Unified On-Chip Software and Hardware Debug for HLS-Accelerated Programs

Matthew B Ashcraft, Jeffrey Goeders
Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT
matthew.b.ashcraft@byu.edu, jgoeders@byu.edu

***Abstract*—Modern high-level synthesis (HLS)-based tools allow for the creation of complex systems where parts of the user's software are executed on a conventional processor, and the other parts are implemented as hardware accelerators via HLS flows. While modern tools allow designers to construct these systems relatively quickly, observing and debugging the real-time execution of these complex systems remains challenging. Recent academic work has focused on providing designers software-like visibility into the execution of their HLS hardware accelerators; however, this work has assumed that the hardware is observed in isolation. In this work we demonstrate techniques toward a unified in-system software and hardware debugging environment, where the user can capture execution of both the hardware and software domains, and their interactions. We present the performance costs of capturing this execution data, exploring the impact of different levels of observation.**

***Keywords*-High-level synthesis, HLS, debugging**

I. Introduction

As high-level synthesis (HLS) tools continue to see more widespread use, a greater emphasis has been placed on system-level design tools where software can run on a traditional processor, and select portions of code can be automatically accelerated in hardware on the FPGA. Unfortunately, debugging such systems can be very challenging. Typically, most debugging and validation can be performed primarily in the software domain, or by using simulation-backed co-execution. However, not all bugs can be resolved through emulation; data-dependent bugs, non-deterministic behavior (eg. race conditions), or bugs that occur after long run times may make simulation insufficient. When these situations arise, on-chip debug is necessitated, which can be a daunting task for an experienced hardware designer, and near impossible for a software developer.

Recent academic work [1]–[3] has come to provide software-like visibility into the execution of HLS generated hardware. These works capture on-chip execution of HLS designs through traces, then allow the user to view the results of the trace in the context of their original software. While this can greatly simplify the observation of on-chip execution, it is focused on hardware execution in isolation, not taking into account a hybrid hardware-software system.

In this work we demonstrate how a unified in-system software and hardware debugging environment can be implemented to observe and debug hybrid execution (Figure 1). We present techniques to capture execution of both software and hardware domains, as well as their interactions, and present the performance costs of capturing data for various levels of observation the user may select. We demonstrate these techniques in an open-source, proof-of-concept system built on the LegUp HLS tool.

II. Background and Related Work

A. In-System Debug of HLS Designs

A major advantage of using HLS rather than RTL is that typically debugging can be performed much faster using hardware simulation and software execution [4], [5]. While this offers excellent observability, there are still certain types of bugs that require on-chip debugging of the HLS circuit, such as bugs from interacting with other hardware or data streams, or bugs that may require long run times to expose.

Several recent works have aimed to make on-chip HLS debugging easier for designers by correlating source code with captured trace data [1]–[3] as well as improving these techniques to capture more data [6], [7]. However, these works ignore heterogeneous systems.

Some work has been done by Xilinx [8] and Verma et al [9] to capture ordering of events in heterogeneous systems, but this is far from captruing control-flow or variable value information as is commonplace in debugging tools.

B. Debug Scenarios for Hybrid HLS Designs

Here we describe a couple hypothetical scenarios where this technology would be helpful to the designer:

Case 1 The HLS accelerator performs some task where computation is split between software and hardware, such as the partitioning technique discussed in [10]. It may be necessary to observe both domains to determine why the produced result is incorrect.

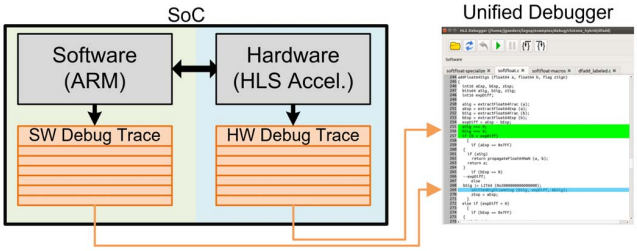

Fig. 1: Overview of unified hybrid HLS debugging.

978-1-7281-0214-6/18/$31.00 ©2018 IEEE
DOI 10.1109/FPT.2018.00072

Case 2 The hardware present on the FPGA is a bump-in-the-wire between the network and the processor, such as in the Microsoft datacenter architecture [11]. A designer may observe incorrect software behavior and wish to observe how the hardware is processing the network data, what is transferred to the software domain, and how that impacts the software execution.

