Theses and Dissertations

2020-07-07

Compiler-Based Tools to Aid in Data Transfer Optimization and On-Chip Debug of Heterogeneous Compute Systems

Matthew B. Ashcraft
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Engineering Commons

BYU ScholarsArchive Citation
Ashcraft, Matthew B., "Compiler-Based Tools to Aid in Data Transfer Optimization and On-Chip Debug of Heterogeneous Compute Systems" (2020). Theses and Dissertations. 8613.
https://scholarsarchive.byu.edu/etd/8613

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Compiler-Based Tools to Aid in Data Transfer Optimization and On-Chip Debug of Heterogeneous Compute Systems

Matthew B. Ashcraft

A dissertation submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Jeffrey Goeders, Chair
James Archibald
Brad Hutchings
Mike Wirthlin

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2020 Matthew B. Ashcraft
All Rights Reserved
ABSTRACT

Compiler-Based Tools to Aid in Data Transfer Optimization and On-Chip Debug of Heterogeneous Compute Systems

Matthew B. Ashcraft
Department of Electrical and Computer Engineering, BYU
Doctor of Philosophy

In recent years, many programmers have shifted to using non-traditional compute devices to accelerate their programs. These non-traditional compute devices, such as GPUs and FPGAs, allow users to divide their program into multiple parts, part to execute on the CPU, and part to execute on the non-traditional compute device or accelerator. Though accelerators can provide large performance improvements, they introduce their own challenges, which often require substantial effort on part of the users to overcome. A few of these challenges include scheduling data transfers between the host and accelerator, supporting on-chip debug of heterogeneous systems, and supporting source-code level on-chip debugging for FPGA OpenCL Kernels. This dissertation demonstrates how these challenges can be addressed through compilers, which already play a vital role in analyzing, optimizing, and transforming users code, and proposes that compilers should take on a greater role in simplifying and improving the use of accelerators.

First, we present techniques to efficiently schedule data transfers through compiler analyses. Compared to transferring data immediately before and after the kernel executes, our scheduling results in orders of magnitude improvements in execution time, number of data transfers, and number of bytes transferred.

Second, we demonstrate techniques to provide on-chip debugging for heterogeneous systems through recording execution on the software in addition to debugging circuitry in the hardware, and provide a temporal correlation between the hardware and software traces through synchronization. This allows us to follow debug data between the hardware and software trace buffers. Due to the added cost of synchronizing the trace buffers, we explore synchronization schemes which can reduce the impact synchronization depending on the code structure. We demonstrate the quantitative impact of these techniques on execution time and hardware and software resources, which are under a 2x increase to execution time in most cases.

Third, we demonstrate how source-code debugging techniques for on-chip debugging can be applied to OpenCL FPGA kernels in heterogeneous systems. We developed techniques and a tool-flow that allows users to select variables to record, automatically insert recording instructions into the kernel source code, synthesize the changes directly into the hardware design using commercial HLS tools, retrieve the trace data through kernel arguments, and present it to the user. Overall, quantitative measurements showed our techniques resulted in modest increases to execution time and hardware resources.

Keywords: compilers, accelerators, GPGPU, data transfers, HLS, high-level Synthesis, FPGA
PREFACE

The contributions presented in this dissertation have been published in journals [1], conference proceedings [2], [3], workshop presentations [4], poster competitions [5], and are currently in the review process [6].

The ideas behind the content from Chapter 3 originally came from a workshop presentation [4], after which the techniques were completely redeveloped and optimized and presented in the journal proceedings [1]. The techniques presented in the workshop, as well as future works relating to it, were presented in a poster competition [5]. The content in Chapter 4, was published in conference proceedings [2], [3]. The content in Chapter 5 has been submitted an is under review for a conference [6].

In all of these contributions, I was primarily responsible for developing the techniques, conducting the research, prototyping the techniques, and collecting the measurements. This was done under the guidance Dr. David A. Penry and Dr. Jeffrey Goeders, who also provided editorial support for the publications.


ACKNOWLEDGMENTS

There are many whose contributions and guidance have helped me get to this point in my life. I would like to express my gratitude to Dr. David A. Penry, who got me started on this path, and taught me to love the complexity and possibilities of compilers. He was always supportive and provided me guidance that has stuck with me since.

I would also like to express my gratitude to Dr. Jeffrey Goeders, who helped guide me in a new direction, and provided research ideas that I could pursue to this point. His guidance and collaboration have been essential, and I can say I have thoroughly enjoyed working with him.

Along with my advisors, I would also like to express my gratitude to the faculty in the Electrical and Computer Engineering and Computer Sciences departments, especially those on my committee, who have guided me on my way as I have pursued this long and difficult journey.

Additionally, I must express my gratitude for the project funding that has supported me throughout my Ph.D. research from the Microsoft Safe and Scalable Multicore Computing grant, National Science Foundation Grant CNS-1054075, and the Electrical and Computer Engineering Department at Brigham Young University.

Lastly, I would like to express gratitude to my wife, Joni, who has stuck by me through this whole process, and whose continual support and encouragement played a heavy role in my completion. It has been a large sacrifice for her to take care of me and our family through this long journey, and I will forever be grateful.
# TABLE OF CONTENTS

Title Page ................................................................. i

Abstract ............................................................... ii

Preface ................................................................. iii

Acknowledgements ...................................................... v

Table of Contents ....................................................... viii

List of Tables .......................................................... ix

List of Figures ........................................................ x

**Chapter 1  Introduction** ........................................... 1

1.1 Motivation .......................................................... 1

1.1.1 Need for Heterogeneous Compute Platforms ....................... 1

1.1.2 Challenges of Accelerating Programs ............................... 3

1.1.3 Compiler Support .................................................. 7

1.2 Scope of this Dissertation ........................................... 7

1.3 Challenges and Objectives .......................................... 8

1.3.1 Challenges ....................................................... 8

1.3.2 Objectives and Contributions ................................... 9

1.3.3 Optimizing Accelerator Data Transfers ............................ 10

1.3.4 On-chip hardware and software debug of HLS-accelerated programs ... 11

1.3.5 Source-Code Level Debugging Environment for OpenCL .......... 13

1.4 Organization ....................................................... 14

**Chapter 2  Background and Related Works** ......................... 16

2.1 Support for Heterogeneous Systems ................................ 16

2.1.1 Current Support for GPUs ....................................... 16

2.1.2 Current Support for FPGAs ..................................... 17

2.1.3 Design Challenges ............................................... 19

2.2 Optimizing Data Transfers ......................................... 19

2.2.1 Compiler-Flow .................................................. 20

2.2.2 LLVM ............................................................ 21

2.2.3 Previous Works Related to Optimizing Data Transfers .......... 22

2.3 On-chip Debugging of Heterogeneous HLS Designs ................. 25

2.3.1 Debugging HLS-generated Designs ................................ 26

2.3.2 Trace-Based Debugging ......................................... 27

2.3.3 Previous Work Related to On-Chip Debugging of HLS Circuits ..... 29

2.4 Chapter Summary ................................................... 32
## Chapter 3  Efficiently Scheduling Accelerator Data Transfers  
### 3.1 Objective and Metrics  
### 3.2 Context  
### 3.3 Motivating Example  
### 3.4 Data Transfer Analysis and Scheduling  
#### 3.4.1 Discover Memory Locations Accessed by Kernels  
#### 3.4.2 Discover Conflicting Memory Accesses  
#### 3.4.3 Generate a Scheduling Graph  
#### 3.4.4 Schedule Data Transfers  
#### 3.4.5 Insert Markers  
### 3.5 Experimental Methodology  
#### 3.5.1 Translation from OpenMP  
#### 3.5.2 GPU Code Generation  
#### 3.5.3 Benchmarks  
### 3.6 Experimental Results  
### 3.7 Chapter Summary  

## Chapter 4  On-Chip Debug of HLS-Accelerated Programs  
### 4.1 Objective and Metrics  
### 4.2 Context  
### 4.3 Background  
#### 4.3.1 Debug Scenarios for HLS-Accelerated Programs  
### 4.4 LegUp HLS Tool  
#### 4.4.1 Overview of Unified On-Chip Debugging in the LegUp Hybrid Flow  
### 4.5 Trace-based Debugging of HLS-Accelerated Programs  
#### 4.5.1 Capturing Hardware Execution  
#### 4.5.2 Capturing Software Execution  
#### 4.5.3 Triggering Data Capture  
#### 4.5.4 GUI  
#### 4.5.5 Impact of Trace-Based Debugging of Heterogeneous Systems  
### 4.6 Synchronizing Hardware and Software Traces  
#### 4.6.1 When to Synchronize  
#### 4.6.2 Synchronization Technique  
#### 4.6.3 Synchronization Schemes  
#### 4.6.4 Synchronization Implementation  
#### 4.6.5 Identifying Shared Objects  
#### 4.6.6 Hardware Implementation  
#### 4.6.7 Software Implementation  
#### 4.6.8 Impact of Synchronization  
### 4.7 Chapter Summary  

## Chapter 5  OpenCL Source-Code Level On-chip Debugging Environment  
### 5.1 Objectives and Metrics  
### 5.2 Context  

vii
5.2.1 ROSE ................................................................. 107
5.3 On-Chip Debugging of OpenCL Programs ................................. 108
5.3.1 Step 1: Identifying Variable Assignments ............................... 109
5.3.2 Step 2: Selecting Variable Assignments for Recording ............... 110
5.3.3 Step 3: Modifying the Source Code to Record Selected Variable Assignments ............................... 111
5.3.4 Step 4: Modifying the Host Program, Executing Hardware, and Retrieving Data .................................................. 114
5.3.5 Step 5: Displaying the Trace Data .................................. 115
5.4 Challenges and Optimizations ................................................. 117
5.4.1 Workarounds to Handle OpenCL Files in ROSE ...................... 117
5.4.2 Triggering ................................................................. 117
5.4.3 Trace Buffer Caching ..................................................... 118
5.5 Experiments and Results ...................................................... 119
5.5.1 Benchmarks ............................................................... 120
5.5.2 Results ....................................................................... 120
5.6 Chapter Summary .............................................................. 122

Chapter 6 Conclusion and Future Work ........................................... 124
6.1 Dissertation Summary ......................................................... 124
6.1.1 Summary of Contributions .............................................. 126
6.2 Future Work ................................................................. 132
6.2.1 Efficient Data Transfers .................................................. 132
6.2.2 Unified Debugging Environment for Heterogeneous Systems .... 133
6.2.3 Source-code Level Debugging Environment for FPGA OpenCL kernels .................. 134

References ................................................................. 137
# LIST OF TABLES

1.1 Summary of Contributions ......................................................... 15  
3.1 Number of data transfers per benchmark ......................................... 59  
3.2 Number of bytes transferred per benchmark ..................................... 60  
3.3 Execution times of benchmarks .................................................... 61  
4.1 Effect of optimizing control-flow capture ....................................... 79  
4.2 Hardware Trace Buffer Insertions ................................................. 96  
5.1 Kernel Debug Runtime and Resource Overheads ................................. 121
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Microprocessor Trends [7], [8]</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Compiler Flow</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Trace-Based Debug Flow</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Execution Environment</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Scheduling Graph</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Scheduling Graph before inlining</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Scheduling Graph after inlining</td>
<td>46</td>
</tr>
<tr>
<td>3.5</td>
<td>hotplate pseudo-code</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>Relative number of data transfers compared to baseline</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Relative number of bytes copied compared to baseline</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>Reduction to execution time compared to baseline</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of unified hybrid HLS debugging</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Architectural Overview</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Design flow for unified on-chip debugging of HLS-Accelerated programs</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Recording Instructions</td>
<td>75</td>
</tr>
<tr>
<td>4.5</td>
<td>Software trace array</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Recording control-flow immediately prior to joining branches</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>GUI Screenshot</td>
<td>81</td>
</tr>
<tr>
<td>4.8</td>
<td>Program-wide recording of only control-flow, or control-flow, loads, and stores</td>
<td>84</td>
</tr>
<tr>
<td>4.9</td>
<td>Overhead of recording data transferred between hardware and software</td>
<td>85</td>
</tr>
<tr>
<td>4.10</td>
<td>Shared Objects</td>
<td>86</td>
</tr>
<tr>
<td>4.11</td>
<td>Synchronization Instructions</td>
<td>88</td>
</tr>
<tr>
<td>4.12</td>
<td>Synchronization Instructions with Trace buffers</td>
<td>88</td>
</tr>
<tr>
<td>4.13</td>
<td>IR representation of loads and stores</td>
<td>89</td>
</tr>
<tr>
<td>4.14</td>
<td>Hardware Implementation of Synchronization ID Module</td>
<td>92</td>
</tr>
<tr>
<td>4.15</td>
<td>Percentage of total backprop trace buffer insertions that fit in the software trace buffer, as represented by the Y axis. Higher is better</td>
<td>97</td>
</tr>
<tr>
<td>4.16</td>
<td>Percentage of total srad trace buffer insertions that fit in the software trace buffer. Higher is better.</td>
<td>98</td>
</tr>
<tr>
<td>4.17</td>
<td>Increase in backprop execution time for various levels of observation. Lower is better</td>
<td>99</td>
</tr>
<tr>
<td>4.18</td>
<td>Increase in srad execution time for various levels of observation. Lower is better</td>
<td>100</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution Environment</td>
<td>106</td>
</tr>
<tr>
<td>5.2</td>
<td>AST Example</td>
<td>107</td>
</tr>
<tr>
<td>5.3</td>
<td>OpenCL Source-code Level Debug Design Flow</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>Variable Selection GUI</td>
<td>110</td>
</tr>
<tr>
<td>5.5</td>
<td>Trace Buffer with Entries from Listing 5.4</td>
<td>112</td>
</tr>
<tr>
<td>5.6</td>
<td>OpenCL Debug Trace Viewer</td>
<td>116</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

1.1 Motivation

Since the advent of the computer, designers have sought ways to provide year-over-year improvements for program execution speed and efficiency. Hardware designers have developed new or improved existing architectures for CPUs. High-level languages have been continuously developed and improved to take advantage of the hardware, allow for more complex programs, and reduce the burden of constructing complex programs on users. Compilers have come to support increasingly complex analyses and optimizations, leading to greater program performance. Additionally, they have come to support various source-code languages and targeted architectures. Unfortunately, the year-over-year improvements in program execution speed have decreased in recent years [7], [8]. This has led more designers and researchers to turn to other means to improve program performance. One of these areas is the use of hardware accelerators and non-traditional compute devices, which run in conjunction with general-purpose microprocessors or CPUs.

1.1.1 Need for Heterogeneous Compute Platforms

General-purpose microprocessors are designed to handle all types of instructions. However, by supporting all types of instructions, traditional microprocessors struggle to excel on specialized workloads. This problem has been exacerbated in recent years due to the power wall limiting the performance improvements in Moore’s Law. For many years, performance of general-performance microprocessors doubled roughly every 18 months due to advances in semiconductor manufacturing technologies. This continued until around the year 2003 when the year-over-year improvements began to decrease due to the power wall, or the heat generated from smaller denser circuits [7], [8]. This is shown in Figure 1.1. As a result of the power wall and challenges in scaling to ever smaller process dimensions, microprocessor architectures have shifted to operate on mul-
Figure 1.1: Microprocessor Trends [7], [8]

multiple data in parallel by replicating computation to make up for the slowdown. Unfortunately, that has not kept up with previous trends in year-over-year improvements. This problem is exacerbated by current technology trends such as machine learning, cloud computing, self-driving cars, and others, which require greater computing power. This decrease in year-over-year improvements, and the need for greater computing power, has pushed more developers and researchers to adopt non-traditional compute technologies, to improve program performance.

Two non-traditional compute devices that are increasingly used to improve program performance are Graphics Processing Units (GPU), and Field-programmable Gate Arrays (FPGA) in heterogeneous compute platforms. Each of these non-traditional compute devices provides great advantages under certain scenarios.

GPUs or general-purpose GPUs (GPGPU), consist of many simplified processors executing in parallel. This approach can take advantage of highly parallel workloads consisting of thousands to tens of thousands of operations happening in parallel. These types of computations are beneficial to rendering images, animation, modeling of complex systems, scientific computing, or other
highly parallel algorithms. When executing programs with these code structures, users can see orders of magnitude improvements to execution time [9]. Though large improvements to execution time can be seen, GPU do have their limitations. GPUs execute as part of a heterogeneous system, and require a host computer to initiate the computation, and provide data for the computation. Additionally, GPUs are expensive in terms of power usage, and they are only beneficial on highly parallel workloads. Workloads with extensive control-flow or any data dependencies execute poorly if at all on GPUs. Also, due to their architecture, the performance of each thread on a GPU is quite poor compared to a general-purpose microprocessors. This highly limits the programs and algorithms that can take advantage of GPU architectures.

FPGAs are one of the other non-traditional compute devices that users turn to improve performance. FPGAs take advantage of custom hardware circuits to improve program performance. FPGAs contain reprogrammable logic that can implement arbitrary digital circuits. This allows users to design hardware circuits, load them onto an FPGA, and obtain faster execution and lower power than traditional microprocessors. Additionally, due to their reprogrammable nature, the hardware designs can be improved, changed, or replaced entirely at any time, even after deployment.

Though these non-traditional compute devices can be used in a variety of ways, one of these is to accelerate programs. Accelerated programs are like traditional programs that have been split into multiple parts. Part of the program executes as software on the general-purpose microprocessor, and the other part executes on the hardware accelerator. This allows users to take advantages of the benefits of general-purpose microprocessors and non-traditional computational devices while getting around their disadvantages.

