2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

# Enabling Long Debug Traces of HLS Circuits Using Bandwidth-Limited Off-Chip Storage Devices

Jeffrey Goeders
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
jgoeders@byu.edu

***Abstract*—High-level synthesis (HLS) has gained considerable traction in recent years. Despite considerable strides in the development of quality HLS compilers, one area that is often cited as a barrier to HLS adoption is the difficulty in debugging HLS produced circuits.**

**Recent academic work has presented techniques that use on-chip memories to efficiently record execution of HLS circuits, and map the captured data back to the original source code to provide the user with a software-like debug experience. However, limited on-chip memory results in very short debug traces, which may force a designer to spend multiple debug iterations to resolve complicated bugs.**

**In this work we present techniques to enable off-chip capture of HLS debug information. While off-chip storage does not suffer from the capacity limitations of on-chip memory, its usage introduces a new challenge: limited bandwidth. In this work we show how information from within the HLS flow can be leveraged to generated a streamed debug trace within given bandwidth constraints. For a bandwidth limited interface, we show that our techniques allow the user to observe 19x more source code variables than using a basic approach.**

## I. Introduction

High-Level Synthesis (HLS) has gained considerable traction in recent years. By specifying an algorithm implementation at the software abstraction level, designers can create digital circuits faster, and with less complexity, than designing at the register-transfer level. Although this technology is attractive for hardware designers interested in faster design times, it is also of particular interest to software designers who may want to accelerate a software application using custom-designed hardware, but lack the necessary hardware design expertise. This is becoming increasingly relevant as FPGAs, which are often used to implement HLS circuits, have been deployed in a growing number of large data centers, including those by Microsoft and Amazon.

This interest has led to an increased initiative toward high-quality HLS compilers, and has resulted in C-based and OpenCL-based design flows from the major FPGA vendors, as well as several academic tools. Much of this recent work has focused on the quality of the compiler tool, with the goal of producing the best performing circuit. However, if HLS is to receive widespread adoption, including significant use within software communities, a full ecosystem of design tools is required. This includes not only the core HLS compiler technology, but also technologies to enable optimization, validation, and debug. In this work we present novel techniques to aid users in debugging HLS produced circuits.

While the source code input into an HLS tool can often be debugged using established software debugging tools (GDB), certain bugs may not be resolvable with such techniques. Bugs that depend on interaction between hardware components in the final operating system, data-dependent bugs that are not reproduced with the user's test-vectors, or bugs that require long run-times to expose, may need to be resolved by observing the circuit in the final operating environment [1]–[3].

In-system HLS debug is challenging, both because the circuit may not resemble the original source code, and because it is typically not possible to obtain full observability into the internals of an executing circuit. Several recent works have addressed these challenges [1]–[9]. The approach used in these works has been to use on-chip memories to record a portion of the circuit execution, and then map the recorded information back to the original source code, such that the user can debug in the context of their program. Since FPGAs typically contain many small memories, there is sufficient memory bandwidth to record many signals at once; however, since total FPGA memory capacity is very limited, only a short duration of the circuit execution can be captured. Even with many optimizations employed, these debug traces typically only capture microseconds of execution [3].

Short execution traces can make debugging a time consuming and laborious process. Designers may have to undertake multiple debug iterations, where the system is executed, a small portion of execution is captured, and the *post mortem* data is analyzed. In such scenarios, being able to capture

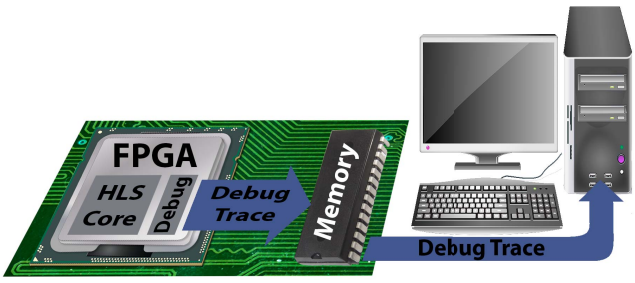


Fig. 1. In-system debug of an HLS circuit using off-chip storage

978-1-5386-4037-1/17 $31.00 © 2017 IEEE
DOI 10.1109/FCCM.2017.29

significantly longer execution traces may reduce the number of debug iterations, saving time in the development process.

In this work we present a framework to use external storage resources to capture execution traces from HLS circuits in real-time (Figure 1). External devices, such as a DDR memory, or a network interface, may provide several orders of magnitude more storage capacity than is available in on-chip FPGA memory. When storing execution traces in on-chip memory, as has been done in past work, one has the advantage of simple scalability; if more variables are to be observed, more on-chip memories can be allocated to capturing the trace, thereby increasing the bandwidth of the storage to meet demand. However, when streaming execution traces to off-chip storage devices, one does not have this luxury; these devices have fixed bandwidth, which is often much lower than the combined bandwidth of many distributed memories inside the FPGA.

