the 8-bit events. The user could also provide an expected maximum number of occurrences of the 16-bit event via a directive. For the purposes of this example let us assume the user has provided a maximum RAR of 32. The individual configurability of EOBs allows the instrumentation software to allocate 32 16-bit EOB entries for each 8-bit EOB entry. Sizing EOBs in this manner helps them fill at approximately the same rate.

Another advantage of the EOB approach is the ability for EOPs to share EOBs when their event signals are asserted during mutually exclusive clock cycles. EOB sharing is accomplished by multiplexing the EOB data inputs and using the event signals to appropriately select each input. The data storage enable signal is then created by logically ORing both event signals together. In the previous example, assume that the 8-bit loop bound event occurs a cycle before the first 16-bit event ensuring that the event signals of the EOPs are asserted during mutually exclusive clock cycles. Having these two EOPs share the same EOB avoids the situation of wasting an entire BRAM to capture the 8-bit event or relying on the distributed memory within the FPGA that might not be available. Adding a multiplexor in front of the EOB increases the potential that the EOP data signal will become a critical path in the design. One approach to mitigate this problem is to apply multi-cycle path constraints to the EOP data input paths. Zheng et. al [10] described an approach to identify and add multi-cycle paths to designs generated by HLS tools. A similar approach could potentially be used to identify opportunities to apply multi-cycle paths to these EOB input probes.

V. Recovering Event Order and Timing

A primary challenge of the EOB approach is that the timing relationship between events is lost. This relationship is inherently preserved by the ELA approach because of the use of a monolithic trace buffer. This problem can be remedied using what we call an *event reference trace*. An event reference trace uses a single EOB to trace the event signals of the EOPs. This is accomplished by connecting the event signals of the EOPs to the data input of an EOBs allocated specifically

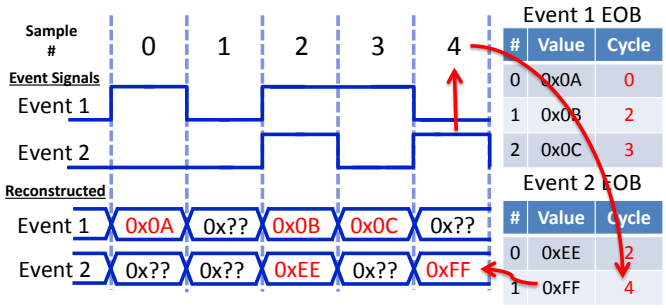

Fig. 1: Example of event order and timing recovery using a cycle-accurate event reference trace.

for this purpose. Then the same event signals are logically ORed together to create the input for the data storage enable signal of the EOB. The logical ORing of the event signals creates a data storage enable input that filters the storage of data during cycles in which no events occur. When needed, a cycle-accurate reference trace can be created by replacing the previous data storage enable signal with a constantly asserted enable signal.

After the EOP trace is complete, the event reference trace can be used to recover event results from the corresponding EOB. This is done by matching the last assertion in the event reference trace with the last value written to the corresponding EOB. This step is repeated moving backwards through each sample of the event reference trace. Figure 1 demonstrates how the recovery algorithm works. The two event signals in Figure 1 represent the contents of a cycle-accurate event reference trace after it has been uploaded. The recovery algorithm begins by examining the contents of the final sample (4) of each event signal. In this case, Event 2 is asserted indicating that an data corresponding to this event is stored into the last entry of the Event 2 EOB. This data is then labeled with the event trace sample number to which is corresponds. The process is continued by repeating these steps on previous event trace samples until the beginning of the event trace is reached. This approach works as long as both trace buffers stop recording at the same time.

Another approach to recovering event order is to use high-level knowledge of relationships between events. Consider, for example, the trace of a completely unoptimized version of a design in which events in the hardware occurred in the same order they do in the original specification. Under these circumstances, the user could step through the source code and reconstruct the order of events by reading them from the EOBs as they are encountered on his walk through the source code. When a control flow decision is encountered the user would use the result from the hardware to ensure he proceeds down the same control flow path as the hardware. The benefit of such an approach is that an event reference trace is not required. However, scheduling and binding optimizations practically guarantee that events will not occur in the same order in the hardware as in the source.