1.1.2 Challenges of Accelerating Programs

This section discusses some of the challenges of accelerating programs with non-traditional compute devices, including those addressed by this dissertation.
GPUs

Though accelerating programs can provide substantial improvements to program performance, they can be difficult to use. Accelerating programs using GPUs requires a parallel algorithm with little control-flow. The user must manually identify the parallel portion of the algorithm, and specify it in the source code through a `pragma` or API calls, such as the CUDA API used for NVIDIA graphics cards. The compiler then splits up the program into two separate programs, one to execute as software, and the other to execute on the GPU, and inserts a kernel invocation call into the software to begin execution on the GPU. Some of the challenges GPUs introduce include identifying parallel code to execute on the GPU, handling data transfers between the GPU and general purpose microprocessor, handling the limited memory on the GPU, and custom tailoring the code to different GPU devices to get the best performance.

Of these challenges, this dissertation will address the handling of data transfers between the GPU and the CPU. Under most scenarios, data needs to be transferred to and/or from the GPU during execution in order to provide the GPU with data for an algorithm, or to retrieve the results of an algorithm. However, data transfers between the GPU and CPU are expensive, so expensive that the performance improvements achieved through using the GPU can be dwarfed by the cost of transferring data [10].

Under situations where the kernel invocation call is not contained within control-flow, and only a single kernel invocation call exists in the program, data can be transferred immediately before or after the kernel invocation call. However, the scheduling of the data transfers becomes more complicated if there are multiple kernel invocation calls that access the same data, or if the kernel invocation calls are contained within control flow. Under these scenarios, efficiently scheduling data transfers requires a thorough understanding of control-flow and data-flow in the program. Though these types of program-wide analyses can be challenging for users to manually perform, compilers are designed with these types of analyses in mind, and should be leveraged to address this challenge.
FPGAs

Support for FPGAs and accelerating programs has greatly improved in recent years. To accelerate programs, the portion of the program executing on the FPGA needs to be translated to a hardware description language, and a means to transfer data between the FPGA and the general-purpose microprocessor must be set up. In the past, this was a particularly large challenge as this required designing a custom circuit using hardware description languages (HDLs) such as Verilog or VHDL. These languages require the user to design circuits at the Register-Transfer Level (RTL), requiring the user to explicitly instantiate logic functions, registers, state machines, memory and memory controllers, and more. Not only is this complex, and time consuming, it requires highly trained experts who are knowledgeable in hardware design. Unfortunately, there are very few people who understand hardware description languages compared to those who understand software coding languages [11]. Fortunately, the development of high-level synthesis (HLS) tools, which automatically generate RTL code from a high-level software language description, have made great strides in allowing software designers to step into this territory, though it is still a work in progress.

Modern HLS tools allow users to specify in the source code which portions of the program they want to execute as software, and which parts are to be generated into hardware designs to be implemented on an FPGA. The HLS tools then generate the software portion into an executable through the regular compiler flow. The portions of the program designated as hardware designs are then translated from software code into a hardware description language. The hardware description language is then run through the standard FPGA vendor tools that synthesize the design into a digital circuit to be implemented on the FPGA. This allows software designers to take advantage of FPGAs to accelerate their programs without having to understand hardware description languages.

Though HLS tools simplify the means of accelerating programs with FPGAs, there are still many challenges to using them. Some of the challenges include the difficulty of describing parallelism in source code, properly specifying optimization directives in the source code, and the difficulty of writing source code the HLS tool will interpret and implement as they desire. Another challenge is debugging the HLS-generated hardware design. Though many of the bugs or performance issues can be found in the source code using source code debugging techniques, or through simulation or hardware emulation, there are still some bugs that may not arise until
the program is executing as software on the general-purpose microprocessor, and hardware as a
hardware design implemented on the FPGA. These types of bugs or performance issues include
bugs dependent on certain I/O data, non-deterministic timings, interactions with other on-chip
modules, or issues that require runtimes too long to simulate to expose.

The particular challenge with bugs or performance issues exposed during program exe-
cution is the need for on-chip debugging techniques. On-chip debugging techniques are already
difficult, as there is no built-in observation tool in FPGAs like you have with a software debug-
ger. To identify the cause of the bug, the user has to rely on embedded logic analyzers to view
changes in a relatively small set of user-selected wires and registers on the device. This problem
is exacerbated by HLS tools, as designers and programmers have to analyze and understand the
HLS-generated hardware design before they can attempt to debug it. Additionally, software pro-
grammers using HLS tools likely have little knowledge of hardware design techniques, making
it extremely difficult for them to analyze and fix them. Fortunately, recent works [12]–[18] have
found ways to correlate the generated hardware designs, and related debug data, with the source
code, allowing user to step through hardware design debug data as though they were stepping
through source code data. These works demonstrate various ways to provide this correlation and
support, however, they are all limited in some ways. They focus on specific HLS tools, hardware
designs executing in isolation, or other techniques. Even though they all have their limitations,
they demonstrate various techniques that can be extended to many different scenarios, and provide
substantial support in this area of research.

Two areas that are not covered by these works are addressed in this dissertation. The first is
support for a source-code level debugging environment for on-chip debugging of HLS-accelerated
programs. Previous source-code level debugging environments for on-chip debugging focused on
hardware in isolation, while this work explores a unified debugging environment for software and
hardware domains on heterogeneous systems. Through this, techniques are demonstrated to allow
users to capture trace data from both the hardware (FPGA) and software (CPU), and present the
data to the user in a unified source-code level debugging environment. Additionally, techniques to
synchronize the trace data from the hardware and software domains are demonstrated. This allows
users to follow trace data across domain bounds when debugging shared objects.
The other area addressed by this dissertation is on-chip, source-code level debugging of OpenCL kernels in HLS-accelerated programs on FPGAs. OpenCL, an extension of C++, is a commonly used source-code language for HLS tools. It is popular because it leverages an existing heterogeneous computing API to specify separation between software and the hardware accelerator, including data transfer and synchronization. The challenge with providing a source-code level debugging environment for on-chip debugging of OpenCL FPGA kernels, is that OpenCL is only supported by commercial tools which are not open-source. To provide the debugging environment that is supportive of all OpenCL HLS tools, we developed techniques based on source-code modifications, rather than needing to change the HLS tool.

1.1.3 Compiler Support

Though accelerating programs for heterogeneous systems introduces many challenges, many of these challenges can be addressed by compilers. Compilers already play an important role helping users adopt to new technologies, languages, and many other developments due to their role in analyzing, optimizing, and transforming code. They take the burden of understanding how different devices and optimizations work away from the user, allowing users to focus on implementing their programs and algorithms as high level code. Through using compilers and compiler tools, researchers have the resources necessary to develop and implement techniques in a place that can lead to wider adoption and support for users. The techniques presented and prototyped in this dissertation are done so through compilers and compiler tools.

1.2 Scope of this Dissertation

As has been noted, and will be further described later in this introduction, this dissertation is focused on providing compiler support for accelerators, particularly GPUs and FPGAs. The scope of this dissertation is rather broad due to the change in research groups part-way through my research. Around the conclusion of my first research publication regarding GPUs, my research group was disbanded, and I joined another research group focused on debugging HLS-generated designs for FPGAs. Fortunately, the foundation for both research groups was using compilers to
address the challenges in accelerating programs using non-traditional compute devices, allowing me to continue in this broad area of research.

1.3 Challenges and Objectives

This section will explain the challenges to using accelerators that are addressed by this dissertation.

1.3.1 Challenges

Though compilers allow for a wide range of techniques to be developed in support of hardware accelerators, this dissertation focuses on three specific challenges:

1. **Efficient data transfers**: Though hardware accelerators can provide substantial improvement to program performance, this improvement can be negated or greatly impacted if data transferred between the hardware and software is not done so efficiently. Transferring data between the hardware and software is expensive and must be handled properly. Transferring data immediately before and after a kernel invocation call is reasonable under certain situations, but if the kernel is deep in the call graph, within nested loops, or if multiple kernels use the same data, then the data may be transferred more than needed. Transferring the data efficiently can require an understanding of control-flow and data-flow in the whole program, which is often beyond the scope of what users are willing to do manually.

2. **On-chip hardware and software debug of HLS-accelerated programs**: Most modern HLS tools support HLS-accelerated programs, where the program is split up into software and hardware parts to run on a general-purpose microprocessor and hardware accelerators. On-chip debugging of these can be difficult due to errors happening in the hardware, software, or in the communication between the two. Works have been done to simplify on-chip debugging of HLS-generated designs through correlating source-code with generated hardware designs [12]–[19], but most of these works have focused on hardware in isolation. These techniques need to be extended to HLS-accelerated programs, where the user can view the hardware and software debug data together. Additionally, techniques are needed to
allow users to traverse domain bounds and follow debug data moving between the hardware and software domains.

3. **Source-code level on-chip debugging for FPGA OpenCL Kernel**: Though techniques have been developed to view and debug trace data from HLS-generated programs at the source code level, most of these techniques are limited, apply to hardware executing in isolation, or do not apply to commercial tools. Unfortunately, support for OpenCL is generally limited to commercial HLS tools. Similar to previous works, on-chip debugging techniques to view trace data in context of the source code should be explored for OpenCL with commercial HLS tools.

### 1.3.2 Objectives and Contributions

The research objectives and contributions of this dissertation are:

- Develop techniques to schedule data transfers between general-purpose microprocessors and GPUs, and measure the reduced impact on program execution. To efficiently schedule data transfers, program-wide analyses are required to identify memory locations used in the kernels, identify aliasing instructions in the host, determine the limits of when data needs to be transferred between the host and the accelerator, and then determine efficient scheduling locations between those limits. The quantitative results should demonstrate the impact the techniques have on execution time and the amount of data transferred between the hardware and software.

- Develop techniques to provide a source-code level debugging environment for trace-based debugging of software and hardware in HLS-accelerated programs, and measure the impact this has on execution time and hardware resources. This involves extending current trace-based debugging techniques for hardware to software, capturing trace data during software execution, then mapping the trace data back to the source code. In addition to capturing software execution, we need a means of synchronizing the hardware and software trace buffers, allowing the user to follow trace data across domain bounds. Measuring the impact of these techniques on execution time and hardware resources will help users determine if the cost
of performing these techniques justifies the extra debug data available through capturing software execution and synchronizing the traces.

- Develop techniques for OpenCL to provide an source-code level debugging environment for on-chip debugging that is applicable to all HLS tools, and measure the impact this has on kernel execution time and hardware resources. This requires determining which data to record during execution, modifying the OpenCL source code to record data to a trace buffer, and retrieving and presenting the trace data to the user post-execution. Modifying the source code will affect the generation of the hardware design due to the additional sequential instructions in the code. The changes to the generated hardware design will affect kernel execution time, and the resources allocated to support the extra instructions that record data to the trace buffer. These changes will be measured to evaluate the cost of supporting these techniques.

The following subsections further introduce these sub-projects.

1.3.3 Optimizing Accelerator Data Transfers

The focus of this dissertation is to develop techniques to improve the usability and performance of accelerated programs using compilers and compiler tools, allowing users to take greater advantage of them with less effort and for greater adoption of non-traditional compute devices. One area that can be difficult for users is scheduling data transfers between hardware and software. Transferring data is essential for most accelerated programs for the user to take advantage of the benefits of each device. Though essential, transferring data is expensive, and can easily dwarf the benefits of using a hardware accelerator if not done properly [10]. For simple programs where the hardware code, or kernel, is called from the main function of the software, and is not contained in any control-flow, scheduling the data transfers is simple. The data can be transferred immediately before and after the kernel invocation. For more complex code structures including call-graphs, and complex control-flow such as nested loops, scheduling data transfers becomes much more difficult. Properly scheduling the data transfers requires an understanding of the control-flow and data-flow in the program in both the hardware and software, as will be demonstrated in Section 3.3. Fortunately, these types of analyses are what compilers excel at.
This dissertation demonstrates the steps and techniques needed to automatically schedule data transfers between hardware and software in efficient locations. These steps include: identifying the memory locations accessed in the kernel to determine which data needs to be transferred, discovering memory operations on aliased locations in the software, generating a scheduling graph used to analyze the program, analyzing the program to determine where the data transfers can be placed, determining efficient locations to schedule the data transfers, and indicating to the hardware code generator where to insert the data transfers.

Chapter 3 puts forth these techniques, how we implemented them into the LLVM compiler, and discusses the impact they have on execution time, number of data transfers, and the number of bytes transferred of the program. The measurements on execution time demonstrate a 1.09X to 53.3X reduction in execution time when using the scheduling techniques. Similar reductions are seen in the number of data transfer invocations and bytes transferred. Though the techniques presented are focused around GPUs, they are applicable to all non-traditional compute devices where the kernel source code is analyzable within the compiler.

1.3.4 On-chip hardware and software debug of HLS-accelerated programs

One of the challenges to adopting new technology is the support available for when things do not work as expected. One of these areas of support is debugging programs. Though users are able to debug the hardware and software source code they send through the HLS tools using source code debugging techniques, there are other bugs that are difficult to identify or even notice before executing the final program on the system. These types of bugs include I/O data, non-deterministic timings, interactions with other on-chip modules, or issues that require extended runtimes to expose. Debugging these types of bugs is difficult, and often requires on-chip debugging techniques, such as trace-based debugging. Though trace-based debugging support for HLS-generated designs has increased, it has generally focused on hardware in isolation, rather than hardware and software executing together in HLS-accelerated programs.

Chapter 4 of this dissertation demonstrates techniques to support trace-based debugging of HLS-accelerated programs through recording traces in both the hardware and software, and synchronizing the traces. This is prototyped through modifying the HLS tool LegUp. When generating HLS-accelerated programs, LegUp uses the LLVM compiler to translate the source code
into an internal representation, then splits the program up into two different programs, one to be
generated into a software program, and one to be generated into a hardware design. From that point
on, the program as a whole is treated as two separate programs. In the portion that is generated
into hardware designs, we take advantage of previous work that provided trace-based debugging
for hardware designs [15].

The software portion of the program must be modified to capture a trace of program execu-
tion according to user specification. To support this we added support for source code annotations
to indicate which data to record. Then we added a software trace buffer, and developed tech-
niques, based on previous works [20], to automatically insert recording functions into the internal
representation. These recording functions record trace data and globally unique identifiers to the
software trace buffer. The globally unique identifiers are used post-execution to correlate the trace
data with the source code data.

In addition to supporting software trace techniques, we need a way of synchronizing the
hardware and software trace buffers, allowing users to follow debug data across domain bounds.
More specifically, we need a way to keep track of operations on shared objects. To address this we
present a synchronization technique based around a global counter representing the most recent
modifications to shared objects that is recorded to the trace buffers when memory operations are
performed on shared objects. We also present multiple synchronization techniques for the soft-
ware code to reduce the impact on software resources and execution time. We demonstrate how
our techniques provide an ordering of operations on shared objects, and the modifications to the
software and hardware design necessary to support it.

To quantify the impact of the software trace and synchronization techniques, we measured
their impact on various benchmarks. Adding the software trace buffer and recording instructions
to the software resulted in an average increase to execution time of 50%. Adding synchronization
to the program affects the amount of trace buffer entries available to debug data, and the execution
time. The hardware trace buffer sees a roughly 33% reduction in the percentage of debug data
maintained in the trace buffer. The software trace buffer sees less of an impact as there are more
resources available to the software. The impact to execution time varied from 1X to 15X increase
to execution time, depending on which data was recorded, and how synchronization was applied.
1.3.5 Source-Code Level Debugging Environment for OpenCL

Previous works have demonstrated how a source-code level debugging environment can be built for HLS-generated designs through modifying specific HLS tools, or adding extra circuitry to the designs, but this can be difficult for certain tools and/or source-code languages, such as OpenCL. Support for OpenCL is generally limited to commercial HLS tools, and though techniques exist to provide some source code level features for on-chip debugging, to our knowledge, a source-code level debugging environment for on-chip debugging of OpenCL kernels has yet to be demonstrated. Chapter 5 of this dissertation puts forth techniques to provide a source-code level debugging environment for on-chip trace-based debugging of FPGA OpenCL kernels. These techniques are based around identifying and selecting data to record, modifying the source-code to record trace data to a buffer during execution, and retrieving, parsing, and presenting that data to the user post-execution. The techniques are broken up into five steps: identifying data to record, selecting data to record, inserting instructions to record data, retrieving the trace data, and presenting the data to the user in a user-friendly format.

The techniques for supporting these steps are demonstrated using Python3, and the source-to-source compiler ROSE. Through using a source-to-source compiler and modifying the source code directly, our techniques are applicable to all HLS tools, including closed-source commercial tools. In addition to the automatic modifications, we discuss the manual modifications to source code we require to support extracting the trace data through a kernel argument. This kernel argument allows the hardware trace buffer to reside in global memory in hardware, and allows the host to extract the trace buffer at the end of execution. Additionally, we demonstrate how a trigger may be inserted to the source code to stop execution, and demonstrate the effects of applying local buffer caching to our techniques.