This paper presents several contributions to enable and optimize the use of external devices in capturing execution traces of HLS circuits:

- A signal-tracing architecture which collects debug data from HLS kernels, assembles the data in an efficient manner, and streams it to an off-chip storage resource.
- A static analysis algorithm to determine the worst-case bandwidth of debug data produced by an HLS kernel, given a user-selected set of variables to observe, including analysis to determine necessary buffering.
- Results showing bandwidth requirements for the CHStone benchmarks, which show that for common system configurations, our techniques can allow a user to observe 19x more source code variables.

The paper is organized as follows: Section II provides background information and related work, Section III presents our off-chip tracing architecture, and Section IV provides results and analysis. Section V provides conclusions.

## II. Background

### A. Approaches for Debugging HLS Applications

While debugging HLS circuits can often be done using software debugging tools, or RTL-simulation-based tools [10], certain bugs may be difficult or impossible to resolve using such techniques. System-level bugs, that depend on other components connected to the design, or on input data, may be especially difficult to debug through simulation. For example, if the HLS-produced hardware circuit interacts with other devices with complex data patterns (eg. media streaming, or packet processing), creating a software model that perfectly replicates every possible data pattern may be very time-consuming, or even impossible if the interface is to a legacy or black-box component. In addition, hardware simulation is typically orders of magnitude slower than hardware execution, which may make it impractical if a bug requires long run times.

In such cases it may be necessary to perform debug *in-situ*, that is, in the final operating environment, where the HLS-produced circuit interacts with all other system stimulus. This removes the reliance on user-created test vectors, and allows the system to execute in real time, capturing orders of magnitude longer operation than in simulation. In-system debug has been the focus of several recent works (eg. [1]–[9]), and this paper presents new techniques for this type of debug.

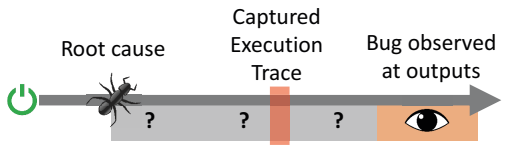


Fig. 2. Locating the source of bugs using captured execution traces, from [11].

### B. Hardware Debug using Signal Tracing

The primary challenge of any type of in-system hardware debug is observability – it is not possible to observe all internal circuit signals during real-time execution. To address this, debugging tools typically store an execution trace, and then allow the user to inspect the recorded values. For example, embedded logic analyzers (ELAs), such as Xilinx's Vivado Integrated Logic Analyzer, or Altera's SignalTap II tool, offer the user the ability to select signals from the circuit they want to observe, which are then recorded into on-chip memories.

FPGAs typically include hundreds of small memories distributed throughout the chip. Using many of these memories, thousands of signals from the user's design can be observed. However, the limited memory capacity means that execution capture is typically limited to thousands of cycles. Designs may execute for seconds, or even hours, before encountering a bug; for a design operating at high speed, this results in an execution trace that is a tiny fraction of total execution.

When using signal-tracing to locate the root cause of a bug, designers typically encounter the challenge shown in Figure 2. When a bug is activated, it may take considerable time before the problem is manifest at the output of the system. The designer must then perform debug iterations, repeatedly selecting different portions of the program to observe while he or she works backwards to determine the root cause of the bug. In such scenarios, being able to capture significantly longer execution traces may reduce the number of debug iterations, saving time in the development process.

By employing large off-chip storage devices, such as DDR memory, execution traces can be increased in size by many orders of magnitude, reducing the time spent debugging [11]. Unfortunately, this option is often not feasible due to the limited memory bandwidth provided by external devices. However, as we show in this work, we can leverage the structure of HLS circuits, combined with static analysis, to enable off-chip recording of execution traces.

### C. Related Work for In-System HLS Debug

There have been several recent works aimed at providing in-system debug of HLS circuits. These works [1]–[9] employ a common technique of recording execution traces in on-chip memory. Once the circuit is halted, these traced are extracted, mapped back to control and data-flow traces in the

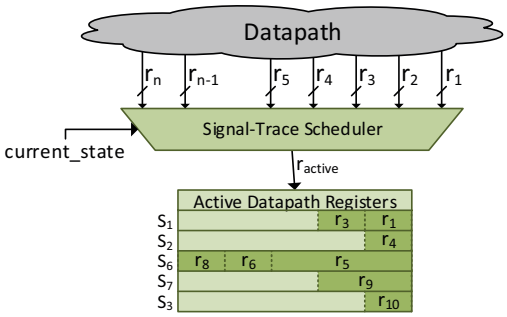


Fig. 3. Dynamic signal-tracing for HLS circuits, from [3].

user's original software program, and provided to the user for inspection and debug purposes.

Of particular application to this work, is the techniques presented in [2], and our work in [3], [4], where we show that the signal-tracing techniques for HLS circuits can be done more efficiently that what is provided by an off-the-shelf ELA tool. Specifically, the HLS schedule can be used to determine the circuit state in which each signal is updated, and logic can be automatically added to feed only the appropriate signals to the trace memory each cycle. Figure 3 illustrates the technique. We showed this technique is 50–150x more memory efficient than the typical approach employed by an ELA. We leverage this technique in our off-chip tracing architecture to reduce the bandwidth of the execution trace.

