Synchronizing On-Chip Software and Hardware Traces for HLS-Accelerated Programs

Matthew B Ashcraft
Brigham Young University
Provo, UT
matthew.b.ashcraft@byu.edu

Jeffrey Goeders
Brigham Young University
Provo, UT
jgoeders@byu.edu

***Abstract*—Complex designs generated from modern high-level synthesis tools allow users to take advantage of heterogeneous systems, splitting the execution of programs between conventional processors, and hardware accelerators. While modern HLS tools continue to improve in efficiency and capability, debugging these designs has received relatively minor attention. Fortunately, recent academic work has provided the first means to debug these designs using hardware and software traces. Though these traces allow the user to analyze the flow of execution on both the software and hardware individually, they provide no means of synchronization to determine how operations on one device affect the other.**

We address this challenge by introducing a synchronization technique that keeps track of operations on shared objects. We identify objects shared between hardware and software and their memory operations, and use unique identifiers to synchronize the traces around these operations. We explore the added costs of this technique on execution time and hardware and software resources, and ways to reduce it through multiple synchronization schemes. This is demonstrated in an open-source prototype targeting the hybrid flow of the open-source HLS-tool LegUp.

***Index Terms*—High-level synthesis, HLS, debugging, Synchronization**

I. Introduction

As the use of high-level synthesis tools has increased, they have come to provide support for more complex systems, including heterogeneous systems. In heterogeneous systems part of the program is executed as software on a conventional processor, and the other parts are implemented as hardware accelerators via HLS flows to run on an FPGA. Tools such as Altera OpenCL SDK, Xilinx's SDAccel and SDSoC, and LegUp HLS have come to support this functionality.

Though these tools have greatly simplified the means of generating complex designs for heterogeneous systems, understanding the resulting designs can be quite challenging. The HLS-generated hardware designs for FPGAs are complex and often require hardware experts to understand them. Adding in the software and interface between the two makes it even more challenging. Some tools have been developed to debug these systems using simulation, but those have their own limits. There are situations in which on-chip debugging may be necessary, such as bugs from IO, parallelism, or bugs that take an extended amount of execution time to arise.

One of the commonly used on-chip debugging techniques is trace-based debugging. Trace-based debugging relies on the user recording variables during execution, including those affected by the bug, and working backwards from the recorded data back up to the root cause. Due to limited memory, users have to select variables to record and analyze post-execution. If the root cause is not identified from the recorded variables, the user can select different variables to be recorded and repeat execution again. This process is repeated until the root cause is found. Each iteration the user gains more information on the effects of the bug from the recorded variables, and eventually its origin. Much has been done to improve trace-based debugging for HLS-generated hardware, including giving the user access to more data [1] and more control over what is observed [2]. However, all of these works have focused on debugging the HLS hardware in isolation, ignoring designs for heterogeneous systems.

Our past work extended this software like visibility to complex designs for heterogeneous systems [3], or HLS-accelerated programs. We demonstrated techniques to additionally capture software traces, and present both hardware and software trace data to the user at the source code level. This allows users to now work through hardware and software traces in order to identify and understand bugs. Unfortunately, it does not provide the means to synchronize the hardware and software traces.

This lack of synchronization between the hardware and software traces can prevent the user from understanding the effects hardware and software have on each other. Recording data to the traces from explicit data transfers can give some indication of how the traces line up in execution, but only so often as the data is explicitly transferred. Conversely, objects in shared memory can be accessed by both the hardware and software at any time, possibly without any indication on the opposing device that it has occurred. Without some way of synchronizing the traces, it may not be possible for the user to determine which loads on software are affected by stores on the hardware, and vice versa. If the bug the user is following through execution relies on shared objects, it may be very challenging for them to determine the root cause of their bug.

To address this problem, we have developed a synchronization technique to synchronize the hardware and software traces when performing in-system debug of hybrid HLS systems. Our technique is based around unique identifiers which are shared between the hardware and software, and represent the state

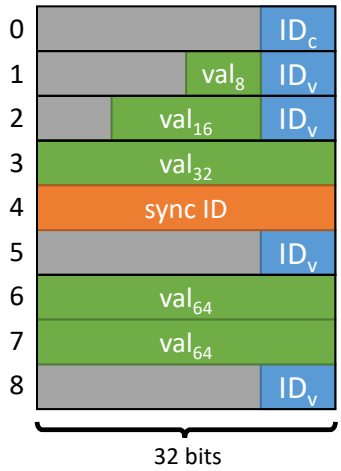

Fig. 1: Software trace array with synchronization ID

of the system at a given point in time. The identifiers are recorded to the hardware or software trace buffers throughout the program, allowing the hardware and software traces to be synchronized at each of these points post-execution. This helps the user gain visibility into the interactions between the hardware and software, and hopefully helps locate the root cause of complex bugs. To test the effectiveness and overhead of our technique, we have developed multiple synchronization schemes based around memory accesses to shared objects and implicit user-inserted synchronization calls. We have explored the costs of these in terms of hardware and software resources, and execution time through a proof-of-concept implemented in the open-source HLS tool LegUp.