To quantify the impact of implementing these techniques, we measure the impact to execution time and hardware resources. There is less than a 25% increase to execution time under most scenarios, and less than a 16% increase in hardware resources. Additionally, we show that contrary to our expectations, local buffer caching drastically increases the impact on execution time and hardware resources, and explain possible reasons why.
1.4 Organization

This dissertation is organized as follows. Chapter 2 discusses the background and related works to this dissertation, including the design flow for GPUs and FPGAs, HLS tools, design challenges, compiler-flow, the LLVM compiler, debugging HLS-generated designs, trace-based debugging, and related works. Chapter 3 presents novel techniques to efficiently schedule accelerator data transfers, a motivating example to demonstrate its importance, and the quantitative results from our testing. Chapter 4 presents our techniques for supporting on-chip debugging of HLS-accelerated programs through recording software execution, mapping the results back to the source code, and synchronizing hardware and software traces, as well as our quantitative measurements demonstrating the impact on execution time and hardware and software resources. Chapter 5 presents our techniques to develop an source-code level debugging environment for on-chip debugging of OpenCL FPGA kernels through source code modifications, along with the impact this has on kernel execution time and hardware resources. Chapter 6 concludes this dissertation, and provides a summary of the dissertations, our contributions, and future research directions. Additionally, the contributions, goals and target architecture for Chapters 3, 4, and 5, are summarized in Table 1.1.
<table>
<thead>
<tr>
<th>Contribution</th>
<th>Chapter</th>
<th>Goal and Approach</th>
<th>Target Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Scheduling of Data Transfers in GPU-accelerated systems</td>
<td>Chapter 3</td>
<td>Reduce the frequency and size of data transfers between host and GPU code through automated program analysis and modification.</td>
<td>X86 + GPU system over PCIe, OpenMP benchmarks</td>
</tr>
<tr>
<td>Techniques for Debugging Hybrid HW/SW HLS Designs in SoC architectures</td>
<td>Chapter 4</td>
<td>Extend previous work focusing on debugging HLS hardware to enable observation-based debug shared-memory HLS systems where the design is partitioned into cooperating hardware and software domains. Experiments measure the runtime overhead from recording execution traces of the both the software and hardware domain. In addition, techniques are explored to synchronize the two execution traces, and measure the overhead of these techniques.</td>
<td>Hybrid HW/SW HLS designs executing in a shared-memory system, such as a SoC FPGA. (Demonstrated using the open-source HLS tool, LegUp)</td>
</tr>
<tr>
<td>Techniques to debug FPGA designs created using commercial OpenCL-based HLS tools</td>
<td>Chapter 5</td>
<td>Explore how previous HLS-debug techniques could be applied to OpenCL-based HLS tools, which requires an approach that can be integrated with closed-source commercial tools.</td>
<td>X86 + FPGA system over PCIe, Commercial OpenCL tools and benchmarks.</td>
</tr>
</tbody>
</table>

Table 1.1: Summary of Contributions
CHAPTER 2. BACKGROUND AND RELATED WORKS

This chapter will discuss the background needed throughout this dissertation, as well as the related works. First the techniques and means for developing programs for heterogeneous systems will be discussed, including detailed explanation regarding support for GPUs and FPGAs, and the current limitations (Section 2.1). Next the background and related works for our first contribution regarding scheduling data transfers between the hardware and software will be discussed (Section 2.2). Following that, the background and related works for our other contributions regarding debugging HLS-accelerated programs will be discussed (Section 2.3).

2.1 Support for Heterogeneous Systems

This dissertation revolves around developing techniques to support accelerated programs on non-traditional compute devices. Before discussing the shortcomings and need for improvement, it is important to discuss the current support for these types of systems.

2.1.1 Current Support for GPUs

As mentioned in Section 1.1.1, GPUs consist of many simplified processor cores executing in parallel, and benefit most from high-parallel code with little to no data-dependency or control-flow. To accelerate programs, GPUs are often used in conjunction with CPUs over the PCIe bus, allowing them to take advantage of the speed and throughput of the PCIe lanes.

Generic support of GPUs has become increasingly available due to the development of various source-code language extensions and APIs. The GPU language extensions are similar to high-level source code languages, like C/C++, but with extra libraries. The main GPU language extensions used include OpenCL, OpenMP, CUDA, and AMD APP SDK. OpenCL and OpenMP are extensions to C++ that apply to both AMD and NVIDIA graphics cards, the two main vendors. CUDA applies only to NVIDIA graphics cards. AMD APP SDK only applies to AMD graphics
cards. The languages exclusive to certain types of cards provide greater control over the features available on those devices. In Chapter 3 of this dissertation, we use OpenMP benchmarks, and then use the LLVM compiler to implement our techniques, replacing OpenMP APIs with CUDA APIs, and generating the accelerated program.

To accelerated programs for GPUs, users use the APIs and language extensions to annotate the source code to specify which portion of the program goes on the GPU, as well as handling the interaction between the GPU and general-purpose microprocessor, such as data transfers. The program is then run through a compiler supporting the specific language extension or APIs. The compiler splits the programs up into two parts, part to run as regular software, and part to run on the GPU. The software portion of the code is optimized as normal. However, the GPU portion of the code is treated as a black box, meaning it performs no operations other than what the user has specified through annotations. The program is then compiled and ready to execute. When the program is executed, it connects to the GPU, and executes as the user described. The LLVM compiler we use to generate our accelerated programs will be discussed later in this chapter.

2.1.2 Current Support for FPGAs

The use of FPGAs to accelerate programs has become substantially simpler in the last decade through the support of a wider variety of systems, and the continual improvement of high-level synthesis (HLS) tools. Modern day FPGAs are used to accelerate programs on both SoCs with shared memory systems, and FPGAs connected over PCIe, allowing for greater adoption in a wider variety of scenarios. In addition, the development of HLS tools has allowed for greater adoption of FPGAs as accelerators due as they are now use-able without knowledge of hardware languages.

HLS Tools

It used to be that developing hardware designs for FPGAs required hardware designs to implement circuits in register-transfer level (RTL) languages, such as Verilog and VHDL. RTL languages allow hardware designers to design synchronous and asynchronous hardware circuits at a higher level of abstraction than specifying individual gates and wires. Though hardware designers
can design large and complex circuits, doing so takes time, as they have to explicitly describe cycle-by-cycle logic behavior, registers, state machines, memory and memory controllers, and more. Additionally, in the workforce there are far fewer hardware designers than software developers, hindering wider-spread adoption of FPGAs.

HLS tools have sought to help bridge this gap, allowing users to design algorithms at the software source-code level, and automatically translate them into hardware designs. Additionally, many HLS tools automatically insert the communication logic between the software and hardware designs, greatly reducing the demands on programmers and hardware designers. The most common commercial HLS tools are Vivado HLS for FPGAs from Xilinx, and the Intel HLS compiler for FPGAs from Intel. Additionally, there are many other commercial and academic HLS tools including Symphony C, Catapult, Stratus, and LegUp HLS.

The HLS tools collectively support a variety of source code languages and FPGAs. The source code languages include C, C++, SystemC, OpenCL, Matlab, and Java. Most HLS tools only support one of these languages. Additionally, many of the HLS tools only support specific FPGAs, or FPGAs from a specific company.

Most of these HLS tools are built as part of an existing software compiler. They use the compiler to translate the source code languages into a simpler intermediate representation (IR), use standard compiler optimizations to optimize the IR, and then translate the resultant IR into a hardware design. Some of the HLS tools are open source, and allow users to develop and implement techniques directly into the tool, while others are closed-source, and can only be modified through round-about means.

Through these HLS tools, users can generate programs for HLS-accelerate programs on heterogeneous systems. To do so users write code in high-level languages, specify which portion of the program is to execute on the FPGA through configuration files or source code annotations, and compile an HLS-accelerated program. The HLS tools generates a software executable and hardware design. In addition, the HLS tools handles most of the communication between the software and hardware, as well as the extra IP needed to run on the specified FPGA. With this, the user can copy the software executable and hardware design onto the heterogeneous system, and execute the HLS-accelerated program.
2.1.3 Design Challenges

There are many challenges that arise when using non-traditional compute devices to accelerate programs. These include adapting code to work efficiently on the hardware device, optimizing that code, handling data transfers, correctly setting up the system, debugging issues in the source code and generated code, and more. Much work has been and continues to be done to address many of these challenges, including the work presented in this dissertation.

There are two challenges that this dissertation focuses on, as previously described in the introduction, whose background and related work will be described in the next sections. The first is efficiently scheduling data transfers for accelerated programs, specifically GPUs, as discussed in Chapter 3. Though this work focuses on GPUs, the techniques could be extended to other non-traditional compute devices. Section 2.2 will discuss the background necessary for this work, followed by the related works.

The other challenge this dissertation addresses is debugging HLS-accelerated programs, as discussed in Chapters 4 and 5. These are accelerated programs generated by HLS tools to execute on general-purpose microprocessors and FPGAs. Chapter 4 demonstrates techniques to provide a unified source-code level debugging environment for on-chip debugging of HLS-accelerated programs through capturing trace data from the hardware and software, and synchronizing the trace buffers. Chapter 5 demonstrates techniques to provide on-chip, source-level debugging of commercial OpenCL FPGA kernels through modifying the source-code using source-to-source compilers. As both of these chapters are focused on source-code level debugging environments for trace-based debugging, the necessary background for them, and related works, will be discussed together in Section 2.3.

2.2 Optimizing Data Transfers

This subsection will discuss the background and related works for Chapter 3, the work from my first research group. First the compiler-flow will briefly be discussed to provide a background of how compilers work, and where our analyses and techniques are applied. Following that, we will discuss the Low-Level Virtual Machine (LLVM) [21] compiler, and its internal representation (IR). The LLVM compiler infrastructure is also used in Chapters 3 and 4 to develop and prototype
the techniques. After the discussion on compilers and the LLVM compiler, the related works for Chapter 3 will be discussed.

### 2.2.1 Compiler-Flow

![Compiler Flow Diagram](image)

Figure 2.1: Compiler Flow

Compilers, by definition, translate from one language to another. Traditional compilers translate from high-level source code languages to assembly or binary. Other type of compilers include HLS tools, which translate from source code languages to hardware designs, or source to source compilers which translate from one high-level language to another, like C to Fortran. Each of these compilers is broken up into three main parts: the front-end, middle-end, and back-end, as shown in Figure 2.1.

The compiler front-end is responsible for interpreting the source code languages and performing optimizations. Traditionally, the front-end of the compiler is responsible for interpreting syntax and generating an internal representation (IR) of the source code. The IR varies based on the compiler. For traditional compilers, the IR is often similar to a high-level assembly language, without support for any high-level language features. Generally the IR in traditional compilers is ambiguous towards source code and target architecture. For source-to-source compilers however, the IR tends to be at a higher layer of abstraction, as it needs to maintain the high-level code constructs to help generate the targeted high-level language. All of the analyses, optimizations, and transformations presented in this dissertation are performed on IRs from different compilers.

The middle-end of the compiler is traditionally where complex analyses and optimizations occur. These analyses and optimization vary depending on the compiler, such as dead-code elimination, alias analysis, dependence analysis, loop transformations, or constant propagation. The analyses and optimizations can be focused on a specific loop, function, variable, or even the
entire program. The techniques presented Chapter 3 to efficiently schedule data transfers are developed and implemented in the middle-end of the LLVM compiler. Additionally, the automatic compiler transformations performed in Chapter 5 to identify variable assignment operations, and insert recording instructions into the source code, are implemented in the middle-end of the ROSE compiler.

The back-end of the compiler further optimizes the code for the target architecture, as well as generates the code for specific target architectures. For general-purpose microprocessors, this could include inserting optimized instructions specific for the target architecture, such as vector instructions for x86 processors. For source-to-source compilers, this could include inserting advanced directives for the target source language. This is also the place where HLS generate hardware designs. After translating the source code into an internal representation, the back-end HLS tool translates the IR to a hardware design language, which is later run through a vendor RTL tool that generates the programmable bit-stream. The techniques presented in Chapter 4 are implemented into the LegUp HLS tool, which is built as a back-end of the LLVM compiler.

2.2.2 LLVM

LLVM [21] is a traditional compiler that was designed to fully support custom analyses, optimizations, and transformations at any stage of the compiler flow. It provides support for a large variety of analyses and optimizations throughout the compiler flow. As many of the techniques presented in this dissertation are prototyped in the LLVM IR, it is important to have an understanding of its representation.

The generated LLVM IR is like a high-level assembly language which maintains single static assignment (SSA) form. In SSA form every variable can only be assigned to once. This results in each source code variable being renamed for each variable assignment. Because of this, data-flow analyses in the LLVM IR are much simpler. Additionally, the code is broken up into basic blocks, which are sections of code with a single entry and a single exit.

An example of generated LLVM IR is demonstrated in Listing 2.2, from the source code in Listing 2.1. The Entry basic block from lines 2 - 7 allocates the sum and i variables, initializes
them to 0, and then branches to the LoopPreheader basic block. The LoopPreheader basic block, on lines 9 - 12, checks if i has reached the bounds for the loop (5). If so, it branches over the loop to the Exit basic block, otherwise it branches to the LoopBody basic block. The LoopBody basic block, on lines 14 - 19, loads the values of i and sum, adds them together, stores the result in the sum variable, and then branches to the LoopIncrement basic block. The LoopIncrement basic block, on lines 21 - 25, increments the value of i, then branches back to the LoopPreheader basic block. The Exit basic block, on lines 27 - 30, loads the value of sum, prints it, and then exits the program.

Due to the structure of the LLVM IR, including SSA form, the simplicity of basic blocks, and the lack of high-level language constructs, performing data-flow, control-flow, and other analyses and optimizations is made simpler. Additionally, the LLVM compiler infrastructure is built in a way that easily allows users to develop and implement new analyses and techniques. For this reason the LLVM compiler infrastructure has become a popular compiler in academia and the commercial industry.

2.2.3 Previous Works Related to Optimizing Data Transfers

For a number of years, researchers have sought to reduce the impact of data transfers in accelerated programs through compile-time or runtime analyses and optimizations. Among the first to propose using compiler analyses and optimizations on data transfers between the host and accelerator in accelerated programs, were the works by Yang et al. [22], Vassiliadis et al. [23], and Lee et al. [24], [25].

```c
int main() {
  int sum = 0;
  int i;
  for (i = 0; i < 5; i++)
    sum += i;
  printf("Sum: %d\n", sum);
  return 0;
}
```

Listing 2.1: Source Code Example
The work by Yang et al. focused on a source-to-source compiler framework for CUDA that analyzed memory access patterns and parallelism within the kernel. Using the information gathered, they implemented various optimizations including coalescing memory accesses, vectorization of memory accesses, prefetching data at the beginning of loops, reordering thread workloads to reduce memory conflicts, and thread merging to improve the locality of memory accesses. In addition, they performed a simple data-flow analysis identify read-only data transferred to and
from the hardware. For the variables identified, they removed the transfer from the hardware back to the software.

The work by Vassiliadis et al. focused on using polyhedral analysis to reduce the amount of data transferred between the hardware and software. The polyhedral analysis analyzes memory access patterns in loops, and determines which elements of arrays are accessed in the kernels. Using this, they transfer only the elements needed to the kernel. Sparse workloads benefit most from this type of optimizations as only the array entries containing compute data are transferred.

The work by Lee et al. put forth a compiler framework for optimizing OpenMP on GPUs. Their initial work focused on loop optimizations that improved the locality of data for kernels on the GPU. They later noticed that code optimizations between OpenMP and CUDA often resulted in loops being split up into smaller regions. This resulted in data being transferred back and forth between the hardware and software between each kernel. To address this, they added an interprocedural data-flow analysis to eliminate redundant data transfers between the hardware and software that performed a basic liveness analysis, and a specialized analysis similar to reaching definitions, to determine if data transfers could be eliminated by keeping data on the GPU. Though their techniques reduced data transfers, it only worked when the loops were sequential in the same control-flow, and did not actively seek to improve the placement of the remaining data transfers. This work most closely resembles the techniques presented in Chapter 3. Though independent from their work, the techniques presented in this dissertation are an improvement to their techniques.

In addition to the previous works that execute at compile-time, other techniques were developed to execute at runtime to reduce the impact of data transfers. During runtime, all of the aliasing information in the program is available, allowing for simpler optimizations of the data transfers. Gelado et al. [26] proposed a shared memory system that allowed the CPU to have access to the hardware memories. By doing so, the CPU could use its existing cache coherency protocols to manage memory. Ishizaki et al. [27] use runtime support to track which array entries were modified in the hardware, and only transferred those back to the software. Bourgoin et al. [28], [29] used the OCaml language [30] to develop the SPOC library, which, similar to [27], tracks which array entries are modified in the hardware, only transferring modified data back to the software. Additionally, they perform compile-time analyses to eliminate redundant transfers similar to [24].
Others have turned to hardware support to reduce the impact of data transfers on non-traditional compute devices. Lustig et al. [31] proposed API and hardware changes that allow for data to be transferred as it becomes available. Data is transferred in small chunks based upon the set granularity size as soon as it is available. Kim et al. [32] proposed using page-fault support to notify the software or hardware when data was needed. If the software attempts to access data but it page-faults, then the hardware has modified the data, and it needs to be transferred back to the software. Similarly for when the hardware triggers a page-fault. Though these techniques provide good results, modifying hardware for this support is unlikely.

Though much work has been done to automatically handle data transfers between CPUs and GPUs, these works have focused on eliminating or reducing current data transfers, not efficient placement of data transfers in the code. Through eliminating data transfers, they completely remove certain data transfers from the program. Through other techniques, they are able to reduce the number of bytes transferred in the data transfers. Eliminating and reducing data transfers can be very beneficial, but sometimes without efficiently placing or scheduling all data transfers, the impact of transferring data cannot be reduced. This will be further demonstrated through code examples in Section 3.3.

The remainder of this chapter provides background information for the second and third contributions of the dissertation, which focus on debugging high-level synthesis designs. Since these contributions are substantially different than the first contribution, the proceeding subsections discuss substantially different background material than what has been discussed in this chapter thus far.

2.3 On-chip Debugging of Heterogeneous HLS Designs

Developing techniques for on-chip debugging of heterogeneous HLS-generated designs is challenging because bugs can arise in the software, hardware design, or in the communication between the two. Because of this, the background and related works presented in the section focus on on-chip debugging of software, hardware, and the system as a whole. This will provide the background and related works necessary for Chapters 4 and 5. First techniques for debugging HLS-generated designs, which execute on FPGAs in isolation, will be discussed, including an ex-
planation of trace-based debugging. After discussing trace-based debugging, the related works for Chapters 4 and 5 will be discussed. The related works will discuss trace-based debugging techniques for HLS-generated designs, including works to allow users to view debug data in context of the source code. After this, techniques to record software execution will be discussed, demonstrating techniques available to modify source code to record trace data. Lastly, debugging techniques for HLS-accelerated programs on heterogeneous systems will be mentioned.