Also applicable to this work is our techniques in [5], where we analyze HLS kernels to determine if they can share access to an on-chip memory for storing a debug trace. Although similar in concept to this work, the analysis is very basic, and breaks down as the user selects more than a few variables to observe. In contrast, this work provides full bandwidth analysis, with buffering analysis, and supports observing all variables if sufficient bandwidth is available.

### D. Context of This Work

In this work we target functional, in-system debug for a general purpose HLS tool. We assume the HLS circuit is implemented as a set of datapath operations, with controlling state logic to implement the HLS schedule, dictating when each operation in the datapath is performed. Such a structure is common in general C-to-RTL HLS tools, such as LegUp [12] and Vivado HLS, where C code is compiled to LLVM IR, before being scheduled and compiled to RTL. We assume a mapping can be established between circuit signals and source code variables, as well as between control state and lines of source code. This has been shown to be possible for the previously mentioned class of HLS tools [1], [3], [4].

## III. Off-Chip Signal Tracing

In this section we present our proposed techniques for off-chip signal tracing of HLS kernels. We first discuss how such a debugging infrastructure would be used within an HLS design environment, and then present our signal-tracing architecture to support such a debugging flow.

| Record? | Variable | Incr. Cost |
|---|---|---|
| ☑ | kernel_1/main/**i** | - |
| ☐ | kernel_1/main/**j** | 24% |
| ☐ | kernel_1/main/**round** | 5% |
| ☐ | kernel_1/encrypt/**key** | 40% |
| ☑ | kernel_2/main/**in_key** | - |
| ☐ | kernel_2/main/**out_key** | 20% |
| **Bandwidth allocated:** | | **37%** |

Fig. 4. Conceptual user-facing debugging tool, where user is guided in selecting a valid set of variables to observe, constrained

### A. User Experience and Objectives

The primary contribution of this paper is not a user-facing debugging tool, but rather the back-end signal-tracing architecture. However, to motivate the need for such an architecture, and describe the constraints imposed on it, we first describe a conceptual user debugging experience. Like previous work [1], [3], we assume a software-like debug interface, where the captured execution trace from the hardware is mapped back to the original software code, and the user can step through the recorded execution, observing the values of variables.

Although previous work showed that on-chip memory could be used to capture *all* source-code variables for a short duration, off-chip high-capacity storage offers much lower bandwidths. In such a system, a user may be forced to select a subset of source-code variables to observe. For such a debug system to be productive to the designer, we believe it must provide the following:

- *Guarantee no data loss:* The system must determine whether the selected variables can be recorded without data loss.
- *Responsive:* When the user proposes a set of variables, the tool must determine validity quickly (within seconds).
- *Guided:* The tool should be able to provide relative costs of observing variables, such that the user is guided in their decisions. A system where the user selects variables, then asks the tool to validate, without any feedback during the selection process, is likely to be frustrating to use.
- *Fully automated*

Figure 4 shows a conceptual user-facing tool, where users could select which variables to observe, while staying below the 100% bandwidth limit. As variables are selected, the total is updated in real-time. Since the cost to record a variable is not constant, but rather depends on which other variables are selected for recording (see Figure 6), the incremental cost to trace variables must be recalculated each time the user changes which variables are observed.

In the remainder of this section we describe our back-end signal-tracing and program analysis techniques which enable the previously described debugging environment.

### B. Signal-Tracing Architecture

Figure 5 illustrates our proposed architecture for streaming debug traces to off-chip storage devices. An HLS kernel

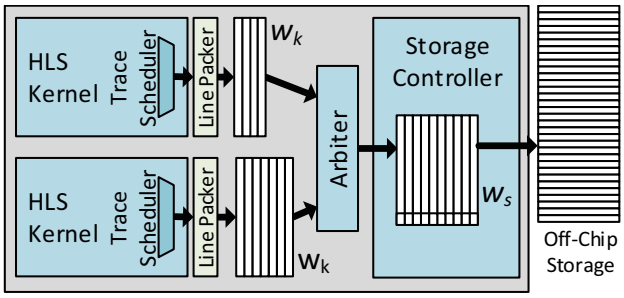


Fig. 5. Off-Chip signal-tracing architecture

generates debugging information, which is a combined control and data flow trace, which is then buffered before being sent to a storage controller via an arbiter.

*a) Storage Controller:* In this paper we do not assume an exact implementation of the storage controller; depending on the resources available in the system, it could be a DDR3/4 memory controller, PCIe controller for an attached solid-state drive, an ethernet controller for a 10G link, or other such high-bandwidth devices. We abstract this controller as a storage device that provides a guaranteed bandwidth ($B_s$), at a given data width ($w_s$). For example, a DDR3 controller may offer a peak data rate of 800Mb/s with 64 data pins. Assuming an FPGA fabric running at 200MHz, the width of the data ($w_s$) would be 256 bits. Assuming data can be streamed to the DDR3 memory with 89% efficiency [13], then the effective bandwidth of the storage ($B_{storage}$) is 45.5Gb/s.