II. Background and Related Work

One of the main benefits of using HLS tools to generate complex designs for heterogeneous systems is the ability to debug the designs at a high level. Due to the nature of HLS tools, these designs are often written using a software programming language. This allows users to debug their designs prior to using the HLS tool through software execution and generic software debugging tools. Additionally, tools for hardware and software simulation can be used to identify flaws in the overall design and other bugs that did not arise during software execution [4], [5].

While software execution and hardware and software simulation offer excellent observability for debugging and optimization, there are still certain types of bugs that require on-chip debugging of HLS generated circuits. If the generated circuit interacts with other hardware or data streams (I/O, legacy IP, network traffic, media streams, etc.) it may not be possible to test all possible interactions in simulation. Other bugs are the results of parallelism, which simulation tools are unable to generate. Further, some elusive bugs may require long runtimes to expose, making simulation impractical.

One of the widely used methods of on-chip debugging is trace-based debugging. As mentioned previously, trace-based debugging involves recording specific data to memory during execution, and then analyzing that data post-execution. There are many challenges with trace-based debugging including limited memory, allocating resources on the device to capture data, selecting the variables to record, and analyzing the trace-data to identify the bug. Recent work has addressed some of these trace-based debugging challenges as they pertain to HLS-generated programs, including increased control over selecting variables to record [2], and increasing the amount of data that can be recorded with a given amount of resources [1].

Another challenge particular to HLS-generated programs is understanding the generated design enough to use the trace-data. HLS-generated RTL can be challenging to understand for even small designs, and even more so for large and/or complex designs. Without understanding these designs, the trace-data is not of much use. To address this, recent work has demonstrated techniques to allow trace-based debugging of HLS-generated designs to happen at the source-code level [6]. This is done by maintaining a correlation between the source-code and generated RTL throughout the HLS-flow of the HLS-tool LegUp. Using this correlation, they map the captured trace-data, to the RTL, then back to the source code, allowing users to step through the trace-data as if it pertained to source-code variables. Though all of this work substantially improves the means of trace-based debugging of HLS-generated designs, it only applies to hardware designs, not those of heterogeneous systems.

Techniques for on-chip debugging of designs for heterogeneous systems are comparatively still in their infancy. Xilinx has added the ability to record a timeline of transfers between the software and hardware using their SDSoC tool [7]. Verma et al. [8] demonstrated how FPGA OpenCL code could be modified to add event counters, allowing for a printout of the ordering of different kernel events in the system. Additionally, they added support for user implemented watchpoints, which constantly observe specific addresses and record changes in their values. Though both of these make progress towards observing heterogeneous systems, they don't capture data for both devices that is commonplace in their respective debugging tools, such as control-flow or variable value information.

To overcome this lack of debugging support, our recent work [3] has sought to extend the source-code visibility of HLS designs in LegUp to designs for heterogeneous systems. This recent work built upon features others have added into the HLS tool LegUp [6]. These features automatically insert a circular trace buffer using on-chip memories during the RTL generation portion of the LegUp compiler. In addition to the trace buffer, they identify FSM and datapath signals that correlate to source code variables, and record them into the trace buffer. This allows them to reconstruct data-flow and control-flow of the hardware-accelerated software modules post-execution.

In addition to taking advantage of these features, the our recent work [3] added support for software traces and capturing software execution. The software trace consists of a circular buffer of a user-determined size with entries of values and IDs, similar to Figure 1, but without the *sync ID* (this

will be explained in Section IV-C). The software tracing techniques are based on Instant-Replay [9], in which data from loads and stores are captured as well as unique IDs representing the location and value recorded in each entry. The ID and corresponding value's datatype are recorded to an SQL database. During execution data specified by the user are recorded to the software trace. This data can include control-flow information, loads or stores to specific memory locations, and/or function arguments. Post-execution the software trace is read backwards, ID first, then value. The IDs are used to query the SQL database to determine the data-type of the specified data, allowing for proper data extraction. The results of both the hardware and software traces are represented through the use of a debugger GUI. This allows the user to step through either of the traces to determine what is happening on either machine. While this greatly expands the possibilities of on-chip debugging of CPU/FPGA-based heterogeneous systems, it is lacking a key component, that of synchronization.