Recall that the operations that correspond to the source-level events are contained in the STG. Since hardware is generated from the STG we can expect that the sequential

relationships of events are also preserved. Therefore, by walking the STG we can perform the same kind of analysis as previously described on the source code and avoid having to use an additional EOB to perform a reference trace. At least two conditions must be fulfilled to perform this kind of analysis. First, all EOBs must start their traces at the beginning of the HLS design operation. Second, all EOPs representing control flow decisions must be included in the trace. In many cases the STG may also be used to infer cycle-accurate timing information for many control flow sequences. This information may not be available when, for example, the control flow stalls while waiting for a memory read over a bus.

VI. Case Study

In this section, we present a short case study which evaluates the use of EOBs for tracing the control flow and state update events of HLS designs instrumented with EOPs. In this case study we evaluate the effect of preserving the signals required to create the EOPs. We also compare the efficiency of capturing events using EOBs to standard ELA approaches. To do this we have instrumented three small HLS benchmarks with EOPs. These designs include a fully-pipelined *fir filter*, a partially-pipelined *sample mean and population variance estimator*, and an un-pipelined *floating-point accumulator*.

A. Experimental Setup

Each benchmark was originally written in C and compiled to RTL using Vivado HLS 2013.2. Vivado HLS was configured to target a ZYNQ 7020 device with a minimum clock period of 10 ns for the fully-pipelined design and 25 ns for the partially-pipelined and un-pipelined designs. The lower clock rate target for the partially-pipelined and un-pipelined designs reduced the latency of the floating point operations in these designs and avoided inserting empty clock cycles (in which no events take place) which could reduce the number of events captured by our baseline approach and artificially inflate our results. Enabling pipelining in the fully-pipelined and paritially-pipelined designs required that we add the *HLS PIPELINE* directive to each design. To achieve an initiation interval (II) of one for the fully-pipelined design required that we apply the *HLS PARTITION* directive to two arrays in the design. Other than the above mentioned exceptions our experiments relied on the default behavior of Vivado HLS 2013.2.

Next, the design was instrumented with EOPs corresponding to the control flow and state update events we desired to trace. Each HLS benchmark was instrumented using a four step process. First, the C-code was examined to determine a set of events that provide a good picture of the execution of the design. Second, the mapping between events in the C-code and RTL was determined. Third, the design was synthesized using XST (14.6) and an EDIF net-list was written. Finally, the EOPs are added as top-level ports of the EDIF netlist. This is done by searching the EDIF netlist for the needed signals and routing them to the top-level ports. In addition to the EOP signals ports are also created for the clock and start signals. If a needed signal was not found in the design a *keep* attribute was added to the RTL source. The RTL was then re-synthesized and the EDIF file recreated.

Design	REGs	LUTs	Min. Clk. Period
Un-Pipelined	358	332	18.738 ns
Partially-Pipelined	817	1908	19.886 ns
Fully-Pipelined	922	696	8.726 ns

TABLE I: Resource Utilization of Baseline Designs

Design	ELA	Mod. ELA	EOB
Un-pipelined	388	752	1505
Partially-Pipelined	750	1424	1507
Fully-Pipelined	1533	1533	1538

TABLE II: Table of Events Captured

Once the design had been instrumented with EOPs an event-trace strategy was selected and trace buffers are configured according to that strategy. The EOP-instrumented EDIF file was then instantiated within a VHDL wrapper containing the trace instrumentation. The wrapper is then run through the Xilinx PlanAhead design flow to generate a bitstream. The bitstream was then downloaded to a ZedBoard and a trace experiment was run. The goal of each trace experiment was to determine the number of control flow and state update events captured by the currently implemented trace strategy. The testbench circuitry is configured so that each experiment is triggered by the DUT start signal and ends when an EOB is full. The contents of the trace buffers are then uploaded to the host machine and the events are tallied.