We assume the storage controller has sufficient internal buffering, such that as long as the arbiter does not exceed the bandwidth capabilities of the controller, no data is lost.

*b) Arbiter:* The arbiter is responsible for dividing the bandwidth of the storage controller, $B_s$, between the HLS kernels with bandwidth, $B_i$. Provided that Equation 1 is satisfied, and that the arbiter pops data from each kernel in at least $B_i/B_s$ cycles, no data will be lost.

$$\sum_{i=1}^{k} B_i \leq B_s \tag{1}$$

To keep track of which data belongs to which kernel in the off-chip storage, the arbiter must add a tag with the kernel index. This means that the buffer for each kernel must have slightly less width than the storage controller. For example, with $k = 2$, a 1-bit tag is sufficient, and $w_k = w_s - 1$.

We do not explore arbiter implementations in this paper; however a simple round-robin arbiter would work fine, and would require little resources, but add the additional constraint that the bandwidth be split evenly ($B_i \leq B_s/k$). More complex arbiters could be used for cases where bandwidth is not evenly distributed.

*c) Line Packer:* The data output by the HLS kernels may not fully utilize the buffer entries (see Figures 3 and 6), resulting in wasted space in the buffer lines. To compenstate for this, we optionally add *line packer* modules which consist of logic to assemble partially used buffer lines into fully utilized lines, on the fly. As we discuss in the results, this greatly increases the efficiency for cases where buffer lines generated by the kernels are only partially utilized.

*d) HLS Kernel:* Each HLS kernel outputs a stream of combined control and data flow information. We describe the construction of this data next.

### C. Construction of Kernel Output Signal

In order to extract the desired debugging information, the HLS kernels must be modified to output the required hardware signals. This includes both control-flow values (state of FSMs), as well as data-flow values (datapath signals).

For outputting the data-flow, we employ the techniques of past work [2], [4], [5], and add custom logic to the HLS kernel which dynamically selects only those signals from the datapath that are changed that cycle (see Figure 3). This logic is referred to as the *trace scheduler*, as it determines *when* each value is traced. This approach significantly reduces the data-flow information that needs to be recorded each cycle.

For outputting the control-flow, previous work simply recorded the FSM state each cycle in a separate memory. However, since we are targeting a single storage device, it is necessary to generate a single data stream containing both control and data flow information. To do this, we modify the *trace scheduler* to also include the FSM value at certain states. We now discuss how we decide which states to record, since outputting data in every cycle would require large amounts of bandwidth, especially if the *line packer* module is not used.

*1) Full control-flow capture:* To obtain a full control-flow trace, it is not necessary to record the FSM state every cycle, but rather it is sufficient to record the FSM state once per basic block. During the HLS process, each basic block in the IR code is converted into one or more FSM states, with straight line execution. Figure 6 shows a state graph with 4 basic blocks. Thus, we modify the *trace scheduler* logic to output the basic block index in one state of each basic block.

The basic block index must be the last piece of information output to the buffer during the basic block. The reason being that the execution trace does not contain any metadata to indicate *what* data is stored in the buffer. Thus to decode the values, one must work backwards, starting with the last piece of data written, the basic block index, which then can be used to decode the datapath values in the buffer, working back to the next basic block index, and so on. Figure 6 provides a sample control flow graph with a short execution trace demonstrating this technique. Although this technique provides the user with a complete picture of the control-flow, as we show in the results, outputting data in many states has the potential to significantly increase the bandwidth. The amount of data required to output the basic block index is very small, so if the *line packer* module is used, the impact on bandwidth is much more minimal.

*2) Partial control flow capture:* In cases where the system is bandwidth restricted, and the user is forced to observe only a few variables, it may not be desirable to record the

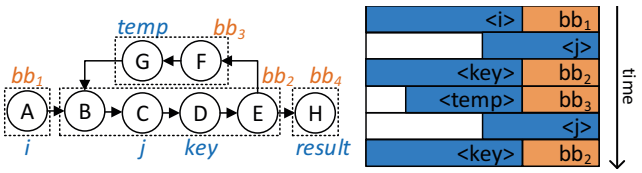


Fig. 6. Sample state graph and execution trace, showing control trace of $\{bb_1, bb_2, bb_3, bb_2\}$. Basic blocks are shown as the dotted boxes, and labeled in orange. Datapath variables are labeled blue, and are adjacent to the state they are updated in.

**function** GETMAXBANDWIDTH($S, n, f$)
  **for all** $s \in S$ **do**
    $traced[0][s] \leftarrow traceWidth(s)$
  **for** $i \leftarrow 1, n-1$ **do**
    **for all** $s \in S$ **do**
      $traced[i][s] \leftarrow max(traced[i-1][ns] \; \forall \; ns \in s.nextStates) + traceWidth(s)$
  **return** $max(traced[n-1])/n * f$

Fig. 7. This algorithm determines the maximum average bandwidth for any path of $n$ cycles. $S$ is the set of states for the kernel, and $f$ is the operating frequency. The *traceWidth(s)* function returns the number of bits traced in state $s$, rounded up to the nearest buffer width, $w$.

complete control flow. Rather, a partial control-flow can be established by only recording the basic block index in blocks where datapath values are traced. This provides the user with a list of events, containing small sequences of control flow with datapath updates; the user will not know the control flow between these events. Although less useful than a complete control flow trace, it may be necessary to use this technique if bandwidth is very limited.