A. Debug Scenarios Involving Synchronization

Synchronization is very important in the trace-based debugging cycle for heterogeneous systems when the effects of a bug appear on both hardware and software. If in analyzing the hardware trace the user determines that the bug came from software, it could be extremely difficult to determine when this happened on the software or to know which variables to add to the trace in order to identify the root cause of the bug. Since hybrid HLS systems are still an emerging technology with a relatively small user base it is difficult to find real-world examples of this occurring, but there are a few hypothetical scenarios in which synchronization would be important:

Case 1 The main computation of an algorithm is split between hardware and software, and repeats until an accuracy threshold is met. Though the hardware and software are not executing in parallel, they share large amounts of data through memory accesses. Somewhere during execution, an error in the shared data arises, and its results propagate throughout both devices. Synchronization allows the user to follow the effects of the bug back-and-forth through both devices to its origin.

Case 2 The FPGA is configured as a bump-in-the-wire between the network and the processor, such as in the Microsoft datacenter architecture [10]. An error in the software is traced back to the results from hardware. Through synchronization, the user can determine which hardware operation was directly responsible for the incorrect result, allowing them to further follow the bug back to its origin.

Case 3 A collection of hardware accelerators are regularly fed work through buffers from a controlling software program in a producer-consumer relationship. When a hardware accelerator is fed invalid input, synchronization allows the user to determine which software operations were responsible for the data that should be traced during the next execution in the debug cycle.

Software	Hardware

	`Shared Object X`
	`X = Z`
	`trace[idx++] = X`
	`trace[idx++] = ++SyncID`
	`...`
	`X = ZZ`
	`trace[idx++] = X`
	`trace[idx++] = ++SyncID`

`Shared Object X`
`Y = X`
`trace[i++] = Y`
`trace[i++] = SyncID`	

TABLE I: Synchronization IDs Example

III. Synchronization

Synchronization allows the user to understand how the hardware and software affect each other during execution. For example, synchronized traces could provide profiling types of information, allowing the user to understand that a software loop is consistently delayed by hardware operations on shared objects. Or it could allow the user to see that the result of a load in software was due to a specific store in hardware. In this latter case, synchronization would be essential to debugging an error involving shared objects. As we are focused on debugging heterogeneous systems, the remainder of this paper will focus on synchronization for debugging purposes.

A. When to Synchronize

Under an ideal scenario synchronization could exist for every instruction, allowing the user to step though the hardware or software traces and know what the other device was doing at any point in time. Unfortunately, this is unrealistic. Any means of synchronizing is going to add overhead to the software or hardware or both. This overhead comes in the form of both execution time and resources. For every synchronizing operation on the software, extra instructions must be inserted, resulting in extended execution time. This can also be the case for the hardware depending on available resources. Additionally, any synchronization between hardware and software will require extra logic and memory for storing the synchronization information on both devices.

Our solution to this comes in the form of a synchronization technique and multiple synchronization schemes in which the technique can be applied. Together these provide the means to follow the operations on the shared objects through both traces, allowing the user to identify bugs that originate from or spread through shared objects.

B. Synchronization Technique

Our technique is based on unique identifiers called Synchronization IDs. The IDs are incrementing values that represent points in execution where the system was synchronized, i.e. the traces of both devices aligned and the sequence of operations on shared objects ordered. When a synchronization operation is needed, one of two sequences will follow. When shared memory has been modified, the synchronization ID is incremented, then recorded to the hardware or software traces. This incremented ID represents the agent who last modified the shared object. When shared memory is read, the synchronization ID is recorded to the hardware or software

```
Global Arrays x, y, z
for(i = 1;  i < 100; i++)
  x[i] = x[i-1] * y[i] / z[i];
```

```
Global Arrays x, y, z
for(i = 1; i < 100; i++)
  x1 = load x[i-1]
  y1 = load y[i]
  z1 = load z[i]
  x2 = x1 * y1 / z1
  store x2, x[i]
```

Fig. 2: IR representation of loads and stores

traces. An example of this is shown in Table I, where code is executing on the software and hardware concurrently. When the shared object *X* is modified, the *SyncID* is incremented, then stored. When *X* is read, the *SyncID* is stored. This ID is used post-execution to find the last modifications to the same shared object, allowing the traces to be synchronized. In this example, the *SyncID* recorded in the software will match with a *SyncID* from the hardware, indicating which value of *X* was assigned to *Y*.

C. Synchronization Scheme

The goal of the synchronization schemes is to be able to achieve 100% synchronization, i.e. inducing total access order on shared objects through synchronization, while minimizing the impact on performance. To this end, we propose three synchronization schemes that can achieve 100% synchronization depending on program layout, while minimizing the impact on the program.