2.3.1 Debugging HLS-generated Designs

Due to the nature of HLS-generated designs, which originate as source code, and are translated into hardware designs, there are additional ways to debug them, but also additional challenges they introduce. HLS-generated designs start as a high-level source code, are translated to an RTL language through the HLS tool, then are generated into logic to execute on the hardware device.

At the source code level, users can debug their code using standard software debugging techniques through software emulation. Software emulation compiles the source code of the hardware design to execute on traditional microprocessors. This allows users to test and debug designs without having to synthesize hardware. The main benefit of debugging through emulation is its fast compilation time. Rather than having to go through synthesis and place and routing every time the design is changed, which can take hours or days, the user can quickly synthesize the new design to run through emulation in as little as a few seconds [33]. The main downside to emulation is the extended execution time, especially for bugs that take extended runtimes to expose. Intel’s FPGA OpenCL emulation platform averages a 5x - 10x increase in execution time compared to executing on the FPGA [33]. There are also other limitations to using emulators including lack of parallelism and race conditions, incorrect conditional channel ordering, sharing the same address between the software and hardware [34], and setting up communication with the hardware properly. Generally, many of the bugs relating to the implementation of algorithms are easier to reproduce at this level, and this should be the first place users go to address bugs in HLS-generated designs. However, bugs affected by emulator short-comings, including race conditions and bugs that arise after extended execution times, are difficult to address through emulation.

The next option in debugging HLS-generated designs is to use hardware simulation. In hardware simulation, tools such as Modelsim [35] are used to simulate the RTL level blocks, pro-
viding a more precise overview of the design’s execution on hardware. The translation from source code to RTL through the HLS tool results in substantial changes to the layout of the program. A few lines of source code can result in hundreds or thousands of lines of generated RTL depending on the IP included, significantly altering the program. Performing hardware simulation of the RTL allows users to view the execution of the generated RTL, and have a better perspective of the control-flow and data-flow of the program on the hardware. The main draw-backs to using hardware simulation are the need for users to supply the correct inputs to the simulation to test the design, and the extended execution time. If the correct inputs are not used to test the simulation, then bugs in the code will not be identified. Determining inputs to the design that can test all possible scenarios is usually quite difficult to do. The execution time of a simulated hardware design is several orders of magnitude greater than hardware execution time. This is due to the computation needed to calculate the changes to the entire design every cycle of execution. For example, each run of a logic simulation of NVIDIA’s Fermi GPU on NVIDIA’s proprietary supercomputer took months [36]. Repeating simulation for each bug in a design could be infeasible.

Though software emulation and hardware simulation allow users to identity and debug many types of bugs, there are some situations which require additional debugging techniques. These types of bugs may arise due to non-deterministic timings, interactions with other on-chip modules, I/O, or may require long run-times to expose. To address these bugs, on-chip debugging techniques are often needed, one of which is trace-based debugging.

2.3.2 Trace-Based Debugging

The general idea behind trace-based debugging is to record hardware signal data during execution, and analyze the data post-execution to determine the cause of the bug. The trace-based debugging flow is demonstrated in Figure 2.2. The flow is broken up into six steps: select variables to record, compile, execute the design and record data, retrieve data, present data, and repeat until the bug is identified.

In selecting nets, the user is trying to find nets that show the effects and cause of the bug. Ideally, the user would be able to record data for all nets throughout all execution, however this is unfeasible. The hardware devices have limited memory, so the user has to decide if they want to record more nets or less. If they record many nets, then they will likely only be able to record
each net for a short period of time. If they record a few nets, they will likely be able to record them for a much longer period of time. This period of time the nets are recorded for is called the trace window. Users generally want to record the fewer nets to increase the trace window.

After the nets are selected, the design is compiled, including synthesis, placement, and routing. During this compilation, the trace buffer is inserted into the hardware design along with debugging circuitry to record the selected nets to the trace buffer. The generated design is then run on an FPGA. During execution the selected nets are continuously recorded to a circular buffer. After execution has stopped, either naturally, or due to breakpoints, the trace data is retrieved from the trace buffer. This data is parsed, then presented to the user as a data trace. If the user is able to determine the cause of the bug, then they are finished, otherwise they repeat this process with new nets selected for recording based upon what they learned.

This is a standard on-chip debugging techniques for RTL designs, and is supported by several commercial tools. These tools include Xilinx’s Integrated Logic Analyzer (ILA) [37], and Intel’s Signal Tap [38]. Though these tools provide this functionality, they target RTL designs, not HLS-generated designs. It is still possible to use these tools with HLS-generated designs, but HLS-generated designs provide additional challenges.

With HLS-generated designs this trace-based debugging becomes substantially more difficult because the user is not the author of the generated design. The user wrote source code which was translated into a hardware design. Hardware designs drive signals and wires at a highly parallel level, which is very different from the source code. Because of this, users must spend an extensive amount of time to understand the HLS-generated hardware design before they can focus on debugging it. To address this, work has been done to allow users to view trace data in context.
of the source code, reducing the burden on users to understand the whole circuit. This work will be discussed in Section 2.3.3.

2.3.3 Previous Work Related to On-Chip Debugging of HLS Circuits

This subsection discusses the related works for Chapters 4 and 5, including trace-based debugging of HLS-generated designs, techniques to record software execution, and debugging HLS-accelerated programs.

Trace-based Debugging of HLS-generated Designs

Much work has been done over the years to improve trace-based debugging techniques for HLS-generated designs [39]. Some of the commercial HLS tools, such as those from Xilinx, have come to provide support for traditional trace-based debugging through logic analyzers [40]. Other improvements for trace-based debugging of HLS-generated designs have focused on recording more data with the same amount of resources [41], or providing additional flexibility or improvement over which nets or variables are selected [17], [42], [43]. However, there is an important difference in debugging hardware designs the user has written, and debugging HLS-generated designs. This difference is the user's familiarity with the code.

When debugging RTL code the user has written, the user understands both the flow of the design, and the signals that are being used. For HLS-generated designs, though the user might be familiar with the source code, the code structure, layout, and variables used in the HLS-generated design are likely quite different from their source code. To debug these designs at the RTL level, the users first have to familiarize themselves with the generated design before they can properly debug it. For small designs this can be challenging, but for large designs, it can become almost unfeasible. To address this challenge, several recent works have sought to allow users to debug hardware execution at the source code level [12]–[19].

Monson et al. [12], [19] demonstrated techniques to record trace data when memory or control-flow operations occur using multiple trace buffers. Because each trace buffer they use contains both the cycle the data was recorded in and the value, they are able to merge the traces
together into a unified view post-execution. Additionally, they demonstrated how these technique could be applied automatically at the source code level through source-to-source compilers.

Two of the other works [13], [15] relied on debug data from the front-end of the LLVM compiler to maintain a correlation between the source code and the IR used by the open-source HLS-tool LegUp. From there they went two different ways. Calagar et al. [13] recorded the correlation data in a database, then relied on Altera’s SignalTap II to record trace data during execution. Using the database they were able to present the trace data to the user at a source code level view. In the other work, which will be further discussed in Section 4.4, Goeders et al. [15] modified the LegUp HLS-tool to automatically insert a trace buffer and debugging circuitry into the hardware design. Post-execution they extract the trace data from the hardware and present it to the user in terms of the source code. This is the work the techniques in Chapter 4 are built upon.

The work by Fezzardi et al. [14] extended on these ideas to add formal verification to viewing trace data at the source code level. They propose that most of the time debugging is spent analyzing the trace data, and formal verification can be used to reduce this time. They demonstrate techniques to apply formal verification to the trace results, and emphasize discrepancies to the user when they are analyzing the trace data at the source code level.

In the work by Pinilla et al. [16], they demonstrated techniques to automatically insert debugging circuitry for hardware designs into the source code using the source-to-source compiler ROSE. They use the internal representation of the ROSE compiler to identify control-flow and assignment operations, and insert instructions to record that data to a trace buffer. They then compile the design using Vivado HLS, execute the hardware design, and retrieve the trace data through halting execution and retrieving the trace buffer over a simple serial communication controller. Though their work was focused on hardware only designs, we extended this idea to develop an on-chip source code level debugging environment for OpenCL FPGA kernels in Chapter 5.

The works by Jamal et al. [17], [18] focus on hardware overlays that allow users to rapidly switch between selected hardware signals between debug iterations. During the generation of the hardware designs, they automatically insert debugging circuitry to record all of the variables in the program, or in specific functions. Then as the user selects new variables to record in different iterations, only the states with the selected variables have debug data recorded. Additionally, they show how debug data from multiple states can be combined to save space in the limited memories.
They also demonstrate techniques to halt the recording of trace data without halting execution. To assist users in selecting variables to record, they introduce a dependency analysis that helps users determine the variables that affect others.

Though these works provide various techniques to support trace-based debugging of HLS-generated designs, they either focus on hardware executing in isolation, or they require the modification of HLS tools. This dissertation will present techniques to support trace-based debugging for HLS-accelerated programs on heterogeneous systems, including OpenCL, which is only supported by close-source commercial HLS tools.

**Software Recording Techniques**

Over the years techniques have been developed to record data during software execution, similar to a hardware trace buffer, often for purposes of capturing non-deterministic behavior. Among the first works to put forth these techniques was LeBlanc et al. [20]. They proposed that as long as all of the inputs to the program were recorded, then the program execution could be reconstructed post-execution. They replaced all of the read and write operations with custom made operations. These custom operations, performed the original read or write, record the result of the read or write, and recorded version numbers representing the order in which the operations occurred. Using the recorded data and version numbers they could reconstruct program execution even for non-deterministic programs.

Following the work by LeBlanc et al., various techniques were developed to apply their work to other scenarios. The R2 [44] and Jockey [45] libraries incorporated recording operations into regular library calls. These recording operations apply the techniques by LeBlanc et al. to inputs and outputs of standard library functions. McDowell et al. [46] took a different approach, and proposed recording software execution using tracepoints, modifying the operating system to record information in system calls, and capturing events to reconstruct event histories in the system. Others have proposed modifying processor architecture to record software execution, but this is beyond the scope of this dissertation [47], [48].

The most robust approach we have seen comes from Microsoft, where they use binary translation to change the program at runtime to record data [49], allowing them to selectively record every assembly instruction in the whole program. Their runtime takes control of a program,
similar to a debugger, and performs binary translation, replacing single complex assembly instructions with many simpler ones. This allows them to inject recording instructions in the middle of previously complex operations. Additionally, users are able to select different data to record each iteration, such as reads, writes, control-flow, events, or even the results of every instruction. Their overhead was approximately a 12 - 17x increase to execution time for CPU intensive applications.

Many of these techniques were inspiration for the techniques put forth in Chapter 4 and 5.

**Hybrid Debug of Software and HLS Hardware**

Though progress has been made in providing support for on-chip HLS debug, most of it has focused on hardware in isolation. The work that has involved hardware and software in HLS-accelerated programs has been fairly limited. Xilinx has added support for recording a timeline of data transfers between the hardware and software in their SDSoC tool [50]. Verma et al. [51] demonstrated how OpenCL code for FPGAs could be modified to add event counters, allowing users to record the ordering of events between the different kernels in the system. Though these make progress towards observing heterogeneous systems, they only provide a few of the many features commonly available in hardware and software debugging environments.

### 2.4 Chapter Summary

This chapter presented the information needed to understand the techniques presented in this dissertation. First, the generation of accelerated programs was discussed, for both GPUs and FPGAs, followed by some of the challenges in designing and using them. Following the discussion on design challenges, the background and related work for Chapter 3 were presented. This included an explanation on how compilers are broken up into three stages, the front-end, middle-end, and back-end, and the function of each of those stages. Next the LLVM compiler was introduced, which is used in Chapters 3 and 4, as well as an explanation of the IR which the techniques are developed for. The related works were then discussed, including techniques that have been developed to reduce the cost of transferring data, including memory management, hardware modifications, and polyhedral analyses.
Next the background and related works for Chapters 4 and 5 are discussed. First an overview of debugging techniques for HLS-generated designs was discussed, followed by a detailed overview of trace-based debugging. Next the related works were discussed for trace-based debugging of HLS-generated designs, techniques to record software execution, and debugging techniques for HLS-accelerated programs.
CHAPTER 3. EFFICIENTLY SCHEDULING ACCELERATOR DATA TRANSFERS

There are many challenges in accelerating programs using non-traditional compute devices to improve program performance, one of which is transferring data back and forth between the host and accelerator devices. It is not the act of transferring data back and forth between devices that is necessarily challenging, but rather determining when is the best time in program execution to perform these data transfers. For heterogeneous systems with a CPU and GPU, connected over PCIe, often the software executes on the host until the location of the original kernel code, where upon it signals the kernel to begin executing, and then waits for the kernel to finish executing on the GPU. After the kernel finishes executing, the software resume execution. A novice approach to scheduling or placing data transfers in this type of environment would be to place the data transfer immediately before and after the hardware kernel is invoked and completed. This can be improved in many situations, but requires data and control-flow analyses of the program to determine efficient placement of data transfers.

Under various code structures, moving these data transfers can have a drastic impact on program performance, possibly negating any benefits gained from using the accelerator. Recent work [10] measured the overhead of transferring different amounts of data between a CPU and GPU, and showed that millisecond overheads can be seen from data transfers as small as a few megabytes. In programs with multiple hardware kernels, or kernels executing iteratively in a loop, this overhead is repeatedly incurred.

Identifying locations in the code to efficiently transfer data is simple for kernels that execute code not affected by control flow, but when nested functions, loops, or control-flow are introduced, it becomes more difficult. This is because efficiently scheduling data transfers requires program-wide control-flow and data-flow analyses to determine which control-flow paths have different data, and when data is needed on each device. The difficulty of these analyses greatly increases when these complex code structures are introduced, and even more so when there are multiple
kernels in the program, each in their own complex control-flow. To properly determine efficient locations to transfer the data, interprocedural data-flow and control-flow analyses are needed. For simple programs, users may manually perform data-flow and control-flow analyses on the source code to determine when to transfer data, but when these complex code structures are introduced, it can become extremely difficult to perform these analyses manually.

Fortunately, complex data-flow and control-flow analyses can be developed for compilers. As programs accelerated by GPUs already rely on compilers to generate their executable, they are uniquely situated to perform these analyses. The remainder of this chapter will demonstrate compiler analyses and techniques to efficiently schedule data transfers.

This chapter presents the motivating example, techniques, and quantitative results originally presented in "Compiler Optimization of Accelerator Data Transfers" by Matthew Ashcraft, Alexander Lemon, David Penry, and Quinn Snell [1], and is organized as follows. Section 3.1 presents the objectives and goals of this contribution. Section 3.2 discusses the context in which these techniques are developed and used. Section 3.3 presents a motivating example to our work, and demonstrates improvements that can be made to scheduling data transfers through example code. Section 3.4 discusses the techniques in each of the five steps necessary to optimize the scheduling of data transfers. Section 3.5 discuss the experimental methodology. Section 3.6 discusses the testing results, and the impact scheduling data transfers had on program execution. Section 3.7 summarizes the chapter.

3.1 Objective and Metrics

The objective of the analyses and techniques presented in this chapter are focused on improving the runtime of GPU-accelerated programs by reducing the frequency and size of expensive data transfer operations, which transfer data between the host memory and GPU memory. For our experiments, the baseline compare against is a novice programming approach, where all kernel data is transferred to the kernel immediately before execution, and after the kernel executes, the modified data is immediately transferred back. This is demonstrated in Figure 3.1. Scheduling data transfers in this way is often found in simple applications where kernel invocations are not located in looping control-flow, and/or in applications where the kernel is nested deep within nested
function calls. In more complex applications, it may be difficult for a programmer to determine the optimal locations in the code to perform data transfers, so our objective is to do it automatically.

While programmer may often do better than the "novice scheduling" approach we use as a baseline, the object of this work is to determine how well this scheduling can be done automatically. That is, while programmers may be able to determine locations in the code to transfer data better than the novice approach, it takes time for the programmer to do this analysis. If we can demonstrate that this can be automated, we can reduce programmer effort in most cases. Additionally, we may be able to improve program performance in cases where the programmer fails to recognize an opportunity for improved data transfer scheduling.

It is challenging to measure exactly how much performance improvement our technique would provide in real-world situations, as guessing where users might schedule data transfers in programs with nested function calls or looping control-flow is difficult, as it is dependent on how much effort users put in to manually analyze the program. For these reasons we feel that comparing against the "novice scheduling" approach is appropriate.

In addition to this, we will manually perform data-flow and control-flow analyses on each of the benchmarks to identify the optimal locations to transfer data to reduce the impact on execution time. These results will be compared with the automatically scheduled data transfers, from the analyses in this chapter, to help determine the effectiveness of the analysis.

3.2 Context

The analyses and techniques presented in this chapter are designed for heterogeneous systems with a CPU and GPU, connected over PCIe, as shown in Figure 3.1. This is a common setup when using GPUs to accelerate programs. As is usual, the main software is executed on the host(CPU), with certain parts of the program being accelerated on the GPU. The PCIe interface allows the CPU to have access to the dedicated GPU memory present on the GPU accelerator card. In general, any datasets used by the GPU need to be present in GPU memory, which is specifically designed to accommodate the high-bandwidth access required by the parallel execution of the GPU-accelerated software. This means that datasets need to be explicitly transferred back and forth between the CPU and GPU memory buffers.
In terms of software frameworks, the techniques presented in the chapter were developed for C-code benchmarks with OpenMP directives to specify parallel portions of the code to accelerate using the GPU. OpenMP is a library extension to the C programming language, which allows users to optimize C source code with parallelization directives. We use the CUDA backend of the LLVM compiler framework to compile the OpenMP-annotated code to CUDA GPU code. CUDA is a C-based programming language exclusively for NVIDIA graphics cards. The communication between the CPU and the GPU is handled by explicit communication in the CUDA code based off of markers inserted into the code, representing the results of our scheduling technique. The granularity of these data transfers is either single variables, or entire data arrays. Note that the use of OpenMP and CUDA can be replaced with any supporting frontend or backend language for accelerators that is supported by the compiler.