One important consideration is the case where the number of bits that need to be traced in a given state exceed the buffer width, $w_k$. In such a case multiple entries will need to be added to the buffer in the same cycle. If the FIFO structure does not support this behavior, then an alternative is to use multiple parallel FIFOs.

### D. Static Program Analysis for Bandwidth

Determining whether a user-selected set of variables can be observed, without data loss, requires an analysis of the user's design. In this section we present an algorithm which uses static analysis to automatically determine the maximum sustained bandwidth generated by the debug data.

We make the observation that the maximum sustained bandwidth of debug data generated by an HLS program, will be the maximum bandwidth produced by any one cycle in the state graph. Although code located outside of a cycle may produce a large amount of debug data in a short period, this bandwidth will not be sustained, and can be accommodated through additional buffering.

To locate all cycles within the graph, we employ D.B Johnsons' algorithm [14], which returns all unique elementary cycles in a graph. We are then interested in determining the cycle that produces debug data with the maximum bandwidth. Although it would be possible to simply check the bandwidth of each cycle (sum the bits of traced data, divide by cycle length), there may be many unique cycles in the graph, making this computationally expensive. Rather, we employ a dynamic programming solution (Figure 7), which determines the maximum bandwidth for any $n$ cycles, and we set $n$ to be equal to the longest cycle returned by Johnsons algorithm. Although this means Johnson's algorithm is only being used to determine the longest cycle in the graph, we know of no better solution to this problem.

In summary, the algorithm is as follows:

1) Use Johnson's algorithm [14] to get all cycles.
2) Find the cycle with the maximum number of hops, $h$.
3) Determine the bandwidth for the kernel, where $B = getMaxBandwidth(S_i, h, f) \cdot f$.
4) Repeat steps 1–3 for each kernel.
5) Check if Equation 1 is satisfied.

Johnson's algorithm, which has complexity $\mathcal{O}((c+1)(|V|+|E|))$, where $c$ is the number of cycles, may take several seconds to run when $c$ is large (see Section IV-D); however, it only needs to be executed once for the design. Then, to determine the impact on bandwidth of incrementally recording additional variables, $getMaxBandwidth()$ can be run repeatedly, which has complexity $\mathcal{O}(n(|V|+|E|))$, and runs much faster when $c >> n$.

It is important that Step 3 be able to be run quickly, since a debugging tool which validates user selection of variables will need to be re-run each time the user changes which variables are selected for recording.

### E. Buffering Debug Data

Although the previous section determines the maximum theoretical sustained bandwidth, there may be times in the circuit execution that experience a large peak in bandwidth over a short period, requiring sufficient buffering. If the kernel buffer is not sufficiently deep, data will be lost. We perform static analysis to automatically determine the required buffer depths for each kernel.

Our analysis algorithm, $getBufferDepth(S, w_k, r)$ is shown in Figure 8. It operates on the set of states, $S$, and is a function of the buffer width, $w_k$, and the rate per cycle, $r$ that buffer entries are processed and removed from the buffer. The algorithm first initializes the required buffer depth at each state to the number of entries required to trace the data at that state (line 4), and adds the state to a list, $sList$ (line 5). The algorithm then iteratively works through $sList$, popping off states, and determining if any immediate successor states will add to the required buffer depth. This is determined by summing the depth of the current state, plus the number of entries added at the successor state, minus the number of entries removed (line 9). The number of entries removed, $r$ may be a fractional value, representing the fraction of cycles that the arbiter will remove an entry from this kernel's buffer. If the required buffer depth increases at the successor states, they are added to $sList$ (lines 10-12). As long as the

```
1: function GETBUFFERDEPTH(S, w_k, r)
2:   sList ← empty list
3:   for all s ∈ S do
4:     depth[s] ← ⌈traceWidth(s)/w_k⌉
5:     sList.enqueue(s)
6:   while sList do
7:     s ← sList.dequeue()
8:     for all ns ∈ s.nextStates do
9:       newDepth = depth[s]+⌈traceWidth(ns)/w_k⌉−r
10:      if newDepth > depth[ns] then
11:        depth[ns] = newDepth
12:        sList.enqueue(ns)
13:  return ⌈max(depth)⌉
```

Fig. 8. This algorithm determines the required buffer depth to prevent data loss. $S$ is the set of states, $w_k$ is the kernel buffer width, and $r$ is the rate at which the arbiter removes items from the kernel buffer.