1) **Scheme #1 - Memory Instructions:** This scheme is focused on synchronizing each memory instruction on shared objects. Under this scheme, modifying shared objects in hardware or software results in the synchronization ID being incremented, and then stored in its respective trace. Reading shared objects in hardware or software results in the synchronization ID being stored in their respective trace. This technique guarantees 100% synchronization as all reads and writes between shared objects are synchronized.
2) **Scheme #2 - Basic Block of Memory Instructions:** Synchronizing each memory instruction on shared objects, while the most thorough, usually results in higher overhead than needed. In the case where there are multiple memory instructions that access shared objects within a single basic block (small section of code with a single entry and single exit), it might be more efficient to synchronize once per basic block rather than for each memory instruction. An example of this can be seen in Figure 2. The first section of code represents the source code, whereas the second represents the resulting loads and stores of the corresponding representation in the HLS tool. In this example *x*, *y*, and *z* are all shared objects, computing the new values for *x*. Under the previous synchronization scheme this code would result in four different synchronizations, one for each load and store, potentially resulting in excess overhead. If the user knows the hardware is not modifying these values concurrently then they might only need to synchronize once each loop iteration, or even once before and after the loop in order to maintain 100% synchronization. For this scheme, we replace all synchronizations in each basic block with a single synchronization at the end of the basic block. This synchronization acts like a write to a shared object, incrementing then recording the synchronization ID. In the case of this example, the synchronization would occur after the *store* instruction before continuing the loop.
3) **Scheme #3 - Direct Synchronization:** The last synchronization technique is that of user-assisted synchronization. Users who understand their code might have a better understanding of when synchronization is needed. In the case of the example in Figure 2, the user might determine that the synchronization is only needed before and after the loop due to locks, lack of parallelism, or other means. This could greatly reduce the overhead while still providing the means of 100% synchronization. Additionally, direct synchronization could be extended to apply either of the previous techniques to only certain shared objects. Though 100% synchronization would not be maintained for the entire program, as long as the user is able to maintain 100% synchronization on shared objects important to them, that should be sufficient.

IV. Implementation

We implemented our techniques in the open-source HLS tool LegUp. This tool is built within the LLVM compiler infrastructure [11], operating on the intermediate representation (IR) of the code. This IR is an assembly like representation of the code that is independent of the source code language, or the generated target architecture. This allows us to more easily analyze and modify the code within the LegUp tool.

A. Design Flow

Our implementation is based on the hybrid flow of the LegUp tool, taking advantage of previous open-source modifications in [6] and [3]. The original design flow is shown in Figure 3. First the C-based source code is optimized using standard compiler optimizations, and the LLVM IR code is generated. This code is then partitioned according to user specifications into two separate pieces of IR: one for the code remaining on the software, and one for the code to be transformed into hardware logic. From this point on the software and hardware IR are handled separately. The software IR is modified to capture data, and the debugging logic is added to the hardware to capture trace data to a circular trace buffer. The traces are retrieved post-execution, and are parsed

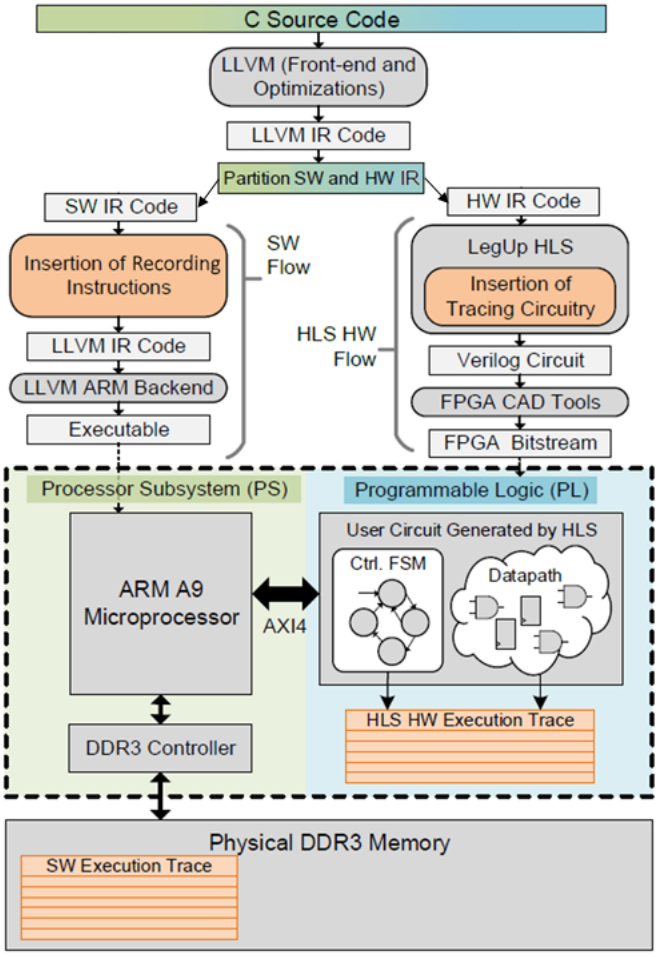

Fig. 3: Design flow for of on-chip debugging of HLS-Accelerated programs, modified from [3]

using data in an SQL database generated during the HLS-flow. These traces are then shown to the user in the form of a debugger GUI.