B. Results

Two sets of experiments were performed on each test design. The first set demonstrates the effect of the keep attribute to preserve name correspondence from the RTL to the net-list. The remaining experiments test the efficiency of different trace strategies used to trace EOPs. Baseline design statistics are presented in Table I. The experimental results in Figures 2 and 3 are normalized to the design statistics in Table I.

The first set of experiments was performed to determine the effect of using the keep attribute to preserve name correspondence between the RTL and netlist. Name correspondence was preserved to ensure that the correct signals were used to instrument the circuit with EOPs. Figures 2 and 3 show that the keep attribute had very little effect on LUT usage or the minimum clock period. This is because many of the signals to which the keep attribute was applied already existed under a different name. The worst effect of the keep attributes was seen in the fully-pipelined design which saw a 4% increase in LUT usage.

In the remaining experiments, EOPs were traced using one of three different trace strategies. The first strategy was the standard cycle-by-cycle ELA trace strategy. In this strategy, all EOP event and data signals were connected directly to the EOB inputs. The EOB enable signals were then tied high to force the EOBs to store all EOP signals on every clock cycle after the trigger (start signal) regardless of whether or not an event had occurred (just like an ELA would). The second (Modified ELA) strategy was identical to the first strategy except that rather than tying the EOB enable signals high we logically OR the event signals from all EOPs and connect the result to the

Design	ELA	Mod. ELA	EOB
Un-pipelined	3	3	3
Partially-Pipelined	5	5	4
Fully-Pipelined	2	2	4

TABLE III: Block RAMs used

enable signal of each EOB. This strategy prevents the EOBs from storing data during cycles when no events are occurring. In the final strategy, the EOBs are instrumented as described earlier in this paper. The event and data signals of each EOP are respectively connected to the enable and data inputs of the EOB. This strategy also took advantage of the EOB sharing techniques discussed earlier in this paper. The benefit of this strategy is that it only stores data when events actually occur. A EOB was also used to capture a cycle-accurate event reference trace in experiments where the EOB strategy was used.

Along with EOBs, each benchmark was also instrumented with circuitry to control the experiment (i.e. detect a trigger and start the trace) and upload the results from the trace buffers. The cost of this additional circuitry (in LUTs) can be seen in Figure 2 by comparing the LUT usage of the baseline circuit with keep attributes to the ELA experiments. The resource requirements of the additional circuitry was determined by building each component independently from the test design and other instrumentation. Together these circuits required 82 LUTs. This number, however, does not include the multiplexor used to select a particular EOB for readback. The size of this multiplexor will vary depending on the number of EOBs used for the trace. The difference in LUT usage between the baseline with keep attribute and the ELA benchmark varied anywhere from 41 to 132 LUTs. It is hard to know how much of that increase to attribute specifically to the control and upload circuitry since the synthesis tool has a second chance to operate on the benchmark netlist when it synthesizes the wrapper.

In all cases, the EOB strategy increased the number of source-level events captured (Table II). The EOB strategy was most successful on the un-pipelined design where it captured 3.88 times more events than the ELA strategy and 2 times more events than the modified ELA strategy. The EOB strategy was also successful for the partially-pipelined design where it increased the number of events captured while using one less BRAM (Table III). The partially-pipelined design is particularly interesting because it represents a more realistic use case as it alternates between dense and sparse regions of events. The sparse nature of the event's in the un-pipelined and partially-pipelined designs created a lot of opportunities for the EOB strategy to optimize the trace. On the other hand, the fully-pipelined design contained fewer opportunities for trace optimization. For example, there were no opportunities for buffer sharing. Implementing the EOB strategy in this instance only resulted an increase of five events overs the ELA strategies.

Figures 2 and 3 show a great deal of resource and minimum clock period parity between the trace strategies. This is a positive result. However, the designs presented in this case study are small. Thus, more work will be needed to determine how the EOB strategy scales to larger designs.