C. Recording Software for Debug

While we know of no other work that captures software execution for debugging hybrid HLS systems, there has been work aimed at recording pure software systems for the purposes of capturing non-deterministic behavior. Instant Replay [12] proposed recording parallel programs, and added version numbers to their reads and writes to correlate recorded data with instructions. This technique has been extended to other situations [13], [14], but these are not conducive to our HLS flows. Our approach will be discussed in Section III-D.

III. Debugging Hybrid Execution

A. Context

We describe our techniques in the context of our prototyped system, in which we implement debugging support within the hybrid HLS flow of the open-source tool, LegUp [15]. The hybrid LegUp flow allows users to partition their source code between software to execute on the ARM processor of an SoC FPGA, and HLS generated accelerators that are implemented in the FPGA fabric. The produced system is a cache-coherent shared memory system, where the HLS accelerators have access to the main memory (DDR) of the ARM processor.

B. Overview of HLS and Debug Flow

Figure 2 illustrates our proposed hybrid HLS flow. The source code is partitioned between the hardware and software, and compiled to the LLVM internal-representation (IR). We have added compiler transformations to the LegUp tool to insert recording instructions into the software destined IR before the ARM executable is generated. This allows us to capture a software trace for post-execution debugging. For the hardware domain, LegUp inserts the tracing circuitry as part of generating the hardware circuit. Upon execution, traces from both the software and hardware record data in the system memory, and FPGA memory, respectively.

After execution, a debugging system communicates with the FPGA to retrieve the execution traces and present the data to the user in a clear fashion, as shown in Figure 1. We are currently testing techniques to synchronize the execution traces, which will be explored in future work.

C. Capturing Hardware Execution

In order to capture execution of the HLS hardware circuit, we make use of the existing open-source tool by Goeders et al. [16], which targets the hardware-only version of the LegUp HLS flow. During generation of the RTL circuit, the tool automatically inserts a trace buffer using on-chip FPGA memories. FSM and datapath signals that correlate to source

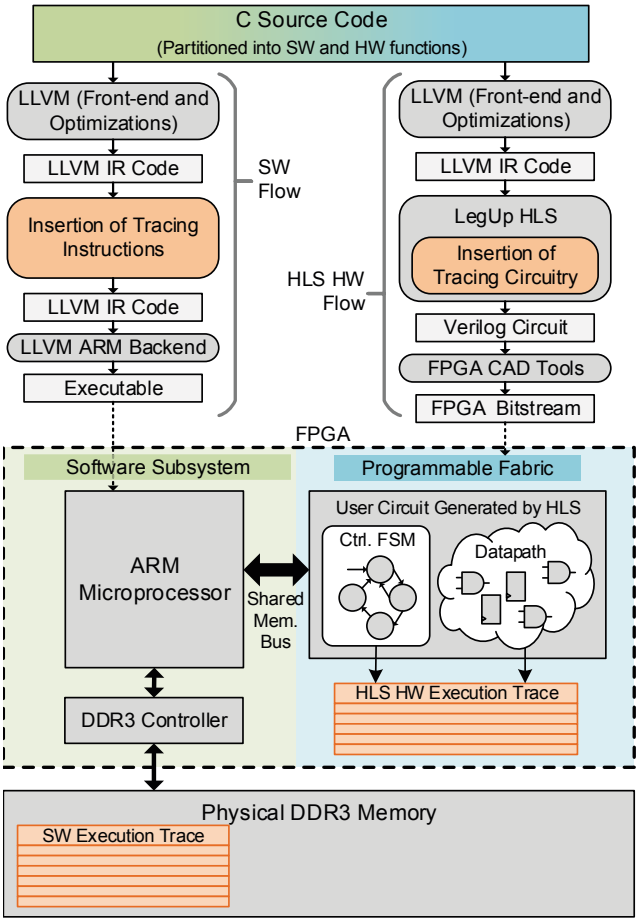

Fig. 2: Design flow for unified on-chip software and hardware debugging for HLS-accelerated programs.

code variables are recorded into a trace buffer, and can be used to reconstruct the control and data-flow of the hardware-accelerated software modules.