In order to implement our synchronization technique, modifications are required throughout the HLS-flow. These modifications will be discussed below.

B. Identifying Shared Objects

In order to properly modify the hardware and software code we need to identify the objects shared between both devices. These are objects that are accessed by both hardware and software through memory instructions. To identify these objects, the program is analyzed prior to the partitioning of software and hardware IR, as shown in Figure 3. At that time, each of the global variables is analyzed to determine if there are load or store instructions accessing it from both the hardware and software. If such memory instructions are found, then the global variable and its memory instructions are added to lists of shared objects and instructions. These lists are later used to properly modify the software IR and in the hardware generation process.

C. Software Implementation

Software modifications are necessary to provide support for each of the synchronization schemes. For the first scheme, synchronizing on memory instructions, we insert recording functions similar to those found in [3] for each of the memory instructions accessing shared objects in the software IR. These functions record the synchronization ID, any associated data, and the unique ID representing the location and data being recorded. The layout of these entries is shown in Figure 1. The synchronization ID is managed by a hardware module, and is retrieved through a volatile load of specific hardware address. The hardware address used is different when synchronizing load instructions versus store instructions. The volatile load is used to ensure the synchronization ID is always up to date. As will be noted in Section V, this volatile load, though necessary to retrieve the correct synchronization ID, has a substantial impact on performance.

The second synchronization scheme, synchronizing the basic blocks of memory instructions, works similarly to the first, except all synchronizations in a basic block are replaced with a single synchronization at the end of the basic block. This synchronization only records the unique ID representing its location, and the synchronization ID, which is retrieved from the hardware address corresponding to software writes.

The third synchronization scheme, direct synchronization, relies on direct function calls manually inserted by the user. We identify these calls, and replace them with the same synchronization calls from the second scheme.

D. Hardware Implementation

Unlike software, multiple instruction can execute in parallel during a given hardware state. So instead of synchronizing on a given instruction, the hardware has to synchronize during specific hardware states, similar to how **Scheme #2** on software synchronizes after a set of instructions. Note that this is the only scheme we have implemented for the hardware.

During the generation of RTL from hardware IR code, the LegUp HLS tool maintains correlations between the original source code and the RTL to be generated. We use these correlations and our shared objects list to identify the hardware states in which memory operations on shared objects are occurring. Using this information we generate the synchronization ID module. The hardware modifications required to support this module are represented in Figure 4.

The synchronization ID module determines if the current hardware state (*DUT State*) contains memory operations on shared objects. If so, the module will process the synchronization ID based upon the operation occurring. The states that write to shared objects increment and then store the synchronization ID to the hardware trace. The states that read from shared objects record the synchronization ID to the hardware trace. These checks are done in order, so if writes and reads to shared objects happen in the same state, it will be treated as a write.

When the synchronization ID module determines that an synchronization ID needs to be recorded to the hardware trace,

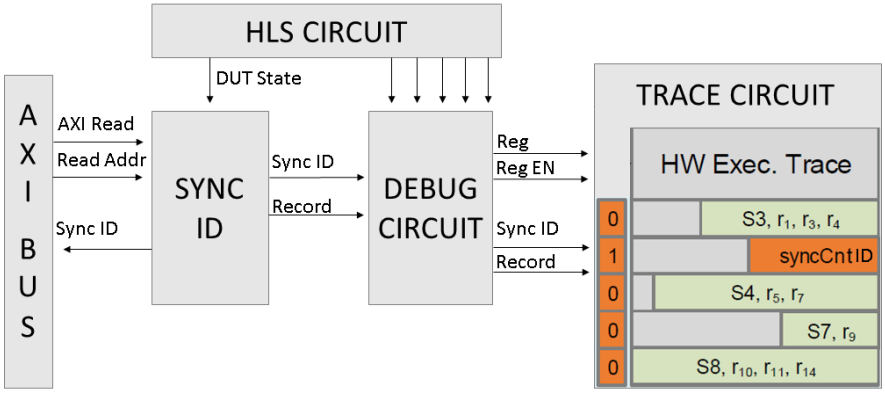

Fig. 4: Hardware Implementation of Synchronization ID Module

	Trace Depth	HW Trace Entries	Buffers Filled	Entries in Trace
backprop Baseline	515	21,552	41.85	2.39%
backprop w/ Sync	515	32,089	62.31	1.60%
srad Baseline	1020	14,910	14.62	6.84%
srad w/ Sync	1020	20,758	20.35	4.91%

TABLE II: Hardware Trace Entries

it sends both the *Sync ID* and a *Record* signal to the debugging circuitry previously added to the LegUp HLS tool [12]. To preserve the original hardware trace data, we have modified this debugging circuity to record both the trace data and the synchronization ID using dual ported memory. The first port is used to record the normal debugging data previously described and implemented in [12]. The other port is only activated when the *Record* signal is set high by the synchronization ID module, at which point the synchronization ID is recorded to the hardware trace.