Though our techniques are developed for heterogeneous systems with a CPU and GPU connected over PCIe, they can be applied to other types of systems. They are applicable to any program in which the software code and the kernel code can be analyzed together. This prevents the use of our techniques in programs where the kernel code is handled separately from the software code, such as with OpenCL kernels. Additionally, because our techniques operate in the middle-
end of the compiler, our techniques are applicable to other accelerators, such as FPGAs or many-core processors.

The flow of our tests is as follows. The source code, with OpenMP directives, is analyzed in the middle-end of the compiler to extract the OpenMP directives representing the location of the kernel. Our analyses are then performed to efficiently schedule data transfers. After our analyses are finished, the LLVM CUDA backend replaces our scheduled data transfer calls with explicit data transfers supported by NVIDIA GPUs.

3.3 Motivating Example

An example program which helps demonstrate the value of scheduling of data transfers can be seen in Listings 3.1, 3.2, and 3.3. These examples represent various improvements to scheduling

```c
for (int J = 0; J < JBounds; ++J){
    #kernel1
    #Transfer Host to GPU X, Y, Z
    for(int YI = 0; YI < YBounds; ++YI) {
        Y[YI] = X[Bounds] + Z[YI];
    }
    #Transfer GPU to Host Y

    #kernel2
    #Transfer Host to GPU X, Y, Z
    for(int XI = 0; XI < XBounds; ++XI) {
        X[XI] = Y[Bounds] + Z[XI];
    }
    #Transfer GPU to Host X

    for(int WI = 0; WI < WBounds; ++WI) {
        W[WI] = X[Bounds / WI] + X[WI];
    }
}
for(int WI = 2; WI < WBounds; ++WI) {
    W[WI] = Y[WI] / W[WI - 2];
}
```

Listing 3.1: Novice Scheduling Example
data transfers, each of which require greater effort in analyzing the program. The code structure for these examples is similar to the particlefilter benchmark from the Rodinia benchmark suite [52]. The interesting aspect of this code is that the kernels are contained within a loop, with data arrays being used both in and out of kernels, forcing data to be transferred back and forth between the hardware and software repeatedly.

The simplest method for scheduling data transfers would be if the user placed their explicit data transfers immediately before and after each Kernel invocation as shown in Listing 3.1. In this example, each of the kernels consists of a simple loop, and the kernel1 and kernel2 are simply the names of the kernels. Under more complex examples, the loop might have different bounds, or perform different operations, but they are still single or nested loops. Under this method, memory locations read from in the kernels are transferred from software to the hardware immediately before the kernel executes, and memory locations written to in the kernels are transferred from the hardware to the software immediately after the kernel finishes executing. Additionally, unless the user has performed a thorough analysis of the memory accesses in the kernels and knows which elements of the data arrays are accessed or modified during execution, then the entire data array needs to be transferred, rather than just the few elements being accessed.

In Listing 3.1, there are four data arrays being used by the hardware and software; W, X, Y, and Z. Data array W is only used in the software code, not in the kernels, meaning it does not need to be transferred to the hardware, or GPU. X and Y are used on both the hardware and software, and more importantly, are both read from and written to in the kernels, meaning they need to be transferred back and forth between hardware and software throughout execution. Z is only read from in the kernels, meaning that it needs to be transferred to the kernels, but not transferred back.

As mention previously, the simplest or novice scheduling technique is for the user to insert data transfers immediately before and after the kernel executes. For this example, kernel1 reads from X and Z, and writes to Y, meaning that all three data arrays need to be transferred to the hardware immediately before the kernel executes, and only Y needs to be transferred back immediately afterwards. Note that Y is transferred to the hardware even though the kernel does not read from it. This is because we do not know which portions of the array the kernel will write to. Depending on the accelerator, this can prevent uninitialized regions of memory on the hardware from writing garbage to the data array in software when the data is returned.
In kernel2, X is written to, and Y and Z are read from. Similar to kernel1, all three objects are transferred to the hardware immediately before the kernel executes, but only X is transferred back immediately afterward. Note that since both of these kernels are contained within an outer loop, the data arrays are transferred back and forth between the hardware and software with each iteration of the loop.

With this example, there are some clear ways to improve the scheduling of data transfers to reduce their overall impact. One improvement, as shown in Listing 3.2, would be to transfer all the data to the hardware before kernel1 executes, then leave it there until after kernel2 finishes executing, at which point X and Y are transferred back. This would reduce the total number of data transfers per iteration of the outer loop from eight to five, a 37.5% reduction. This is fairly obvious for this example, however, if these kernels were located in different functions in various parts of the program it would become substantially more difficult to perform interprocedural data

```
for (int J = 0; J < JBounds; ++J) {
    #kernel1
    #Transfer Host to GPU X, Y, Z
    for (int YI = 0; YI < YBounds; ++YI) {
        Y[YI] = X[Bounds] + Z[YI];
    }

    #kernel2
    for (int XI = 0; XI < XBounds; ++XI) {
        X[XI] = Y[Bounds] + Z[XI];
    }
    #Transfer GPU to Host X, Y
    for (int WI = 0; WI < WBounds; ++WI) {
        W[WI] = X[Bounds / WI] + X[WI];
    }
    for (int WI = 2; WI < WBounds; ++WI) {
        W[WI] = Y[WI] / W[WI - 2];
    }
}
```

Listing 3.2: Improved Scheduling Example
and control-flow analyses necessary to determine that this is possible. These types of analyses are used to determine where data is used in the program, and the branches taken in the program.

Another improvement would be to look for places where data can be scheduled results in less data transfers, such as moving the data transfers outside of the outer loop that encompasses the kernels. Within the outer loop in this example, \( Z \) is only read from within the kernels. This means that \( Z \) could be transferred to the hardware before the outer loop begins executing. Similarly, within the outer loop \( Y \) is only read from or written to within the kernels, meaning it can be transferred to the hardware before the outer loop begins executing, and be transferred back to the software after the outer loop finishes executing. \( X \) however is used in both the hardware and software within the outer loop, meaning it still has to be transferred back and forth with each iteration of the outer loop.

The end result of optimizing the placement of data transfers for this example is shown in Listing 3.3:

```plaintext
# Transfer Host to GPU Y, Z
for (int J = 0; J < JBounds; ++J){
    # kernel1
    # Transfer Host to GPU X
    for (int YI = 0; YI < YBounds; ++YI) {
        Y[YI] = X[Bounds] + Z[YI];
    }
    # kernel2
    for (int XI = 0; XI < XBounds; ++XI) {
        X[XI] = Y[Bounds] + Z[XI];
    }
    # Transfer GPU to Host X
    for (int WI = 0; WI < WBounds; ++WI) {
        W[WI] = X[Bounds / WI] + X[WI];
    }
    # Transfer GPU to Host Y
    for (int WI = 2; WI < WBounds; ++WI) {
        W[WI] = Y[WI] / W[WI - 2];
    }
}
```

Listing 3.3: Optimized Scheduling Example
3.3. Scheduling the data transfers as shown would further reduce the number of data transfers each iteration of the loop from 5 to 2, greatly reducing the data transfer overhead for this example code.

3.4 Data Transfer Analysis and Scheduling

This section describes the techniques to analyze and schedule the data transfers in more efficient locations. The techniques assume that the locations of the kernels in the code are known. We describe how we gather this information for our testing in Section 3.5.1. The techniques for efficiently scheduling data transfers can be broken up into 5 steps:

1. Discover memory locations accessed by kernels,
2. Discover conflicting memory accesses,
3. Generate a scheduling graph,
4. Schedule data transfers,
5. Insert markers.

These techniques were designed and implemented around the middle-end of the LLVM compiler framework [21], which is agnostic to the source-code language used, and the architecture of the targeted accelerator device.

3.4.1 Discover Memory Locations Accessed by Kernels

One of the difficulties to dealing with the IR of high-level languages in compilers is handling memory accesses and aliasing. Aliasing is when multiple pointers point to the same location. In order to identify the memory locations accessed in the kernels, and later identify the conflicting memory accesses, we adapted an interprocedural points-to analysis based upon Lattner’s Data Structure Analysis [53].

Lattner’s data structure analysis is broken up into three parts. First each function is analyzed to determine local aliasing. The memory accesses for each function are represented as nodes in a graph with edges representing aliasing. Next all of the aliasing information is passed down the
call graph through each function call, propagating the aliasing information from the top of the call graph to the bottom. After all of the aliasing information is passed down to the end of the call graph, the aliasing information is passed up the call graph from callees to callers until the top-level function is reached, propagating all of the information collected at the end of the call graph back to the beginning. This provides one of the best compile time alias analyses on interprocedural programs. The end result is an interprocedural, context-sensitive, field sensitive analysis.

This analysis provides a mapping of all memory locations used within the kernel, and can be used to generate the initial conservative data transfers for memory accesses that are not allocated within the kernel. For our techniques, it is assumed that memory locations allocated within the kernels are not used in the software, and thus do not need to be transferred. This is also verified through the data structure analysis. For scheduling data transfers, all memory locations marked as read within the kernels are initially scheduled to be transferred from the software to the hardware, and all memory locations written to in the kernels are initially scheduled as being transferred from the software to the hardware, and then back to the software. This was necessary in our environment to prevent the hardware from writing uninitialized data to the data arrays in the software when transferring data back from the hardware.

3.4.2 Discover Conflicting Memory Accesses

Having identified the variables which need to be transferred between the hardware and software, the bounds for where the data transfers can be scheduled need to be determined. One of the limiting factors in moving data transfers is the conflicting memory accesses. These are memory accesses outside of the kernel that could access the same locations accessed inside the kernel.

The results of the data structure analysis described in Section 3.4.1 provide all of the aliasing information in the program, and particularly, those for conflicting memory locations. Note that this aliasing information is conservative, as some aliasing can not be determined until runtime. Through analyzing the generated graph of nodes and edges, representing memory locations and aliasing respectively, all of the instructions that could access the same memory locations accessed in the kernel are identified. These are called conflicting instructions. If the conflicting instructions are executed before the kernel executes, then the memory locations accessed by those instructions cannot be transferred to the kernel until after the conflicting instructions have finished executing.
If the conflicting instructions execute after the kernels, then the aliasing memory locations in the kernel have to be transferred back to the software before the conflicting instructions execute. A similar situation is demonstrated in Listing 3.3, where \( X \) has to be transferred from the GPU to the host before it is needed in the host loop on line 16.

In addition to identifying the conflicting memory locations, the data structure analysis is used to identify the memory allocation instructions (both heap and stack) for locations that are transferred between the hardware and software. These allocation instructions are needed by the host runtime for the allocation and mapping of memory on the hardware.

### 3.4.3 Generate a Scheduling Graph

Analyzing the data-flow, control-flow, and call graphs is necessary to efficiently schedule the data transfers. Instead of analyzing these individually, they are merged together to create a scheduling graph. Essentially, the data-flow and control-flow graphs are merged together, then the call graph is used to inline all of the data-flow and control-flow graphs from other functions of interest. The functions of interest are those which contain either kernels or conflicting memory accesses. Additionally, because functions of interest may be contained in different parts of the call graph, all of the functions on the paths of each function of interest back to their common ancestors in the call graph are inlined into the scheduling graph. This ensures that the interprocedural analysis for scheduling data transfers is able to identify all paths between conflicting memory instructions and kernels.

An example of what this scheduling graph looks like is shown in Figure 3.2, and is based upon the example code shown in Listing 3.1. In the scheduling graph the control-flow between basic blocks is maintained, and the instructions in each basic block are the contents of the nodes. The exception to this is function calls in the middle of the basic block, as shown in Figure 3.3. In the figure, there is a call to `ComputeVal` in the middle of the basic block. To inline the call graphs, these nodes are split in two, the first node contains all of the instructions up to the function call, and the second node contains the remaining instructions in the basic block after the function call. The edge from the first node points to the entry node of the graph of the function called, and the edge from the last node in the graph of the called function points to the node containing the instructions after the function call. The result of doing this on Figure 3.3 is shown in Figure 3.4.
In addition to this, the scheduling graph maintains context sensitivity. This means that the graphs of functions of interest with multiple call sites, as in functions that are called from more than one location, are inlined at each of the locations they are called from. However, call sites within the kernels are ignored. Kernels are treated as a single node rather than split up by basic blocks, as shown in Figure 3.2. This is because the analysis later performed on the scheduling graph is looking for where to place data transfers outside of the kernels, not in them.

Lastly, the back edges for this graph are handled differently. To ensure that the graph is acyclic for the later analyses, the back edges are placed in a separate data structure. Note that the
loop edges in the graph from the loop header to the loop exit are forward edges for when the loop conditions are not met, and the loop immediately exits. Additionally, to ensure the graph is acyclic, recursion is not supported.
3.4.4 Schedule Data Transfers

In order to determine locations to schedule data transfers that reduce the impact on performance, two analyses need to be performed. These analyses need to provide information on the paths through the scheduling graph, and when certain variables are active. To gather this information, a modified liveness and dominator tree analyses are performed.

The liveness analysis is a data-flow analysis used to determine when variables are active, or currently in use. This information is necessary in determining the life of variables both in hardware and software. For example, in Listing 3.3, liveness analysis determined that \( Y \) was only live on the hardware throughout the execution of the outer loop. Because of this, the data transfers for \( Y \) were moved from before and after the kernels to outside of the outer loop. The modified liveness analysis will be discussed later in this section.

The dominator tree analysis information is used to ensure data transfers are not duplicated, and that they are kept out of loops. When analyzing the paths to and from the kernels, it is possible for multiple data transfers for the same memory location to be inserted on multiple paths. The information from the dominator tree analysis aids in ensuring no paths to or from the kernels were missed, and that data transfers were not duplicated. Additionally, the information from dominator tree analysis is used to ensure the data transfers are not placed within loops the kernel invocation did not originate in, which would likely worsen the overhead. The generation of the dominator tree analysis will be discussed further later in this section.

After discussing the implementation of liveness and dominator tree analysis, how they are used in scheduling data transfers will be discussed.

Liveness Analysis

To schedule efficient data transfers, it is important to understand when variables are live, or in use, on either the software or hardware. This information is provided by liveness analysis. Liveness analysis is a well-defined data-flow analysis which tracks when variables are live, or in use. It is almost like scoping information for a variable, but instead of being alive from when it is created until it is deleted or the function ends, it is live from its creation until its last use. Liveness analysis can determine if variables in the kernel need to be transferred to or from the hardware,
if variables can be created in the kernel, or if variables can remain on the hardware in-between kernels. The work by Lee et al. [24], [25] also utilized liveness analysis to eliminate redundant data transfers between kernels, however their work only analyzed kernels located immediately after one another in the source code.

The liveness analysis we developed is different from the traditional liveness analysis because it keeps track of variables in both the software code and the kernels, including data transfers between the two. This version of the liveness analysis is shown in Algorithm 1.

Initially, the analysis goes through all of the nodes in the scheduling graph and sets the inputs and outputs (lines 1-11). For kernels, the inputs ($HWIN$) and outputs ($HWOUT$) of each node are set to the memory locations that need to be transferred to ($TIN$) and from ($TOUT$) the kernels. For nodes representing software code in the scheduling graph, the inputs ($IN$) are set to memory locations used in the node ($USE$) and the outputs ($OUT$) are set to an empty set. After initializing the inputs and outputs, all of the nodes ($N$) are added to the worklist.

Next the worklist is processed until empty. The next node in the worklist is retrieved, then traditional liveness analysis is performed on it (lines 16-21). First the nodes outputs are set to the union of the inputs of all the nodes successors. Next a temporary copy of the inputs ($oldIn$) is made, and the new inputs are computed. The new inputs list is a union of uses in the node and the outputs of the node that are not defined ($DEF$) in the node. In other words, if a variable is used by a successor of the node, but it is not defined by the node, then it must be passed in from a previous node. If the new list of inputs differs from the old input list, then all of the nodes predecessors are added to the worklist.

The next portion of the algorithm (lines 23-42) is the new portion of the analysis designed to handle the kernels and the flow of variables between the hardware and software. The analysis looks for the boundary node for the data transfers. The boundary nodes are the earliest or latest points in the program where data transfers may be scheduled. These nodes provide a window in the scheduling graph where the data transfers may be scheduled. To find the boundary nodes, the inputs to the kernels ($HWIN$) are propagated backwards from the kernels, towards the beginning of the program, until they reach conflicting instructions. Similarly, the outputs from the kernels ($HWOUT$) are propagated from the kernels towards the end of the scheduling graph until they reach conflicting instructions. Note that each node has a list of hardware inputs ($HWIN$) and
outputs (*HWOUT*). These represent variables that are transferred to the hardware in or before the node for inputs, or in or after the node for outputs.

The remainder of the algorithm operates as follows. First the hardware inputs of the current node are backed up to *oldHWIn*, and a list of the hardware inputs of the current nodes successors are assigned to *HWSuccIn*. Next, any hardware inputs to the nodes successors that are not defined in the node are added to the nodes hardware inputs. If a node is found that defines its successors input, then that node is the boundary node for that memory location. If the new list of hardware inputs is different from the previous list, then all of the node predecessors are added to the worklist (lines 25-32). This analysis is then repeated for the outputs, but instead of looking for the nodes successors inputs not defined in the node, it looks for predecessors hardware outputs that are not defined or used in the node (lines 32-42). If the predecessors hardware outputs are used or defined in the node, then that node is a boundary for those memory locations. Otherwise the hardware output for the node is updated, and the nodes successors are added to the worklist.

After the liveness analysis has completed, the input and output arrays for each node are collected for both hardware and software (*IN*, *OUT*, *HWIN*, *HWOUT*). These arrays are later used to determine where the data transfers may be scheduled. Additionally, *HWIN* and *HWOUT* keep track the hardware inputs and outputs in relation to each kernels. This is necessary as the boundary node for a memory location may dominate one kernel it is used in, but not others. If it was unknown which kernel the memory location was used in it could severely limit the scheduling of data transfers.