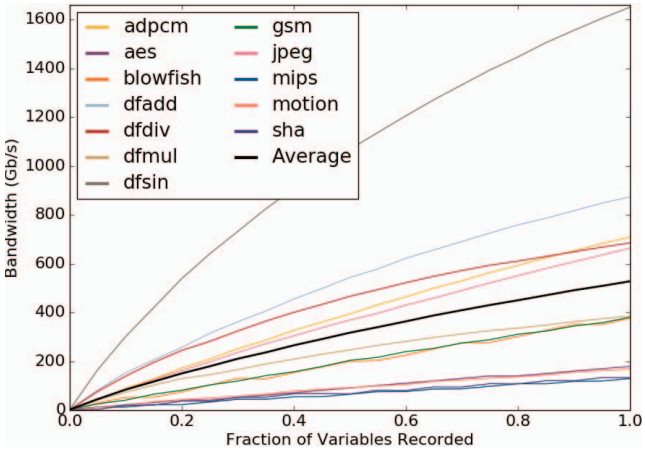


Fig. 9. Bandwidth results using baseline debug trace approach.

bandwidth constraints are validated beforehand, the algorithm will eventually terminate. The return value is the maximum required depth of any state, where the ceiling value is taken to handle cases where $r$ is a fractional value.

In cases where there are multiple HLS kernels, and arbitration is employed, a few extra entries in the buffer may be required to reflect the case where a kernel must wait until being serviced by the arbiter. If using a simple round-robin arbiter, with $k$ kernels, the total buffer depth required for each kernel is an extra $k-1$ entries, making the total buffer depth $getBufferDepth(S_i, w, f)+(k-1)$. More complicated arbitration schemes would require additional analysis to determine this additional factor.

## IV. Experimental Results and Analysis

### A. Bandwidth Results

The primary goal of our architecture, is to provide a framework that maximizes the number of variables a user can observe for a given bandwidth. The experiment in this section demonstrates the substantial reduction in bandwidth that our

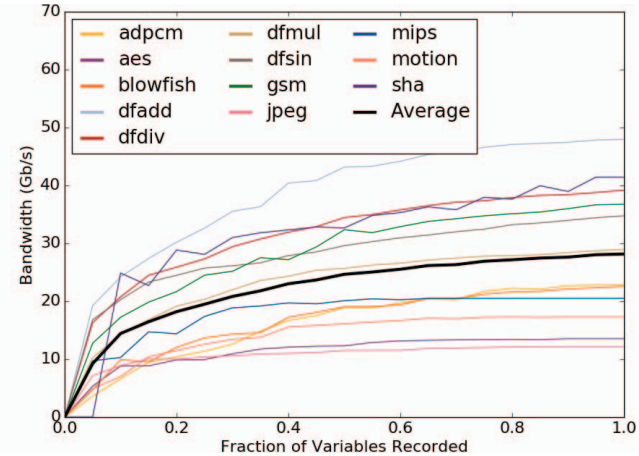


Fig. 10. Bandwidth results using our debug trace architecture.

architecture provides, thus allowing the user to observe many more variables for the same bandwidth constraints.

*a) Baseline:* As a baseline approach, we assume a basic system in which variable values from the HLS kernels are fed directly to a storage controller. Such a system lacks the packet structure and bandwidth analysis we presented in this paper, and thus, assumes a simple approach where all user-selected variables are fed to the storage every cycle. This is analogous to the technique used in ELAs, and requires significant bandwidth. Figure 9 provides a plot of the bandwidth requirements for such a system. For this experiment we used the CHStone benchmark suite [15], compiled using LegUp 4.0 [12], and swept the fraction of source-code variables that are observed. For each data point we performed 100 trials where we randomly selected a set of variables to trace, and calculated the average bandwidth. In this basic approach, recording all variables requires an average bandwidth of 528Gb/s, which is impractical for most systems. For example, if the user was limited to only 50Gb/s (roughly equivalent to DDR3 bandwidth), he or she could only observe an average of 6% of the source code variables.

*b) Our approach:* We repeated the same experiment using the architecture presented in this paper. We assume that only a partial control flow is captured, as described in Section III-C2, and we do not include the *line packer* module. Figure 10 provides a plot of the bandwidth requirements, and shows that the CHStone benchmarks require an average bandwidth of 28Gb/s to capture all variable updates, which is a 19x improvement over the baseline case. This shows that for a bandwidth constrained storage device, our technique would often allow a user to observe an order of magnitude more variables than the baseline approach.

### B. Worst-Case Bandwidth vs. Actual Bandwidth

Next we provide data on how our static analysis, which determines the theoretical worst-case bandwidth, compares to actual bandwidth requirements when the circuit is executed. For this experiment we assume the same parameters as above.