D. Capturing Software Execution

Software execution is captured by inserting extra operations into the compiler IR code that record the desired data to a specified location.

Capturing Data to a Trace Array

A globally declared array is automatically inserted to store the software execution trace data, and is used as a circular buffer, constantly overwriting the oldest entry. Each entry consists of data, and an ID field. The ID value represents unique control-flow and variable assignment locations within the program where data was captured.

After execution, the software trace data is extracted from the shared memory, read backwards, ID first, followed by data. The ID field is used to query an SQL database containing the size and layout of the current entry, allowing proper extraction of the data.

Selecting What to Record

```
#record input argument 0 elements 1-9
Void transferToFPGA(int *array, int startIdx, int
    endIdx){ ...}
```

```
for(i = 1; i <= 9; i++)
  recordValue(array[i], 1);
transferToFPGA(array, startIdx, endIdx);
```

Fig. 3: Pseudocode representations of recording directives. Top box shows user-entered directives; bottom box is the resultant generated code.

While in hardware we can tap into signals and feed them into a trace buffer without impacting cycle-by-cycle circuit timing, the same is not true for software. Any recording operations that are inserted into the program will likely impact software performance and slow the program down.

To reduce the impact to program performance, we introduce mechanisms to allow the user to select only certain parts of the software program to record at the source code level. This is accomplished through source-code annotations that can be attached to individual variables or functions. Variable annotations record all loads or stores for a given memory location, while function annotations can record at three levels; program-wide, function wide, or calling locations. Program-wide annotations record all control-flow, loads, or stores in the entire program. Function-wide annotations record the same, but only within a given function. Calling location annotations allow recording each argument individually, as inputs or outputs to the program, and allow the user to capture entire arrays being passed to or from the function. An example of this is shown in Figure 3.

E. Triggering Data Capture

Since the software and hardware trace buffers are of finite capacity, it is essential that the user be given control to trigger when in execution data capture occurs. In past HLS debug work [3], breakpoints located in the source code have been used to indicate when capture should halt. For hybrid debugging, however, it is essential that when a breakpoint is encountered, both domains halt recording into the trace buffers, otherwise the other domain might continue catpure and overwrite data the user cares about.

We provide a memory-mapped register that the software can read to determine whether the hardware has encountered a breakpoint. The software domain always checks this before writing to its buffer. An interrupt-driven system would be possible, but it would increase the system complexity, and may be problematic if the user code already has their own interrupt handlers.

While this allows the hardware to halt the software, we have not currently implemented a technique to do the reverse as it would be dependent on the processor architecture, require some form of hardware breakpoints in the CPU, and possibly OS-level support.

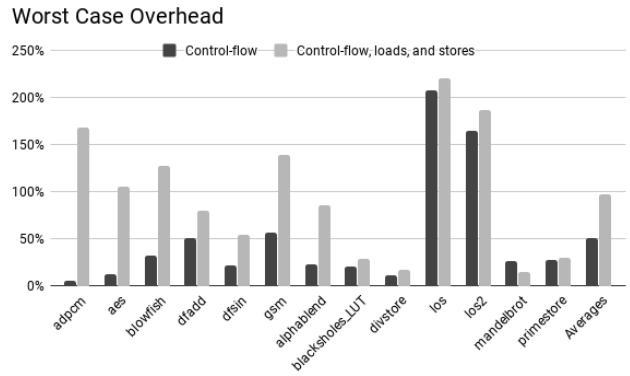

Fig. 4: Program-wide recording of only control-flow, or control-flow, loads, and stores

As a proof-of-concept of our techniques, we have modified the open-source GUI included in [16] to demonstrate a hybrid hardware/software debugging environment. This tool provides a way for a user to step through and inspect hardware and software traces.

IV. Results

The majority of our contributions involved extending hardware-only HLS debug to add software execution tracing and trace synchronization. In this section we measure the impact of our modifications to the software program on its performance.

A. Benchmarks and Methodology

All of our testing was performed on the Terasic Cyclone V DE1-SoC board. Inserting tracing operations into the software results in program slowdown. To understand this overhead we have tested 13 benchmarks, 6 from CHStone [17], and 7 pthread-based benchmarks designed for parallel hardware-software execution in LegUp. Of the pthread benchmarks, five are from [18]; *blackscholes*, *los*, *los2*, *divstore*, and *mandelbrot*. The other pthread benchmarks, *alphablend* and *primestore*, are found in LegUp's benchmarks. *alphablend* alphablends two images and *primestore* computes the amount of prime numbers under a specified limit.