In summary, we have found that the EOB strategy is

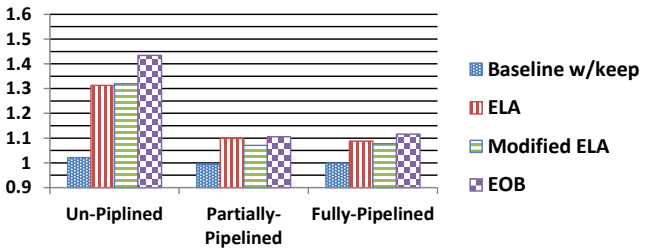

Fig. 2: LUT usage for each experiment normalized to the baseline.

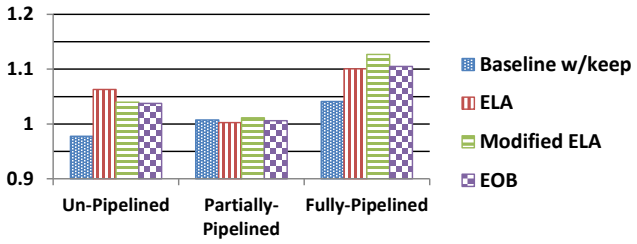

Fig. 3: Minimum clock period for each experiment normalized to baseline.

effective for designs that have sparse event regions (e.g. the un-pipelined and partially-pipelined designs) and not effective for designs that contain only densely-populated event regions (e.g. the fully-pipelined design). The fact that each trace strategy was successful under different circumstances suggests that achieving maximum event visibility requires the application of the right trace strategy. A proper analysis of information available from HLS tools has the potential to allow the creation of an automated approach to selecting trace strategies in a way that is not immediately possible for hand-coded RTL designs.

VII. Related Work

The instrumentation of HLS designs for debug is not a new topic. Instrumenting HLS ciriuits for debug has also been previously proposed[11][12][6][13][14]. However, most previously proposed methods are scan-chain[11][12][6] or bitstream readback[14] based. These approaches require fined-grained clock control which is not always available at late stages of the design flow. A problem with such approaches is that they are not able to capture bugs that occur only when the design is running at full speed in the final system. Researchers at the University of Florida also investigated assertion based debugging[13]. Assertion based debugging can be used while operating the design at full speed in the final system and would be compatible with the approaches presented in this paper.

A common limitation of trace-based approaches is the limited on-chip memory that can be used to implement the trace buffer. The focus of this paper has been to increase the efficiency of the use of this limited memory Trace compression is common approach for achieving this end[15]. Trace compression has even been implemented on FPGAs[16][17][18]. Trace compression is an orthogonal approach and can be used in tandem with the approaches presented in this paper.

Prabhakar has also suggested a scheme in which the inputs of an ELA style trace can be shared[19]. However, his approach is limited to sharing a trace buffer input between two signals. The EOB sharing approaches in this paper can handle an arbitrary amount of sharing.

VIII. Conclusions and Future Work

This paper has presented a new approach for tracing source-level events during the in-system execution of HLS designs. Our approach instruments HLS generated hardware with EOPs that provide observability of source-level events. To capture the run-time execution of the HLS-generated hardware we have proposed the use of small, independent trace buffers known as EOBs. The most important feature of EOBs is that they can be configured to capture data only when an event occurs. Additionally, the depth of the buffer can be independently configured based on the needs of the source-level event it is assigned to capture. These two features allow EOBs to capture events in a more memory efficient way than the standard ELA approach. We have also proposed two new approaches to recover the order and timing of source-level events captured into different EOBs. In addition, we have introduced the concept of RAR which can be used to help configure EOB depth efficiently.

We also found that EOBs can be used to implement different strategies for tracing source-level events through EOPs. Our most successful strategy demonstrated the ability to capture 2.0 and 3.88 times more source-level events than standard ELA-like configurations. However, the success of this strategy was limited to HLS benchmarks that contained at least some degree of control flow. We also found that for highly-pipelined designs EOBs can also be configured to implement a more standard ELA-like strategy that removes the need for an event-reference trace and lowers the performance overhead.