In order to differentiate between the original hardware trace data and the synchronization IDs we have added a single bit onto the beginning of each trace entry as shown in the Trace Circuit in Figure 4. This bit is set high for entries containing the synchronization ID, and low under all other situations. This allows the user to differentiate between normal trace entries and synchronization IDs post-execution.

The hardware specific addresses accessed by the Software are seen on the AXI bus with a read flag (*AXI Read*) and a specific read address (*Read Addr*). The read address corresponding to software load instructions of shared objects returns the current synchronization ID. The read address corresponding to software store instructions to shared objects results in incrementing and then returns the synchronization ID.

V. Benchmarks and Results

Our synchronization technique impacts program execution in two main ways; execution time and trace entries. To better understand its impact, we have collected execution times for various levels of observation with and without synchronization, and for each of the synchronization schemes. These times have been collected using a high-resolution hardware timer that was added to the designs. Additionally, we have measured the number of trace entries to both hardware and software traces to better understand the impact of our technique and synchronization schemes. We have gathered these results on the Terasic Cyclone V DE1-SoC board.

A. Benchmarks

To test the effect of our synchronization technique and schemes, we have gathered data from two benchmarks from the Rodinia benchmark suite [13], [14]. These benchmarks are Back Propagation (*backprop*) and Speckle Reducing Anisotropic Diffusion (*SRAD*). For testing purposes, we have changed explicit data transfers between hardware and software to global variables. This allows for a more thorough testing of our techniques due to memory operations on shared objects as opposed to explicit data transfers which naturally synchronize the devices.

The *backprop* benchmark is a machine learning benchmark focused on neural networks. It consists of two phases; a forward phase that computes weights, and a backward phase that measures the error and computes new input weights. Each of these phases contain multiple memory operations on shared objects, all within nested loops. Our version of the benchmark computes the forward phase on the CPU, and the backward phase on the FPGA. Additionally, we have sought to gather more data by placing an outer loop around the algorithm that iterates until the measured error reaches a threshold.

The *SRAD* benchmark is a partial differential equation based diffusion algorithm. It is focused on removing speckles in an image without compromising important image features, and is commonly used on ultrasonic and radar images to improve image clarity. Our version of the benchmark executes the main compute of the partial differential equation in hardware, and the remainder in software, both of which are contained within an outer loop. The software portion of the main compute

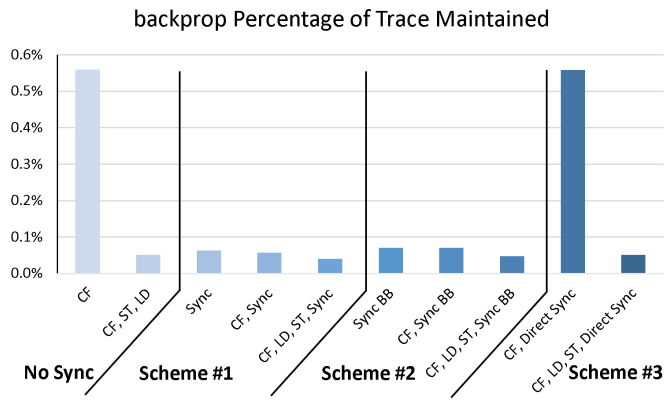

Fig. 5: Percentage of total *backprop* trace entries that fit in the software trace buffer. Higher is better.

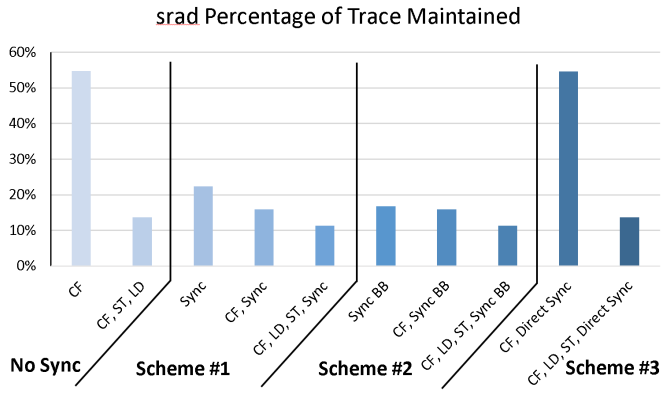

Fig. 6: Percentage of total *srad* trace entries that fit in the software trace buffer. Higher is better.

reads through the image during each iteration of the loop and calculates standard deviations to send to the hardware.