B. Results and Analysis

Our testing was focused on two specific areas; worst case overhead from recording data during software execution, and the cost of recording data transferred between hardware and software. The worse case overhead on software execution was measured by moving almost all of the computation to the software, leaving enough on the hardware to maintain hybrid execution, and running two tests; recording program-wide control-flow only, and recording program-wide control-flow, loads and stores. The results can be seen in Figure 4. Recording data transferred between hardware and software was done by accelerating the default computational work for CHStone, and splitting the computational work between

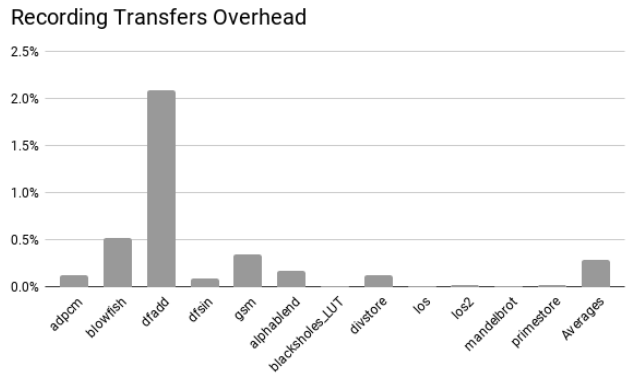

Fig. 5: Overhead of recording data transferred between hardware and software

hardware and software for the pthread benchmarks. Note that the CHStone benchmarks halt execution on the software while the hardware executes, whereas the pthread benchmarks execute concurrently on the hardware and software. The results can be seen in Figure 5.

Under the worse case impact on software execution, shown in Figure 4, the average execution time for recording program-wide control-flow increased by 50%, and varied from 5% on *adpcm* to 207% on *los*. These results were highly dependent upon the amount of control flow in the computational region. The average execution time for recording program-wide control-flow, loads, and stores increased by 97%. These execution times varied from 15% on *mandelbrot* to 220% on *los*. Under normal hardware accelerated conditions, the results from worst case impact would be substantially lessened as much of the control-flow, loads, and stores would have been moved to the hardware, and thus not recorded on software.

The overhead for recording transfers, as shown in Figure 5, was an average increase to execution time of 0.29%. The overhead varied from 0.0024% on *mandelbrot* to 2.09% on *dfadd*. However, *dfadd* was the only test that increased execution time by more than 0.52%, likely due to the short execution time of *dfadd*.

Though recording on software increases hybrid execution on heterogeneous systems, the overhead is highly dependent upon the needs of the user.

V. Conclusion and Future Work

Increasing use of HLS tools and FPGAs have allowed designers and programmers to take advantage of heterogeneous systems relatively quickly, however observing and debugging the real-time execution of these complex systems has remained challenging. We have presented techniques to capture hardware and software trace data, as well as a way to integrate them in an understandable way to the user. By integrating our techniques into the open-source HLS tool LegUp, we have provided a proof-of-concept of a unified software-like debugging environment.

Future work will aim to capture more data with less impact, analyze the need and cost of synchronizing data between devices, and study the impact of these techniques to a greater measure.

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, "Effective FPGA debug for high-level synthesis generated circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[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. 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.

[7] 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.

[8] 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.

[9] 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.

[10] J. Cong, Z. Fang, Y. Hu, and D. Wu, "K-Flow: A Programming and Scheduling Framework to Optimize Dataflow Execution on CPU-FPGA Platforms: (Abstract Only)," in *International Symposium on Field-Programmable Gate Arrays*, Feb. 2018, pp. 287–287.

[11] 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.

[12] 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.

[13] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, Dec. 1989.

[14] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, 2006, pp. 154–163.

[15] 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, pp. 24:1–24:27, 2013.

[16] J. Goeders and S. J. Wilton, "Allowing Software Developers to Debug HLS Hardware," in *International Workshop on FPGAs for Software Programmers*, Aug. 2015.

[17] 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.

[18] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *International Conference on Field-Programmable Technology*, Dec. 2013.