The case studies presented in this paper were performed manually (for the most part) to determine the feasibility of our ideas. The manual instrumentation process limited the size and number of designs which we could instrument. The primary focus of future research will be to identify methods of automating the instrumentation process. Automation of the instrumentation process will permit us to further vet our ideas on a larger number of larger designs. In the process of automating the instrumentation process we expect to investigate a number of exciting new research areas. The most important of these research areas is determining techniques for maintaining as much correspondence through optimization between source-level events and final hardware as possible. Another important area of future research is the automated configuration of EOBs. Techniques need to be identified and evaluated for estimating RAR and determining when and how EOBs can be shared. Additionally, we plan on investigating methods to use RAR to identify which EOPs should be traced by EOBs and which should be traced by connecting them to a high-speed serial I/O such as SERDES. This has the potential to preserve more of the on-chip memory for where its needed most. Finally, we also plan on investigating methods of incrementally inserting EOPs and EOBs to improve the productivity and usefulness of our debugging instrumentation.

Acknowledgment

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1265957.

References

[1] J. Cong, L. Bin, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhiru, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[2] K. Rupnow, L. Yun, L. Yinan, and C. Deming, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Conference Proceedings, pp. 1102–1105.

[3] J. Noguera, S. Neuendorffer, S. Haastregt, J. Barba, K. Vissers, and C. Dick, "Implementation of sphere decoder for mimo-ofdm on fpgas using high-level synthesis tools," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 119–129, 2011.

[4] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," pp. 433–438, 2006.

[5] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, and P. A. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, Conference Proceedings, pp. 228–237.

[6] K. S. Hemmert, "Source level debugging of circuits synthesized from high level language descriptions," 2004.

[7] E. Hung and S. J. Wilton, "Incremental trace-buffer insertion for fpga debug."

[8] J. Keeley, "An incremental trace-based debug system for field-programmable gate-arrays," 2013.

[9] Xilinx, "Vivado design suite user guide: High-level synthesis," vol. UG902–Version 2013.4, 2013.

[10] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in *FPL*, 2013, Conference Proceedings.

[11] G. Koch, U. Kebschull, and W. Rosenstiel, "Debugging of behavioral vhdl specifications by source level emulation," in *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, Conference Proceedings, pp. 256–261.

[12] C.-T. Chen, K. K #252, #231, #252, k #231, and akar, "A source-level dynamic analysis methodology and tool for high-level synthesis," pp. 134–140, 1997.

[13] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis techniques for in-circuit assertion-based verification," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, Conference Proceedings, pp. 1–8.

[14] Y. S. Iskander, C. D. Patterson, and S. D. Craven, "Improved abstractions and turnaround time for fpga design validation and debug," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Conference Proceedings, pp. 518–523.

[15] E. Anis and N. Nicolici, "On using lossless compression of debug data in embedded logic analysis," in *Test Conference, 2007. ITC 2007. IEEE International*, Conference Proceedings, pp. 1–10.

[16] Y.-T. Lin, W.-C. Shiue, and I.-J. Huang, "A multi-resolution ahb bus tracer for real-time compression of forward/backward traces in a circular buffer," pp. 862–865, 2008.

[17] N. Ohba and K. Takano, "Hardware debugging method based on signal transitions and transactions," in *Design Automation, 2006. Asia and South Pacific Conference on*, Conference Proceedings, p. 6 pp.

[18] G.-R. Tsai, L. Min-Chuan, and C.-H. Lin, "A real-time two-level trace compressor for fpga-based soc on-chip debugger," in *Innovative Computing, Information and Control, 2007. ICICIC '07. Second International Conference on*, Conference Proceedings, pp. 267–267.

[19] S. Prabhakar, "Algorithms and low cost architectures for trace buffer-based silicon debug," Thesis, 2009.