**Algorithm 1** Liveness Analysis

**Require:** *N* is the set of nodes  
**Require:** *IN*(n): variables passed into node *n*  
**Require:** *OUT*(n): variables passed out of node *n*  
**Require:** *SUCC*(n): successors of node *n*  
**Require:** *PRED*(n): predecessors of node *n*  
**Require:** *DEF*(n): variables defined in *n*  
**Require:** *USE*(n): variable uses in node *n*  
**Require:** *K*: set of kernel nodes  
**Require:** *HWIN*(n): hardware variables passed into node *n*  
**Require:** *HWOUT*(n): hardware variables passed out of node *n*  
**Require:** *TIN*(n): naive transfers to the hardware for kernel node *n*  
**Require:** *TOUT*(n): naive transfers from the hardware for kernel node *n*
for all $n \in N$ do

if $n \in K$ then

\[ \text{Initialize kernel nodes} \]

\[ \text{HW IN}(n) = \text{T IN}(n) \]

\[ \text{HW OUT}(n) = \text{T OUT}(n) \]

else

\[ \text{Initialize non-kernel node} \]

\[ \text{IN}(n) = \text{USE}(n) \]

\[ \text{OUT}(n) = \emptyset \]

end if

end for

worklist $\leftarrow N$

while worklist $\neq \emptyset$ do

$n \leftarrow \text{worklist}.\text{pop}()$

\[ \text{Host variable analysis} \]

\[ \text{OUT}(n) = \bigcup_{s \in \text{SUCC}(n)} \text{IN}(s) \]

\[ \text{old IN} = \text{IN}(n) \]

\[ \text{IN}(n) = \text{USE}(n) \cup (\text{OUT}(n) - \text{DEF}(n)) \]

if old IN $\neq \text{IN}(n)$ then

worklist $\leftarrow$ worklist $\cup \text{PRED}(n)$

end if

oldHW IN $= \text{HW IN}(n)$

HW Succ In $= \bigcap_{s \in \text{SUCC}(n)} \text{HW IN}(s)$

for all $v \in \text{HW Succ In}(n)$ do

if $v \notin \text{DEF}(n)$ then

\[ \text{HW IN}(n) \leftarrow \text{HW IN}(n) \cup v \]

end if

end for

if oldHW IN $\neq \text{HW IN}(n)$ then

worklist $\leftarrow$ worklist $\cup \text{PRED}(n)$

end if

oldHW Out $= \text{HW OUT}(n)$

HW pred out $= \bigcap_{p \in \text{PRED}(n)} \text{HW OUT}(p)$

for all $v \in \text{HW pred out}$ do

if $v \notin \text{USE}(n)$ and $v \notin \text{DEF}(n)$ then

\[ \text{HW OUT}(n) \leftarrow \text{HW OUT}(n) \cup v \]

end if

end for

if oldHW Out $\neq \text{HW OUT}(n)$ then

worklist $\leftarrow$ worklist $\cup \text{SUCC}(n)$

end if

end while
Generate Dominator Tree

Dominator tree analysis was implemented on the scheduling graph to prevent the insertion of duplicate data transfers on the same path, to ensure data transfers are not unnecessarily scheduled in loops, and to ensure all paths to and from kernels have the correct data transfers. Dominator tree analysis is a well defined analysis that determines dominators. By definition, node $X$ dominates $Y$ if every path in the program to $Y$ goes through $X$.

Additionally, each node has an immediate dominator, which is generally the closest dominator to a given node. In other words, if there are three sequential nodes in the graph, $X$, $Y$, and $Z$, where $X$ goes to $Y$, and $Y$ goes to $Z$, then $X$ dominates $Y$ and $Z$, and $Y$ dominates $Z$. The immediate dominator of $Y$ is $X$, and the immediate dominator of $Z$ is $Y$. There can only be one immediate dominator for any given node.

In addition to dominators and immediate dominators, post-dominators and immediate post-dominators are needed. Post-dominators and immediate post-dominators follow the same rules as dominators and immediate dominators, except that instead of all paths in the program from the beginning to $Y$ passing through $X$, all paths in the program from $Y$ to the end of the program pass through $X$.

The implementation of the dominator tree analysis on the scheduling graph is based off the dominator tree analysis defined by Tarjan and Langauer [54]. The original analysis provides dominators and immediate dominators. Additionally, it has been implemented with predecessors and successors swapped to get the post-dominators and immediate post-dominators of all of the nodes. When scheduling the data transfers, the dominator and post-dominator tree information is used to ensure that all scheduled data transfers dominate or post-dominate the kernels that the memory locations are used in. To simplify later analyses, the results of the dominator tree and post-dominator tree analysis are copied into each of the nodes affected. Using the previous example, that would mean $X$ would have no dominators, and $Y$ and $Z$ as post-dominators, $Y$ would have $X$ as its dominator and $Z$ as its post dominator, and $Z$ would have $X$ and $Y$ as dominators, and no post-dominators. The result of copying the dominator and post-dominator tree information to each of the nodes, is that now each of the nodes contains a list of its ancestors and descendants, or predecessors and successors.
Scheduling Data Transfers

From the results of liveness analysis the window in which data transfers may be scheduled is identified. Applying the results of the dominator and post-dominator tree analyses to this window further narrows down the locations in which data transfers should be scheduled.

The liveness analysis provides the boundary nodes for memory locations transferred to and from the hardware. For transfers from the software to the hardware, the boundary nodes are the last locations the variable is defined in before the kernel invocation on all paths to the kernel invocation. Additionally, the data transfers from the software to hardware must happen after any converging control-flow with differing hardware inputs and software outputs. These locations are commonly found on the back edges of loops containing kernel invocations or nested loops with conflicting memory accesses. Under these situations, the same variable can end up as a hardware input to the loop header node, and a software output coming from the back edge into that node. This would necessitate the data transfer from the software to the hardware be scheduled at the beginning of the loop. This type of situation is seen for memory location X in Listing 3.3 because X is used in the kernels, but it is also used in the software before the loop back edge.

For data transfers from the hardware to the software, the boundary nodes are the first nodes that contain variable uses or definitions of the memory locations. In other words, the boundary nodes for data transfers from the hardware to the software represent the first locations in which the variables in the hardware are needed in the software, and thus the data needs to be transferred back. Similar to transfers from software to hardware, converging control-flow for transfers from hardware to software require data transfers to be scheduled. If the same memory location is accessed on the software followed by the hardware in the same loop, then the data has to be transferred back to the software before the back edge of the loop. Under these situations the data is transferred back and forth between the hardware and software each iteration of the outer or encompassing loop.

The algorithm for scheduling data transfers is shown in Algorithm 2. The algorithm analyzes the boundary nodes in relation to each memory location and the kernels they are used in (line 1). The first part of the algorithm (lines 3-10) uses dominator tree analysis to determine if then boundary nodes are good places for the transfers to be scheduled.

The boundary node is considered to be a good location to schedule the data transfer if the memory location being analyzed dominates the kernel it is used in. This is important under
Algorithm 2 Scheduling Transfers

Require: $b$ is the boundary node
Require: $c$ is the memory location accessed by a boundary node
Require: $K(b)(c)$ contains the kernels affected by $c$ in $b$
Require: $\text{DOM}(n)$ is an ordered list of dominators of node $n$
Require: $\text{DESC}(n)$ contains all descendants of node $n$
Require: $C(n)(c)$ contains the scheduled transfers of $c$ for kernel node $n$

1: for all $k \in K(b)(c)$ do
2: \hspace{1em} $\text{NewTransfer} = k$
3: \hspace{1em} if $b \notin \text{DOM}(k)$ then
4: \hspace{2em} for all $d \in \text{DOM}(k)$ do
5: \hspace{3em} \hspace{1em} if $d \in \text{DESC}(b)$ and $d \notin \text{DOM}(b)$ then
6: \hspace{4em} \hspace{1em} \hspace{1em} $\text{NewTransfer} = d$
7: \hspace{3em} \hspace{1em} else
8: \hspace{3em} \hspace{1em} \hspace{1em} break
9: \hspace{3em} \hspace{1em} end if
10: \hspace{2em} end for
11: \hspace{1em} else
12: \hspace{2em} $\text{NewTransfer} = b$
13: \hspace{1em} end if
14: if $C(n)(c) \neq \emptyset$ then
15: \hspace{1em} $\text{CurrentTransfer} = C(n)(c)$
16: \hspace{1em} if $\text{CurrentTransfer} \notin \text{DOM}(\text{NewTransfer})$ then
17: \hspace{2em} $C(n)(c) = \text{NewTransfer}$
18: \hspace{1em} end if
19: else
20: \hspace{1em} $C(n)(c) = \text{NewTransfer}$
21: end if
22: end for

In a few situations. If the memory location transferred from the software to the hardware are all updated or modified in a loop the kernel is not in, then placing the data transfer in that loop may result in many unnecessary data transfers, likely greatly reducing program performance. The other situation involves if-else statements. If a memory location transferred to the hardware is modified by a branch of an if-else statement, but not in the other branch, then scheduling the data transfer in the branch modifying it will not guarantee the data is available on the hardware. In both of these cases, the solution is to move the data transfer to the closest successor or descendant in the scheduling graph that dominates or post-dominates the associated kernel.
To identify if the boundary node is a good place to schedule the data transfer, the algorithm checks if each memory location associated with the boundary node dominates the kernel it is used in. If so, the node is set to be the new transfer location for the memory location. If not, then the analysis goes through the ordered list of dominators for the kernel (line 4), beginning at the kernel and going up to the entry of the program, and looks for the last dominator that is still a descendant of the boundary node. This is where the data transfer is scheduled. Note that on line 5 it checks that the descendant does not dominate the boundary node. This can happen when the kernel and the boundary node are within the same loop.

Next the algorithm checks to see if the newly scheduled data transfer for the memory location has already been scheduled by another kernel (lines 14-21). If it has already been scheduled, then the node which most closely dominates the kernel is chosen as the new scheduling location.

### 3.4.5 Insert Markers

Based on the final scheduling locations from Algorithm 2, markers are inserted into the code to indicate to the hardware code generator where the data is supposed to be transferred. These markers are later identified and replaced in the target architecture back-end with calls to transfer data.

### 3.5 Experimental Methodology

In order to demonstrate the effectiveness and importance of scheduling data transfers, the techniques were tested on four benchmarks from the Rodinia Benchmark suite [52], and a bespoke hotplate benchmark, which will be described in Section 3.5.3. These benchmarks were executed

```c
#pragma omp parallel for shared(weights, Nparticles) private(x)
for (x = 0; x < Nparticles; x++){
    weights[x] = 1/((double)(Nparticles));
}
```

Listing 3.4: Optimized Scheduling Example
To support the accelerator in the source code language, OpenMP directives included in the benchmarks were used. In the benchmarks, data-parallel loops were marked using the OpenMP parallel for directives. As the hotplate benchmark is not from the Rodinia benchmark suite, it was manually annotated. An example of these annotations from the particlefilter benchmark is shown in Listing 3.4, where the annotation is placed immediately before the loop, specifies the loop is parallel, and specifies the shared and private variables. The handling of OpenMP directives will be discussed in Section 3.5.1.

As mentioned previously, all of these techniques were implemented in the middle-end of the LLVM compiler framework version 3.8, using LLVM’s front-end compiler, Clang.

Next, Sections 3.5.1 and 3.5.2 will discuss the translation from OpenMP to our framework, and the generation of GPU code as the target architecture. This will be followed by an explanation of the benchmarks used in Section 3.5.3.

### 3.5.1 Translation from OpenMP

Within the IR of the LLVM framework, most of the calls to OpenMP directives are automatically replaced with Intel OMP runtime calls by the Clang front-end compiler. The Intel OMP runtime calls are designed to take advantage of parallelism during execution. Some OpenMP directives, such as `firstprivate` are just replaced with local variables within the kernels. The only remaining calls are to some of the OpenMP API libraries. Thus to determine the locations of kernels, parallel loops, and reductions in the benchmarks, the IR is searched to identify all of the the API calls which create OpenMP tasks, initiate and terminate loops, and handle reductions. These are used to reconstruct the original structure of the OpenMP directives.

After the original structure of the OpenMP directives has been reconstructed, the kernel code is analyzed to ensure there are no invalid operations for the GPU code generation. These invalid operations come in the form of library calls that are not supported by the CUDA libraries used in the GPU code generation. Additionally, the GPU code generator is unable to support the transfer of pointers-to-pointers to the GPU. To help prevent these from arising, loop-invariant loads
are hoisted out of the kernel invocation loops. This hoisting allows globals pointing to arrays to be used in benchmarks.

### 3.5.2 GPU Code Generation

In order to generate code for different devices, the LLVM module containing the program is split into two different modules. One contains the original source code with the exception of the kernels, and the other contains the kernels.

In the module containing the kernels, the loops are strip-mined to divide them among threads in a grid-stride fashion. Next the math library function calls are replaced with those from NVIDIA's *libdevice* library, and the module is compiled with the NVPTX back-end. The NVPTX back-end is a back-end for the LLVM compiler targeting GPU architectures.

Runtime calls are inserted into the kernel invocation locations. These are calls to a wrapper for the CUDA device API, which handles device memory management and the final reduction of partial results. The source code module is then compiled as the host driver for the program. The parameters for the GPU are stored in a configuration file that is loaded at runtime, allowing for some post-compiler performance tuning.

### 3.5.3 Benchmarks

The benchmarks tested from the Rodinia benchmark suite include *Breadth-First Search*, *particlefilter*, *Speckle Reducing Anisotropic Diffusion*, and *Needleman-Wunsch*. These benchmarks stand out for their use of data parallel loops, and their reuse of memory locations in multiple kernels. Benchmarks with single kernels and no control-flow around them would not benefit from our analyses, and can be assumed to achieve the same performance with and without our efficient data transfer scheduling.

*Breadth-First Search* (*bfs*) traverses all connected nodes in a graph, and contains of two kernels, both of which reside in an encompassing loop.

*particlefilter* is a medical imaging benchmark used for tracking cells in video frames. It contains ten kernels, some of which are invoked from within an encompassing loop.
variables are modified in the encompassing loop in both the hardware and software, necessitating the repeated transferring of data between the hardware and software.

*Speckle Reducing Anisotropic Diffusion (srad)* is an algorithm based upon partial differential equations. It is used to remove spots from ultrasonic and radar applications without affecting important features. It contains two kernels within an encompassing loop. Within the encompassing loop the main array is read on the hardware and software, forcing data to be transferred between the hardware and software each iteration of the loop.

*Needleman-Wunsch (nw)* looks for the optimal alignment in DNA sequencing. It also contains two kernels within an encompassing loop. In our testing, we measured the results of this benchmark with different input variables representing the size of the data array, 2048 and 4096.

The *hotplate*, whose outline is shown in Figure 3.5, computes a finite differences approximation for simple heat transfer using a 5-point stencil. It uses Jacobi iteration on a 2D regular mesh represented as a 2D array of values. There are three main kernels in the benchmark, *INIT*, *COMPUTE*, and *COUNT*. *INIT* initializes the main array. *COMPUTE* is the main compute kernel invocation contained within an encompassing loop. *COUNT* performs a reduction on the array results from *COMPUTE*. This benchmark is particularly interesting because the data is initialized, computed, and counted in different kernels on the hardware. This means all of the data transfers for

```
Require: current and next are the stencil arrays
Require: keepGoing is the finished flag
Require: INIT initializes the arrays in a kernel
Require: COMPUTE performs the stencil computation on current and next within a kernel and returns if the computation is finished
Require: swap swaps the current and next pointers
Require: COUNT analyzes the result of the stencil computation within a kernel
    for i = 0; i < numTrials; ++i do
        keepGoing = 1
        INIT(current, next)
        while keepGoing == 1 do
            keepGoing = COMPUTE(current, next)
            swap(&current, &next)
        end while
        numHotCells = COUNT(current)
    end for
```

Figure 3.5: *hotplate* pseudo-code
the large stencil arrays, current and next, can be removed. The reduction values from COMPUTE and COUNT still need to be transferred from the hardware to the software. Additionally, because the swap function swaps pointers not values, the pointer arguments to the COMPUTE kernel are swapped each iteration, and the runtime maps them to the correct memory location on the GPU each iteration. In our testing, we measured the results of this benchmark with different sizes of the data array: 1024, 2048, and 4096.

Note that among the results, there are multiple entries for the nw and hotplate benchmarks for different input sizes. This is because these two benchmarks were most affected by the scheduling of data transfers.

3.6 Experimental Results

To determine the impact of scheduling data transfers, the impact on run time, the number of bytes transferred throughout execution, and the total number of transfer calls made by the scheduled data transfers were measured. The results are a comparison between using the techniques introduced, and the baseline, which transferred data immediately before and after each kernel. For the baseline, all memory locations read in the kernels were transferred from the software to the hardware, and all memory locations written to in the kernels are transferred from the software to the hardware before the kernel executes, then back to the software after the kernel executes. In addition we manually performed data-flow and control-flow analyses on each benchmark to determine the optimal placement of data transfers, and compared that with the generated results.

The results of the tests are shown in Figures 3.6, 3.7, 3.8, and the corresponding Tables 3.1, 3.2, and 3.3. Figure 3.6 and Table 3.1 show the relative number of transfers between the hardware and software compared to the baseline, or in other words, the number of times the function call wrapper for data transfers was called. Figure 3.7 and Table 3.2 show the relative number of bytes transferred between the hardware and software compared to the baseline. Figure 3.8 and Table 3.3 show the multiplicative decrease in execution time compared to the baseline.