TABLE I
WORST-CASE ANALYSIS VS. ACTUAL BANDWIDTH (GBIT/S)

| Benchmark | 10% P.C.F. | | 10% F.C.F. | | 100% F.C.F. | |
|---|---|---|---|---|---|---|
| | Est. | Act. | Est. | Act. | Est. | Act. |
| adpcm | 6.6 | 3.1 | 30.4 | 12.2 | 30.4 | 13.6 |
| aes | 6.9 | 1.3 | 20.3 | 7.8 | 20.3 | 7.8 |
| blowfish | 9.8 | 2.1 | 43.5 | 22.6 | 43.5 | 22.6 |
| dfadd | 24.8 | 7.0 | 58.3 | 28.5 | 65.1 | 31.5 |
| dfdiv | 20.2 | 2.5 | 45.3 | 8.3 | 50.0 | 10.1 |
| dfmul | 13.9 | 1.9 | 39.8 | 7.1 | 45.7 | 9.1 |
| dfsin | 20.7 | 4.6 | 48.2 | 14.8 | 51.2 | 16.6 |
| gsm | 15.8 | 6.9 | 48.7 | 34.7 | 48.7 | 35.1 |
| jpeg | 9.1 | 4.9 | 32.5 | 19.2 | 32.5 | 19.2 |
| mips | 10.7 | 2.9 | 32.4 | 21.7 | 28.7 | 21.7 |
| motion | 6.7 | 0.1 | 47.5 | 12.4 | 47.5 | 12.4 |
| sha | 24.6 | 2.6 | 43.3 | 25.7 | 49.7 | 25.9 |

Results are reported in Table I for three different configurations: 1) 10% of variables traced, and partial control-flow, 2) 10% of variables traced with full control flow, and 3) 100% of variables traced with full control flow. To obtain actual bandwidth values, we simulated the circuit execution using Modelsim, and reported the largest bandwidth averaged over any period of 1000 cycles.

There are a few important observations to make from this data. First, regarding the closeness of the worst-case to the actual bandwidth. In many cases the worst-case analysis comes fairly close to the actual bandwidth when simulating the circuit; however, in certain cases, especially when few variables are observed, the worst-case significantly over estimates. This likely occurs when the worst-case is found in a loop that is executed for a relatively few number of iterations. Unfortunately this is a limitation of our static analysis approach, as no estimation is made as to how long a specific loop will execute for. More thorough methods of program analysis may provide stricter bounds on the worst case.

Second, as can be seen from the data, enabling full control-flow tracking immediately increases the required bandwidth to levels near 100% observation, even when only 10% of variables are tracked. The reason for this behaviour is that once tracing is added to every basic block in the program, it is guaranteed that tracing will need to be performed in every loop of the design, and the smallest loop will immediately result in a high worst-case bandwidth.

Even though the control-flow data requires very few number of bits, without the *line packer* module, every piece of data that needs to be traced will consume a full line in the buffer.

### C. Full Control Flow and Line Packer

Figure 11 shows the average bandwidth across the CHStone benchmarks for various system configurations. The first configuration, *partial control, no packer* is identical to the experiment from Figure 10. The second configuration shows the impact when full control is enabled, and as seen in Table I, the bandwidth remains high even when tracing few variables.

When the line packer is enabled (Section III-B0c), the small pieces of data needed to trace the control flow will be packed into a single line of the buffer, significantly reducing bandwidth. A more fine-grained (8-bit vs 32-bit) packer provides better results. However, our experiments show the line packer is very expensive in terms of area, requiring 867 Stratix V ALMs for the a 32-bit granularity packer, and 3660 ALMs for an 8-bit packer.

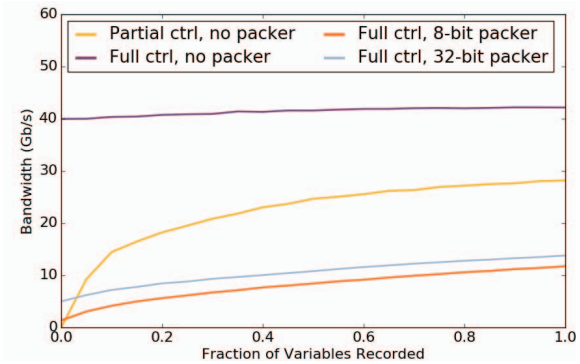


Fig. 11. Effect of line packer on full and partial control flow traces.

TABLE II
RUN-TIME OF STATIC ANALYSIS FUNCTIONS

| Benchmark | Johnsons | getMaxBandwidth() | getBufferDepth() |
|---|---|---|---|
| adpcm | 40 ms | 0.5 ms | 0.7 ms |
| aes | 47 ms | 0.4 ms | 0.7 ms |
| bf | 33 ms | 0.3 ms | 0.5 ms |
| dfadd | 8 ms | 2.1 ms | 0.4 ms |
| dfdiv | 110 ms | 125.6 ms | 0.6 ms |
| dfmul | 5 ms | 2.0 ms | 0.2 ms |
| dfsin | 7550 ms | 675.0 ms | 1.9 ms |
| gsm | 21484 ms | 142.9 ms | 1.1 ms |
| jpeg | 835 ms | 420.5 ms | 3.4 ms |
| mips | 8 ms | 1.3 ms | 0.3 ms |
| motion | 49 ms | 4.2 ms | 0.7 ms |
| sha | 7 ms | 0.7 ms | 0.2 ms |

### D. Run-time

Next we present the run-time results of our static analysis techniques. As described in Section III-D, our bandwidth analysis first consists of running Johnson's algorithm [14] to determine the longest cycle in the design. Johnson's algorithm was executed using the *simple_cycles()* function of the *networkx* Python 3 package. The results show this takes between 7ms and 22s, depending on the number of cycles in the design. However, this part of the analysis needs to only be run once on the design. Then, when the user proposes a set of variables to observe, the $getMaxBandwidth()$ function is run, which takes between 0.3 and 675ms. Since this analysis executes very fast, it meets the requirements of our envisioned user experience from Section III-A; that is, as a user chooses variables to observe, the bandwidth requirements can be validated in real-time. Determining the incremental bandwidth cost of all other variables would take seconds for smaller benchmarks and a couple minutes for larger ones.