B. Results

One impact of adding synchronization entries to the trace buffers is the reduced availability of entries for traditional debug data. As mentioned previously in Section II, trace-based debugging relies on the user being able to record enough data during execution to determine the cause of the bug or to know which data to record during the next debug iteration. The more entries that are allocated to recording synchronization, the less traditional debug data is available after execution. The measurements in Section V-C and Section V-D are based around this reduction in traditional debug data after execution , or more specifically, the percentage of the overall trace entries that can fit in the trace. For example, if the software trace array could fit 50k entries, and there were 100k entries during execution, then 50% of the total entries could fit in the trace. If our technique were used with **Scheme #1** and added 25k entries during execution then only 50k out of 125k entries, or 40% of the total entries, would fit in the trace at any point in time. The more entries used by synchronization data, the lower the percentage of overall trace entries that fit in the buffer, and the lower the amount of debug data available to the user during this debug iteration.

The impact on execution time is due to the extra memory operations on software for retrieving and recording the synchronization ID. Note that each of the tests achieved 100% synchronization.

C. Impact on Hardware Trace Entries

The hardware trace measurements shown in Table II, demonstrate the effect of synchronization on each benchmark. *Trace Depth* represents the size of the hardware trace buffer in terms of the number of entries it can hold. *Hardware Trace Entries* represents the total number of entries to the hardware trace buffer throughout execution. *Buffers Filled* represents the number of times the buffer wrapped around to its starting address. *Entries in Trace* represents the percentage of total hardware trace entries that can fit in the hardware trace at any point in time.

This data shows the overall reduction in data captured due to the increased number of hardware trace entries from synchronization data. Adding synchronization to *backprop* increases the total number of trace entries by almost 50%, meaning 50% of hardware states that recorded data to the trace buffer accessed a shared object. Due to the extra trace entries, the hardware trace buffer is filled more frequently, reducing the percentage of debug data retained in the trace from 2.39% to 1.60% of all hardware trace entries during execution, a reduction of 33%. Adding synchronization to *srad* increases the number of trace entries by almost 40%. This in turn reduces the percentage of total entries retained in the trace from 6.84% to 4.91%, a reduction of 28%.

D. Impact on Software Trace Entries

The results from the software trace measurements are shown in Figures 5 and 6. For each of our tests 64KB were allocated to the software trace array. Data was collected for each synchronization scheme under various levels of observation including recording control-flow (*CF*), recording loads (*LD*) and recording stores (*ST*). The results are broken up into sections based on the synchronization scheme; **No Sync**, no synchronization [3], **Scheme #1**, synchronizing on memory instructions (*Sync*), **Scheme #2**, synchronizing on basic blocks with memory instructions (*SyncBB*), and **Scheme #3**, user directed synchronization (*Direct Sync*).

Of note is the similarity in graph distribution between the benchmarks even though the percentages of trace maintained are orders of magnitude in difference. This shows that the impact on the trace entries is similar even under vastly different code structures.

Also of note is the absence of user directed synchronization by itself. For these tests, the direct synchronization was placed in between each stage of *backprop* (each phase contained 2 stages), and before and after the hardware computation for *srad*. These locations provided 100% synchronization while

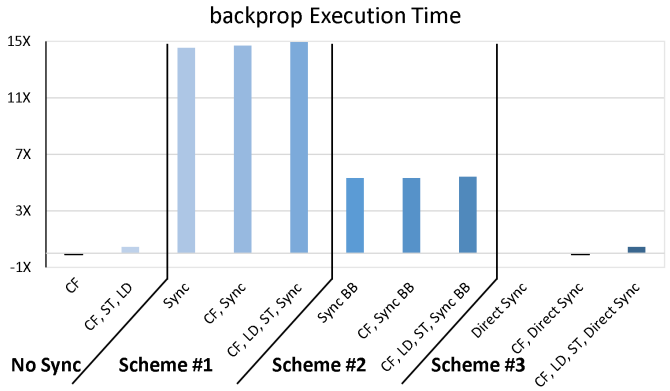

Fig. 7: Increase in *backprop* execution time for various levels of observation. Lower is better.

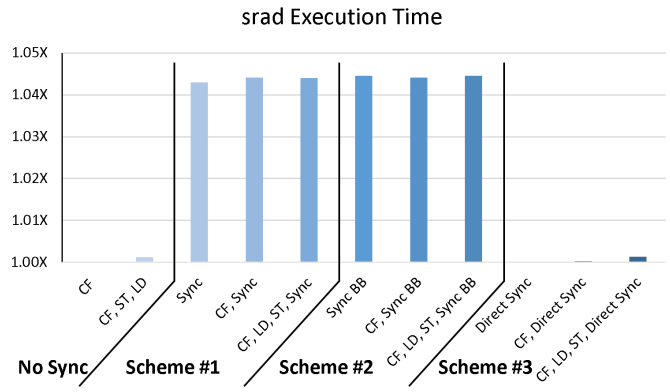

Fig. 8: Increase in *srad* execution time for various levels of observation. Lower is better.

still allowing 100% of the trace entries fit in the software trace array, which skewed the results of the graph.