The relative number of transfers, shown in Figure 3.6 and Table 3.1, demonstrate the reductions in calls to transfer data from scheduling the data transfers. The average number of calls to transfer data was reduced to only 23% of the baseline benchmarks. Particlefilter saw the smallest reduction in calls to transfer data due to multiple variables used in both the hardware and software

58
in the main loop of the benchmark, resulting in data transfers occurring every iteration of the loop. Conversely, *nw* sees the largest reduction because all of the data transfers are scheduled outside of the loop encompassing the kernels.

![Relative Number of Data Transfers](image)

**Figure 3.6: Relative number of data transfers compared to baseline**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline(Transfers)</th>
<th>Scheduled(Transfers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>204</td>
<td>31</td>
</tr>
<tr>
<td>hotplate.1024</td>
<td>18,060</td>
<td>3,610</td>
</tr>
<tr>
<td>hotplate.2048</td>
<td>18,060</td>
<td>3,610</td>
</tr>
<tr>
<td>hotplate.4096</td>
<td>18,060</td>
<td>3,610</td>
</tr>
<tr>
<td>nw.2048</td>
<td>765</td>
<td>3</td>
</tr>
<tr>
<td>nw.4096</td>
<td>1533</td>
<td>3</td>
</tr>
<tr>
<td>particlefilter</td>
<td>709</td>
<td>317</td>
</tr>
<tr>
<td>srad</td>
<td>240</td>
<td>29</td>
</tr>
</tbody>
</table>

**Table 3.1: Number of data transfers per benchmark**

The reduction in bytes transferred between the hardware and software, shown in Figure 3.7 and Table 3.2, demonstrates the importance of scheduling larger data transfers. Though *hotplate* saw a similar reduction in calls to transfer data compared to other benchmarks, it sees a much larger reduction in bytes transferred than the other benchmarks. This is because the large
data transfers containing the data arrays were optimized away, leaving behind simple data transfers with single elements.

The improved execution time, shown in Figure 3.8 and Table 3.3, ranges from 1.09 times faster on \textit{bfs} to 53.3 times faster on \textit{nw}. Additionally, even the three benchmarks which see the least reduction in execution time (\textit{bfs}, \textit{particlefilter}, and \textit{srad}) still achieve an average speedup of 1.48 times faster than their baseline tests. The reduction to execution time was largely dependant upon being able to schedule the all large data transfers outside of encompassing loops, which was not possible for these three. \textit{hotplate} and \textit{nw} both see substantial reductions in bytes transferred or calls to transfer data, resulting in the largest reductions in execution times.

\begin{table}[ht]
\centering
\begin{tabular}{|l|l|l|}
\hline
Benchmark & Baseline(Bytes) & Scheduled(Bytes) \\
\hline
\textit{bfs} & 611,998,584 & 42,999,904 \\
\textit{hotplate.1024} & 60,607,707,240 & 14,440 \\
\textit{hotplate.2048} & 242,430,785,640 & 14,440 \\
\textit{hotplate.4096} & 969,723,099,240 & 14,440 \\
\textit{nw.2048} & 12,847,107,060 & 50,380,812 \\
\textit{nw.4096} & 102,928,127,988 & 20,1424,908 \\
\textit{particlefilter} & 3,088,436,152 & 374,853,272 \\
\textit{srad} & 48,322,314,240 & 6,711,148,544 \\
\hline
\end{tabular}
\caption{Number of bytes transferred per benchmark}
\end{table}
In addition, the results of manually performing data-flow and control-flow analyses to determine the optimal scheduling locations was nearly identical to the automatically scheduled data transfers, and the effects on execution time were identical.

Though it is unlikely for users to schedule data transfers similar to the baseline for these benchmarks, immediately before and after the kernel executes, it is quite possible they would if the kernels and loops were spread out throughout the call graph. Unless they manually performed interprocedural data and control-flow analyses, it is likely the placement of data transfers would be somewhere between the baseline, and the optimized scheduling.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline(s)</th>
<th>Scheduled(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>1.53</td>
<td>1.41</td>
</tr>
<tr>
<td>hotplate.1024</td>
<td>9.58</td>
<td>0.60</td>
</tr>
<tr>
<td>hotplate.2048</td>
<td>29.63</td>
<td>1.90</td>
</tr>
<tr>
<td>hotplate.4096</td>
<td>112.54</td>
<td>7.26</td>
</tr>
<tr>
<td>nw.2048</td>
<td>1.59</td>
<td>0.09</td>
</tr>
<tr>
<td>nw.4096</td>
<td>11.82</td>
<td>0.22</td>
</tr>
<tr>
<td>particlefilter</td>
<td>30.21</td>
<td>16.35</td>
</tr>
<tr>
<td>srad</td>
<td>15.67</td>
<td>10.48</td>
</tr>
</tbody>
</table>

Table 3.3: Execution times of benchmarks
3.7 Chapter Summary

This chapter describes the importance of scheduling data transfers, the difficulty of doing so, and provides techniques for compilers to automatically schedule data transfers in more efficient locations. The placement of data transfers can have a large effect on the performance of the program. The overhead from poorly placed data transfers can negate the benefits of using an accelerator.

Finding efficient locations to place data transfers can be simple for kernels that execute once in sequential code, but for those in loops, function calls, and control-flow, it can become substantially more difficult. Under these situations, control-flow and data-flow analyses are necessary to determine where the data transfers can be placed. This may still be reasonably done manually for simple code structures, but when nested in loops, function calls, and control-flow, it can become very difficult for users to analyze manually.

These types of analyses are traditionally found within compilers, where whole program analysis is feasible. In this chapter techniques were presented to efficiently schedule data transfers. First, all of the memory location in the kernels are identified, and initial data transfers are determined. Next all conflicting memory accesses are found. After performing and analyzing the alias analysis a scheduling graph is created. This scheduling graph is a combination of control-flow, data-flow, and the call graph. This graph is then analyzed using a modified liveness analysis to determine the window in which the data transfers may be scheduled. After this the data transfers are scheduled using dominator tree analysis to ensure the data transfers are placed in efficient locations. Markers are inserted at these locations to indicate to the backend where data should be transferred.

The impact of the techniques presented was demonstrated in the results. It was shown that reducing the number of data transfers, or the number of bytes transferred, both had a substantial impact on performance. Even the three benchmarks with that saw the least improvement still saw a 1.48x reduction in execution time.
CHAPTER 4. ON-CHIP DEBUG OF HLS-ACCELERATED PROGRAMS

With the increasing use of high-level synthesis (HLS) tools, a greater emphasis has been placed on system-level design tools. These tools can generate complex programs that run on both the hardware and software. The software is run on a traditional processor, and select portions of code can be automatically generated into hardware designs to execute on the FPGA. Additionally, these tools handle the interfacing between the two. For example, Altera OpenCL SDK and Xilinx’s Vitis tools both use HLS to accelerate OpenCL kernels on the FPGA fabric. The user partitions their software into code to run on a conventional processor (host code), and kernel code which is generated into hardware designs via HLS. Likewise for embedded solutions, Xilinx Vitis and LegUp HLS allow the user to partition their code into software to run on an embedded ARM processor and FPGA shared memory system to accelerate programs.

While these tools make it relatively straightforward to construct complex designs for heterogeneous systems, it can be very challenging for users to debug the generated designs. Users can take advantage of common debugging techniques available at the source code level, or even use software and hardware simulation to debug the designs. However, debugging the generated hardware designs can be much more difficult. Though the user may be familiar with the source code, once that is translated into a hardware design the entire code structure, layout, and flow of the program may have changed. Users often have to study the generated design for a significant amount of time before they can understand it enough to debug it.

Unfortunately, some bugs do not manifest under software and simulation debugging techniques. Bugs that involve parallelism, I/O, or that manifest after extended runtimes often require on-chip debugging techniques. One of these on-chip debugging techniques is trace-based debugging, described in Section 2.3.2. Though the techniques for trace based debugging have improved over the years, most of them have been focused around recording more data with less overhead [17], [39], [41]–[43].
Applying trace-based debugging to HLS-generated designs is substantially more difficult due to the user’s unfamiliarity with the generated designs. Though the user wrote the source code algorithm for the HLS tool, the generated design is generally different quite different. This is due to the nature of hardware designs versus the code structure of software programs. Software is generally written to execute sequentially. Hardware designs on the other hand are highly parallel, and execute as state machines and data signals. Fortunately, recent work [12], [13], [15] has sought to make trace-based debugging of HLS-generated systems easier by allowing users the ability to view trace data in context of the source code. Some of these works [13], [15] took advantage of the open-source nature of the HLS tool LegUp [55] to modify the source code and maintain a correlation between the source code and the generated hardware design. Using this correlation, they are able to map the trace data back to source-code variables, allowing users to step through the trace data in context of the source code. Though these works make great strides towards simplifying the demands on the user, they only focused on debugging hardware in isolation.

As mentioned previously, many HLS tools now support generating complex designs for heterogeneous systems. Means for on-chip debugging of these systems are still developing. To address this, this chapter puts forth the techniques presented in "Unified On-Chip Software and Hardware Debug for HLS-Accelerated Programs" by Matthew Ashcraft and Jeffrey Goeders [2] to provide trace-based debugging for heterogeneous systems. These techniques focus on recording software data to software traces, hardware data to hardware traces, and the interactions between

Figure 4.1: Overview of unified hybrid HLS debugging.
the hardware and software, as shown in Figure 4.1. To determine the impact of these techniques, the cost of adding them to program execution has been measured, and will be further discussed.

After discussing the techniques for capturing software trace data in addition to hardware trace data, as well as the cost on program performance, this chapter will put forth techniques presented in "Synchronizing On-Chip Software and Hardware Traces for HLS-Accelerated Programs" by Matthew Ashcraft and Jeffrey Goeders [3] to synchronize the hardware and software traces. Synchronizing the traces is important to understand the effects hardware and software have on each other. It allows users to follow trace data back and forth between the domains. Without it, it may become infeasible for users to follow trace data back to the origin of the bug. In addition to discussing synchronization, and putting forth techniques to synchronize the traces, this chapter will discuss the cost of adding synchronization to HLS-accelerated programs on heterogeneous systems.

### 4.1 Objective and Metrics

As mentioned in Section 2.3.3, previous work have demonstrated how to maintain a correlation between the source code and HLS-generated designs, allowing users to view trace data in context of the source code. The main objective of this chapter is to extend that debugging ability
to HLS designs that are part of larger software systems in a shared-memory environment, as is becoming more commonplace in FPGA SoC designs.

Like previous work, the goals of this work are to capture traces of the design execution, enabling the designer to debug the design post-execution by providing an execution trace that users can step through, similar to software debuggers. While previous works focused only on collecting an execution trace of the HLS hardware, this work focuses on the challenges introduced in collecting execution traces of both hardware and software domains in a shared-memory system, and determining a temporal correlation between the two.

Previous work focused on measuring the resource overhead introduced by inserting trace debugging circuitry into the HLS hardware. As our work mostly re-uses past work debugging circuitry, our results will focus instead on the cost of recording trace data in the software domain, measuring the impact this has on execution time.

In addition to recording software execution, the other major objective of this chapter is to develop a temporal correlation between the hardware and software traces, or as termed by this work, synchronize the hardware and software traces. This synchronization is necessary to allow the designer to follow debug data across domain bounds. Without the ability to follow the effects of bugs back to their origin, which can require crossing between hardware and software domain bounds, on-chip debugging becomes much more difficult. However, synchronizing the hardware and software traces will affect execution time, due to extra software instructions, and the amount of data maintained in each of the hardware and software trace buffers, due to synchronization information recorded to the trace buffers. We will look into the quantitative results, and ways to reduce this impact through various synchronization schemes.

4.2 Context

The techniques described in this chapter target a hybrid software/HLS hardware system where the software and HLS hardware operate in a shared-memory space, as is commonplace in modern SoC FPGA systems. This SoC-based platform is targeted by modern HLS tools, such as Xilinx’s SDSoC (now named Vitis) flow. Since we require modifications to the HLS tool to prototype our techniques, we elect to prototype our techniques using the hybrid flow of the open-source HLS tool, LegUp [55]. The hybrid flow of LegUp takes a C source code language file, as well as a
configuration file specifying which functions should be compiled into HLS accelerators to execute on the FPGA fabric. The hybrid flow of LegUp then partitions the source code between functions mapped to software and functions mapped to hardware. The software functions to execute on an ARM processor of an SoC FPGA, and the hardware functions run through the HLS flow to generate a digital hardware design for the FPGA fabric, as shown in Figure 4.2. The generated program is a cache-coherent shared memory system, where the HLS accelerators are AXI-master devices, allowing them to have access to the main memory (DDR) of the ARM processor. When HLS-accelerated code access global variables and structures, the hardware is automatically synthesized with the appropriate logic to make the necessary read and write requests over the AXI bus. The granularity of these data transfers is dependent on the benchmarks, and varies from non-pointer variables, to multiple entries in an array, to entire data arrays. LegUp, and the hybrid flow we utilize are further described in Section 4.4.

While this shared memory SoC does reflect a specific type of system that is supported by certain commercial HLS tools, such as Xilinx’s SDSoC, there exist many other popular system constructs, such as PCIe-based systems commonly targeted by OpenCL-based flows, as well as streaming-based systems. This is a rapidly changing landscape, and it would be beyond the scope of this work to demonstrate a debugging flow for each of the platforms. However, the techniques and concepts demonstrated in this chapter are extensible to these other types of platforms, likely with substantial modifications. For example, some of the techniques presented in this chapter are used in Chapter 5 to provide a source-code level debugging environment for on-chip debugging of OpenCL FPGA kernels.

4.3 Background

There are multiple factors involved in providing techniques for trace-based debugging of complex designs on heterogeneous systems, including those similar to the architecture shown in Figure 4.2. In the hardware, this includes the ability to maintain a correlation between the source code and the generated design, record variables during execution to a trace buffer, extract the trace data post-execution, and correlate that data with the source code. The techniques necessary to support these factors in the hardware were provided in recent work [15]. This work, as well as techniques to record software data during execution, were discussed in Section 2.3.
4.3.1 Debug Scenarios for HLS-Accelerated Programs

As hybrid HLS systems, or HLS-accelerated programs on heterogeneous systems, are still an emerging technology, and the user base is relatively small, it is difficult to find documented cases where on-chip debugging techniques were necessary on heterogeneous systems. Below are described hypothetical scenarios where on-chip debugging techniques for HLS-Accelerated programs would be helpful to designers:

Case 1 The HLS-generated program for a heterogeneous system splits computation between hardware and software, such as the partitioning technique discussed in [56]. In this case the hardware and software are executing in parallel. The effects of a bug cross domain bounds, making it difficult to determine its cause. Following the bug back to its origin will likely require the observation of both hardware and software domains, and a temporal correlation between the two.

Case 2 The HLS-generated program treats the FPGA as a bump-in-the-wire between the network and processor, such as in the Microsoft datacenter architecture [57]. 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. Synchronization between the traces might be needed to follow the trace data between domains.

Case 3 The HLS-generated program operates in a producer-consumer relationship, with the software feeding work to multiple hardware accelerators. The hardware produces an invalid output, and it is not clear if it is a bug in the hardware, or whether a malformed or unexpected data was sent to it by the software. Using trace-based debugging and synchronizing the traces, the user is able to follow the effects of the bug back across domain bounds to its origin in the software.

Case 4 The HLS-generated program splits up the main computation of its algorithm between the hardware and software. This computation repeats iteratively until a threshold is met. The hardware and software are not executing in parallel, but they share large amounts of data through shared objects. During execution a bug arises, and its effects propagate between
both domains. Through analyzing the synchronized traces, the user is able to follow the
effects of the bug back-and-forth between domains to the origin of the bug.

In each of these cases, trace-based debugging environment for complex designs on het-
erogeneous systems is needed. This environment involves observing each of the software and
hardware domains individually, and combining that information through synchronization to form
a view of the system as a whole.

4.4 LegUp HLS Tool

LegUp is an open-source HLS-tool developed by the University of Toronto [55], built as a
back-end to the LLVM compiler. As an HLS tool, it focuses on generating hardware designs at a
function level, and has supported HLS-accelerated programs from the start. An important feature
of the LegUp HLS-tool is its open-source nature. This makes it possible for researchers to modify
the source code of the tool to prototype techniques they have developed. As the techniques put
forth and prototyped in Chapter 4 are built into the LegUp HLS-tool, we will explain how it works
in greater depth.

4.4.1 Overview of Unified On-Chip Debugging in the LegUp Hybrid Flow

Our hybrid design flow for on-chip debugging of hardware in LegUp is shown in Figure
4.3. First the source code is read in and optimized by the front-end of the LLVM compiler, and
translated into the LLVM (IR). This IR is then split up into two different modules, or programs,
one for each the hardware and software. The functions moved to hardware are specified by the
user through a configuration file, and the remaining are left as software. Each of the modules is
treated as separate program from this point onward.

The hardware is transformed into a hardware circuit. In addition generating a hardware
circuit, debugging circuitry is inserted into the circuit to allow for on-chip debugging. This circuitry
was developed by Goeders et.al. [15] to capture datapath and finite state machine (FSM) signals,
which represent data-flow and control-flow in the source code respectively. Additionally, they
maintain a correlation between the source-code, IR, and the generated hardware circuit. This
correlation allows them to present the trace data to the user in context of the source code.
Figure 4.3: Design flow for unified on-chip debugging of HLS-Accelerated programs
The software IR is modified to handle communications between the software and hardware, and according to the techniques proposed later in this chapter, to insert a software trace buffer, as well as instructions to record data to the software trace buffer. It is then run through the ARM backend that generates ARM executable code.

Post-execution, a debugging system retrieves the hardware and software execution traces from the SoC. Using the correlation maintained between the source-code and generated code, the data is presented to the user in a unified debugging environment, similar to Figure 4.1.

4.5 Trace-based Debugging of HLS-Accelerated Programs

This section describes the proposed architecture for enabling unified hardware and software trace-based debug of HLS-Accelerated Programs.