We also include the run-time results for the $getBufferDepth()$ function, which would only need to be executed when the user was satisfied with their selection, and decided to synthesize the hardware. This also executes quickly, ranging from 0.2 to 3.4ms.

### E. Area

We do not provide an in-depth area analysis in this paper, as for the most part, the area overhead is similar to overheads discussed in previous works for on-chip execution tracing [3], [5]. The main contributor to area overhead is the *trace scheduler*, which selects which signals to record each cycle and multiplexes them into the debug data stream. This same logic, albeit with a slightly different stream format, was presented in our previous work [3], [5]. We found this scaled linearly with the number of source-code variables that were observed, and reached a maximum of 27–1675 Altera Stratix V ALMs for all variables observed, depending on the CHStone benchmark.

Additional area will be required to implement the arbiter, which will vary based on the complexity of the arbitration scheme, and the number of kernels connected to the arbiter. In [5] we showed that a simple-round robin arbiter required approximately 50 Stratix V ALMs per kernel. Additional area will also be required for the storage controller; however, this is highly dependent on the type of storage medium and implementation of the controller.

### F. Major Limitations and Future work

There are a number of notable limitations with the techniques presented in this paper. Firstly, it requires exclusive access to an off-chip storage controller. However, in some systems, the user may want to share access to this controller between their design and the debug logic. For example, if DDR memory was used as a storage medium, the user's design may also need access to this memory. Sharing access to these resources is beyond the scope of this paper, although we plan to address it in future work.

In this paper we assumed that the user design needed to execute at full speed. However, in certain cases, where system interactions are independent of the execution speed of the user design, it may be possible to slow the user circuit such that bandwidth required for the debug data is also decreased, allowing greater observability.

The other major caveat with the results in this paper is that the bandwidth values presented in Section IV-A are using an academic HLS tool (LegUp) with the default optimization set. A commercial tool, with additional optimizations, may produce an optimized solution with lower latency, higher throughput, or faster clock speeds. Such a solution would likely have larger bandwidth requirements for the same debug data. Thus, for the same bandwidth limits, fewer variables may be observable; however, we believe the architecture and associated analysis techniques presented in this paper will still apply.

## V. Conclusions

In this paper we presented novel techniques to output debug streams from HLS circuits to off-chip storage resources. While past work has focused on recording HLS circuit execution into on-chip memories, limited on-chip FPGA memory results in very short captured execution traces. In this work we presented an off-chip tracing architecture, that when combined with static analysis of the user's design, can allow for off-chip storage of execution traces, within a set bandwidth limit, while guaranteeing no data loss. This increases the length of execution trace that can be captured by several orders of magnitude. We envision that this work will provide for faster debug iterations, and greater productivity from HLS technologies.

## References


[1] N. Calagar, S. Brown, and J. Anderson, "Source-level debugging for FPGA high-level synthesis," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[2] J. S. Monson and B. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–6.

[3] J. Goeders and S. J. Wilton, "Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[4] ——, "Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 127–134.

[5] ——, "Using Round-Robin Tracepoints to Debug Multithreaded HLS Circuits on FPGAs," in *International Conference on Field-Programmable Technology*, Dec. 2015, pp. 40–47.

[6] ——, "Quantifying Observability for In-System Debug of High-Level Synthesis Circuits," in *International Conference on Field Programmable Logic and Applications*, Aug. 2016.

[7] J. S. Monson and B. Hutchings, "Using source-level transformations to improve high-level synthesis debug and validation on fpgas," in *International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 5–8.

[8] ——, "Using Source-to-Source Compilation to Instrument Circuits for Debug with High Level Synthesis," in *International Conference on Field-Programmable Technology*, Dec. 2015.

[9] J. P. Pinilla and S. J. Wilton, "Enhanced Source-Level Instrumentation for FPGA In-System Debug of High-Level Synthesis Designs," in *International Conference on Field-Programmable Technology*, Dec. 2016.

[10] K. Wakabayashi, "CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification," in *International Symposium on VLSI Design, Automation and Test.*, Apr. 2005, pp. 173–176.

[11] Frederic Leens. (Apr. 2016). Debugging FPGAs at full speed, Exostiv Labs, [Online]. Available: http : / / www . exostivlabs . com / logic - observations/ (visited on 06/10/2016).

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, 24:1–24:27, 2013.

[13] Xilinx, "UltraScale Architecture-Based FPGAs Memory IP," Tech. Rep. PG150 v1.3, Nov. 2016.

[14] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975. DOI: 10.1137/0204007.

[15] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.