E. Execution Time

The other effect of adding synchronization is the increase in execution time, mostly due to the use of volatile loads of synchronization IDs by the software. This is shown in Figures 7 and 8. The greatest impact on performance is from synchronizing on memory operations in the *backprop* benchmark with an almost 15X increase in execution time. However, by moving the synchronization to the end of their basic blocks, the increase in execution time drops to just over 5X. Due to the structure of the code, we were able to use direct synchronization to reduce the increase to an almost negligible amount. *srad* saw much smaller increases in execution time due to the smaller amount of synchronizations, with a maximum increase of 0.045x the original execution time. This was not improved by synchronizing on the basic blocks due to the lack of multiple synchronizations in a single basic block. Similar to *backprop*, *srad* saw negligible increases in execution time from user directed synchronization.

These results show the viability of each of the synchronization schemes. For designs such as *srad*, synchronizing on every memory instruction only results in a 1.045X increase in execution time, and guarantees 100% synchronization. For other tests, such as *backprop*, synchronizing at the end of basic blocks with memory instructions will usually still allow for 100% synchronization for a 5X overhead. If this is overhead is too high, then manually inserting synchronization calls may be used if the user has a solid grasp of data-flow on shared-objects throughout the program.

VI. Conclusion and Future Work

This work demonstrated techniques to synchronize hardware and software traces from HLS-accelerated programs. Synchronization between these traces is necessary to understand how hardware and software affect each other, particularly through shared objects. To address this problem, we have put forth a synchronization technique based upon unique identifiers, and multiple synchronization schemes. The technique relies upon the hardware and software recording the identifiers to their respective traces when they access shared objects. The schemes synchronize the software trace at either each memory operation on shared objects, basic blocks that contain those memory operations, or at locations directed by the user.

To demonstrate our proposed synchronization technique and schemes, we have implemented a prototype system based in the open-source HLS-tool LegUp. We modified LegUp to identify the objects shared between hardware and software, as well as memory operations that access them, and use that information to modify the software and hardware according the the synchronization schemes. At each synchronization location on the software, the program retrieves the identifier from the hardware, and records it to the software trace array. A hardware module maintains the synchronization IDs and identifies states in which to synchronize. During these states it records the identifier to the hardware trace using dual-port memory. Additionally, we added a bit to each hardware trace entry to differentiate debug data from the synchronization ID.

To determine the effect of our technique and schemes on the design, we measured the impact on the percentage of total hardware and software trace entries that could fit within the trace buffer at any point in time, as well as the impact on execution time. We discussed situations in which each of the schemes may be most beneficial based upon the results.

Future work in this area could aim to reduce the impact synchronization has on the program execution, as well as expand these techniques to more widely used frameworks.

References

[1] J. Goeders and S. J. E. Wilton, "Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, 2017.

[2] A.-S. Jamal, J. Goeders, and S. J. Wilton, "Architecture Exploration for HLS-Oriented FPGA Debug Overlays," in *International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 209–218.

[3] M. B. Ashcraft and J. Goeders, "Unified on-chip software and hardware debug for hls-accelerated programs," in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec 2018, pp. 354–357.

[4] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *International Conference on Computer Design*, Oct. 2015, pp. 251–258.

[5] L. Yang, S. Gurumani, D. Chen, and K. Rupnow, "AutoSLIDE: Automatic Source-Level Instrumentation and Debugging," in *International Symposium on Field-Programmable Custom Computing Machines*, 2016.
[6] J. Goeders and S. J. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.
[7] J. H. S. Skalicky and V. Kathail, "SDSoC trace: A higher abstraction for system-level profiling, debugging, and tracing of mpsoc systems," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2017.
[8] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-c. Feng, "Developing dynamic profiling and debugging support in opencl for fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17, 2017, pp. 56:1–56:6.
[9] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. 100, no. 4, pp. 471–482, 1987.
[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. A. J. F. M. Haselman, S. H. M. Humphrey, P. K. J.-Y. K. Daniel, L. T. M. K. Ovtcharov, M. P. L. W. S. Lanka, and D. C. D. Burger, "A Cloud-Scale Acceleration Architecture," in *International Symposium on Microarchitecture*, Oct. 2016.
[11] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization*, Mar. 2004.
[12] J. Goeders and S. J. Wilton, "Allowing Software Developers to Debug HLS Hardware," in *International Workshop on FPGAs for Software Programmers*, Aug. 2015.
[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.
[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, Dec 2010, pp. 1–11.