4.5.1 Capturing Hardware Execution

To capture execution of the HLS-generated design, we make use of the existing modifications to the hardware-only version of the HLS flow in LegUp developed by Goeders et al. [58]. These modifications automatically insert a trace buffer using on-chip FPGA memories during the generation of the RTL circuit. Traces of the finite state machine (FSMs) are recorded to the trace buffer to reconstruct control-flow in context of the source code. Additionally, datapath signals are recorded to the trace buffer for the corresponding source code variables being recorded. Together the FSM and datapath signals can be used to reconstruct the control-flow and data-flow of the software generated into hardware designs.

4.5.2 Capturing Software Execution

Recording software execution is typically done through inserting extra operations into the program to store data of interest throughout program execution. While there are several stages during the compilation flow where recording instructions can be inserted, we elect to insert recording instructions at the compiler IR level, similar to other works [19], as this is where we can also take advantage of the compiler analyses.
Overall, recording software execution is broken up into multiple steps, which will be addressed in this section. First the user selects what to record through source code annotations. These annotations can be applied to specific variables, functions, or to the program as a whole, providing the user with greater control over what to record with less effort. These annotations are identified in the IR and recording instructions are auto-inserted into the IR according to the annotations. The recording instructions record the specified data and globally unique identifiers to the software trace buffer. The globally unique identifiers are also recorded to an SQL database to aid in correlating the trace data with the source code post-execution. The software trace buffer is a globally allocated array automatically inserted into the software IR, and is treated as a circular buffer. The program then executes and data is recorded to the software trace buffer. After execution, the trace buffer is retrieved, parsed, then presented to the user in a unified debugging environment. Each of these steps will now further be addressed.

Selecting What to Record

Though in hardware we can tap into signals and feed them into a trace buffer with minimal impact to circuit performance, this is not possible in the software. Recording data to the software trace buffer requires additional software instructions, and will likely reduce software performance. While it would be nice to record all control-flow and data-flow in the program, the impact of doing so is likely unacceptable. The impact of doing so will be demonstrated in Section 4.5.5.

To reduce the impact to program performance, we introduce mechanisms to allow the user to select which parts of the software program they would like to record, similar to how users select signals to record in the hardware. It is unlikely the user will have a solid understanding of the LLVM IR, so we elected to support C source code annotations. Annotating at the source level, and automatically modifying at the IR level, we can take advantage of the simplicity of software code, while still leveraging the analyses and optimizations of the compiler.

We allow users to enable recording of the following elements:

1. Control-Flow: Annotations can be placed on any function declaration or definition to enable the recording of control-flow, as shown in Listing 4.1. This listing contains pseudo-code representing the source-code annotations, and Listing 4.2 represents the resultant auto-inserted
# record control_flow

```c
Void foo(...) {
  if (...) {
...
  } else {
...
}
}
```

Listing 4.1: Source code annotations to record control-flow

```c
if (...) {
  recordControlFlow(201);
} else {
  recordControlFlow(202);
}
```

Listing 4.2: Auto-inserted recording instructions in the IR to record control-flow

```c
# record all store instructions

void foo(...) {
  X[index] = y;
  if (...) {
    M[index] = z;
  } else {
    N[index] = k;
  }
}
```

Listing 4.3: Source code annotations to record variables

```c
X[index] = y;
recordValue(X[index], 101);
if (...) {
  M[index] = z;
  recordValue(M[index], 102);
} else {
  N[index] = k;
  recordValue(N[index], 103);
}
```

Listing 4.4: Auto-inserted recording instructions in the IR to record variables
instructions in the IR, recordControlFlow. The argument for recordControlFlow is a globally unique ID representing which branch was taken. This will further be explained later. The user may further specify the capture of all control-flow in the software program, or just the control-flow of the annotated function.

2. **Variables:** The user can select a variable to record in the program by placing an annotation on the variable declaration. The annotations allow for all load and/or stores to/from that variable to be recorded. Additionally, LegUp’s alias analysis is used to identify and record aliasing Variables. The user may also place annotations on a given function to record all loads or stores in that function or the entire software program, as shown in Listing 4.3. The effect of this is shown in Listing 4.4, where the instruction recordValue is inserted after each store instruction. recordValue records the value in the stored location, as well as a globally unique identifier representing the data recorded.

3. **Function Arguments:** Function annotations play a special role because functions can act as transitions between sections of the program, such as functions that start FPGA execution or functions that transfer data. Because of this, the function annotations more robust. Function annotations include recording all arguments, individual arguments, inputs, outputs, values
accessed through pointers, and/or multiple elements of an array. Listing 4.5 provides example annotations on the function `transferToFPGA`, a function which transfers data from the software to the hardware. The resultant auto-inserted code in the IR is represented in Listing 4.6.

An example of how annotations might be applied in the LLVM IR is shown in Figure 4.4. In this control-flow and data-flow graph example, the user has selected to record both store instructions on X, and control-flow throughout the function. After each of the store instructions to X, instructions are inserted to record the value of X, and a globally unique ID that represents the data recorded. This ID is later used to correlated the recorded data with source-code variables. In each of the control-flow branches instructions are inserted to record which branch was taken, with globally unique IDs representing each branch.

The actual implementation of the recording functions is shown in Listing 4.7, with comments to explain each instruction. First the index into the trace buffer is retrieved from a global variable, and is masked to ensure it stays within array bounds as a circular buffer. Then the pointer address is calculated, the trace data is recorded to the trace buffer, and then the index is incre-


Listing 4.7: LLVM IR Recording Function

mented. Similar operations then occur for the ID. Lastly the incremented index is stored back in the global variable.

Capturing Data to a Trace Array

Software execution trace data is stored in a large allocated memory region, which is automatically inserted into the program as a globally declared array. Through a directive, the user can choose the size of this array. Naturally, the larger the array size, the longer the execution trace can be captured; however, its size is limited by main memory.
Figure 4.5 shows a sample 32-bit wide trace array, containing five different types of data recorded into the array (32-bit wide array was the native machine size for the prototype system, but this could be changed). Each entry in the trace array contains an ID field. This ID field contains a globally unique identifier that represents where the data was recorded in the program. The first entry in Figure 4.5 ($ID_c$ at index 0) captures control flow information only, indicating that the program flow has reached the end of a branch.

The remaining entries in Figure 4.5 ($ID_v$) represent variable values that are captured in the trace. As shown in the figure, 8 bit and 16 bit values are shifted and packed with the ID field, reducing the number of writes to the trace array. 32 bit values require a separate entry from the ID, and 64 bit values require two additional entries. Each time an entry is added to the trace array, a global pointer is incremented to point to the next entry. If this entry has reached the bounds of the array, then it is reset to zero.

A SQL database is maintained that contains information about the program such as functions, variables, instructions, etc. The database contains information on each ID value, indicating the types and size of data recorded, and how the data relates to the program elements.
After execution, the software trace array is extracted from the shared memory, as well as the global pointer representing the last location stored to. Beginning at that location the data is read backwards, ID first, followed by data. The ID is always inserted after the data so that it can be read first, and the size and layout of the corresponding data can be queried from the database. Data continues to be read and reassembled until an empty element of the array is reached ($ID = 0$), indicating an unfilled portion of the buffer, or until the index wraps back around to the starting location.

**Optimizing Control-Flow Capture**

A naive approach to recording control flow would be to add an entry to the trace array for each software instruction that was executed. However, this would have very high overhead. Since the basic blocks have only a single entry and exit point, it is sufficient to record only a trace of the basic blocks. This can be further optimized by the fact that if a basic block has only one predecessor, we can determine control flow must have transitioned from that basic block without having to record it. Thus, we only need to add entries to the trace array at converging branches as shown in Figure 4.6. This allows one to reconstruct the entire control-flow when reading backwards through the trace data. Table 4.1 indicates the number of basic blocks present in each of our benchmarks, as well as the number of blocks that we must record using this optimization. The benchmarks are from CHStone LegUp, and are later discussed in Section 4.5.5. On average, we have to add a control flow ID into the trace array in 71% of basic blocks.

![Figure 4.6: Recording control-flow immediately prior to joining branches](image-url)
Though not implemented in our techniques, another possible optimization is to remove instructions to record control-flow in branches that record variable or function operands. This optimization has been demonstrated by Pinilla et al. [16].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Basic Blocks</th>
<th># Converging Blocks</th>
<th>% Traced</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>35</td>
<td>17</td>
<td>49%</td>
</tr>
<tr>
<td>aes</td>
<td>101</td>
<td>84</td>
<td>83%</td>
</tr>
<tr>
<td>blowfish</td>
<td>45</td>
<td>34</td>
<td>76%</td>
</tr>
<tr>
<td>dfadd</td>
<td>103</td>
<td>68</td>
<td>66%</td>
</tr>
<tr>
<td>dfsin</td>
<td>196</td>
<td>132</td>
<td>67%</td>
</tr>
<tr>
<td>gsm</td>
<td>129</td>
<td>103</td>
<td>80%</td>
</tr>
<tr>
<td>alphablend</td>
<td>15</td>
<td>10</td>
<td>67%</td>
</tr>
<tr>
<td>blackholes_LUT</td>
<td>84</td>
<td>56</td>
<td>67%</td>
</tr>
<tr>
<td>divstore</td>
<td>11</td>
<td>6</td>
<td>55%</td>
</tr>
<tr>
<td>los</td>
<td>57</td>
<td>44</td>
<td>77%</td>
</tr>
<tr>
<td>los2</td>
<td>57</td>
<td>44</td>
<td>77%</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>23</td>
<td>18</td>
<td>78%</td>
</tr>
<tr>
<td>primestore</td>
<td>26</td>
<td>20</td>
<td>77%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>68</strong></td>
<td><strong>49</strong></td>
<td><strong>71%</strong></td>
</tr>
</tbody>
</table>

### 4.5.3 Triggering Data Capture

As the software and hardware trace buffers both have limited capacity, trace-based debugging generally supports triggers, which indicate when to stop recording data to the trace buffer. The typical approach in the hardware domain is to continually write data to the trace buffer in a circular fashion, and halt capturing at some point in the execution. In past HLS debug work [15], breakpoints/trigger-points located in the source code have been used to indicate when capture should halt.

When triggering a breakpoint in complex program for heterogeneous systems debugging, it is essential that both domains halt recording into the trace buffers. If a hardware breakpoint is triggered, but the software does not stop, then the buffer might wrap around and overwrite data important to the user.
To support halting data capture in the software when a hardware breakpoint is triggered, a memory-mapped register accessible to both hardware and software is used to indicate a breakpoint has been encountered. This single-bit register is checked before each time the software writes to the trace array. If the register is set low then the hardware has triggered a breakpoint, and the software skips over the recording functions. Though the software continues to execute, the trace buffer is no longer written to, preserving the data at the time of the breakpoint. It would be possible to create an interrupt-driven system rather than the polling technique we employed. While this would likely reduce the overhead, it would increase the system complexity, and may be problematic if the user code already has their own interrupt handlers.

Though this allows for hardware breakpoints to prevent the software from recording data to the trace buffer, we currently have no way of doing the reverse and halt the hardware from the software, or more specifically we have no way of halting the software in general. Halting the software would be dependent on the processor architecture, as it would require some form of breakpoints in the CPU, and possibly OS-level support.

4.5.4 GUI

As a proof-of-concept, we modified the open-source LegUp Debugging GUI from Goeders et al. [58] to demonstrate a unified hardware and software debugging environment. The resultant debugging GUI is shown in Figure 4.7. This GUI provides the first unified debugging environment for on-chip debugging of heterogeneous systems, where the user can step through the hardware and software traces independently. Lines of code representing current trace data in the hardware are highlighted in green, and the line of code representing current trace data in the software are highlighted in cyan (not shown).

4.5.5 Impact of Trace-Based Debugging of Heterogeneous Systems

Supporting trace-based debugging of HLS-accelerated programs on heterogeneous systems mainly involves modifications to the software, resulting in additional overhead to execution time. This section will discuss the impact to execution time from multiple benchmarks, and the methodology used to obtain the measurements.
Benchmarks and Methodology

As mentioned in Section 4.1, our results are focused on the cost of recording trace data in the software domain, specifically the impact this has on execution time. To collect measurements, we use the Terasic Cyclone V DE1-SoC board, which is an SoC with a CPU and FPGA, communicating through shared memory and over the AXI bus. The software execution times were obtained by adding a high-resolution hardware timer to the system, and interfacing with it from the tested software code. Additionally, any communication with the debugging system, such as print statements, were moved outside of the timed code region.
Recording trace data on the hardware and software results in additional overheads. The hardware requires additional resources to record data, but can do so with no overhead to execution time. Conversely, all recorded data on the software results in added overhead to the software execution time.

To see the effects on execution time we have tested 13 benchmarks, 6 from CHStone [59], and 7 pthread-based benchmarks in the LegUp HLs tool that are designed for parallel hardware-software execution. Five of the pthread benchmarks are from [60]; blackscholes, los, los2, divstore, and mandelbrot. The other pthread benchmarks are found in LegUp’s benchmarks (alphablend and primestore). alphablend alphasblend two images and primestore computes the amount of prime numbers under a specified limit.

Note, we did not explore the memory overhead, as that is set by the size of the software trace buffer. Compared to the hardware, the software trace buffer can be as large as the user desires, given it fits in main memory.

Results and Analysis

To determine the impact of supporting trace-based debug of HLS-accelerated programs on heterogeneous systems, we recorded execution times under two scenarios: 1. the greatest or worst possible overhead to execution time from recording all control-flow, loads, and stores in the entire program, and 2. the cost of recording only data transfers between the hardware and software, which is done through annotating the data transfer function call.

The worst case scenario shows the bounds of impact on overhead by greatly increasing the sample size. This was measured by moving all of the computations to the software, leaving only enough in the kernels to ensure the program still acted as an HLS-accelerated program. With almost all of the code residing in the software, there was many more load, store, and control-flow operations than normal, and we were provided with a better sampling size. With this code structure, we recorded the execution time under two different sets of annotations. First recording all control-flow in the program. Second recording all control-flow, loads, and stores in the program. By annotating the programs to record both of these, the impact of recording load and stores versus control-flow can be seen.
The impact under the worse case scenario is shown in Figure 4.8. When recording control-flow, the average execution time increased by 50%, varying from 5% on adpcm to 207% on los. These results are dependent upon the complexity of the control-flow within the program, particularly the control-flow within nested loop structures. An interesting result is the increased execution times of adpcm, aes, los, and los2. It can be seen in Table 4.1, that adpcm has the least amount of convergence at 49%, and aes has the highest amount of convergence at 83%, yet their average increases to execution time are among the least of all benchmarks. However, los and los2 have closer to average amounts of convergence, yet are outliers in terms of increased execution time. This appears to be due to the amount of convergence in their main algorithms. The main algorithms for los and los2 are loops containing nested loops and many if statements. Conversely, adpcm and aes have simpler loop structures. If the results from los and los2 were removed from the results, the average increased execution time would drop to 26%.

When recording all loads, stores, and control-flow in the program, the average increase execution time was 97%. These execution times varied from 15% on mandelbrot to 220% on los. From what we identified analyzing the results, the impact to execution time in our tests is largely dependant on the complexity of the main algorithm in the program, which normally would have executed on the hardware. The benchmarks with loops containing complex control-flow, such as los, resulted in a large amount of control-flow data recorded to the software trace buffer, which in turn greatly affected execution time. Other benchmarks, such as adpcm, contained simple control-flow, but contained many load and store operations in the main loop of the benchmark, likewise resulting in large increases to execution time. With most of this control-flow, loads, and stores moved back to the hardware, the impact to execution time will substantially lessen.

Recording data transfers is interesting because explicit data transfers represent junctions in the program where computation moves between hardware and software. Explicit data transfers are done through API calls that transfer specific data between the hardware and software. This is opposed to shared objects, which can be modified by both the hardware and software. Dealing with shared objects through synchronization will be discuss later in this chapter. To measure the impact of recording data transfers, we restructured the code to split the computation between hardware and software, necessitating data transfers between the devices. For the CHStone benchmarks, the default annotated computations were treated as a kernel and generated into hardware designs.
For the pthread benchmarks, the kernel threads were split evenly between hardware kernels and software threads. Note, the CHStone benchmarks halt software execution while the hardware is executing, whereas the pthread benchmarks execute concurrently.

The impact of recording data transfers is shown in Figure 4.9. The average increase to execution time was 0.29%, and ranged from 0.0024% in mandelbrot to 2.09% in dfadd. However, only dfadd increased execution time by more than 0.52%. This is likely due to dfadd’s short execution time, which was three times faster than the second shortest execution time. Due to the short execution time, the duration of the data transfers takes up a larger percentage of the overall execution time. Generally the impact to execution time for recording transfers is minimal. This is shown by the CHStone benchmarks blowfish and gsm, which transferred and recorded the largest amounts of data, yet still see a minimal impact to execution time.

4.6 Synchronizing Hardware and Software Traces

Though the techniques presented thus far in this chapter allow for hardware and software traces to be recorded during execution, and provide means to view these traces post-execution,
they are lacking in one important area, that of synchronization. Synchronization is a means of aligning the hardware and software traces, allowing users to follow data back and forth between the domains.

Without a means of synchronization, or aligning the trace buffers, following trace data across domains can become extremely difficult if not infeasible. An example of this is shown with the shared object X in Figure 4.10. In this example, shared object X is written to from two functions in the hardware kernels, and is read at some point in the software code. In this example, either of the assignments to X in hardware should be valid and there are no race conditions. If the user identifies a bug in their program, and they follow the trace data back to the use of X in software, then how do they know which assignment to X in hardware was responsible. It might be possible to figure out which is responsible in some cases, but in other cases it would likely be infeasible to do so without a means of synchronizing the traces.
4.6.1 When to Synchronize

Determining when synchronization is needed is important in reducing the overhead synchronization has on the program. It would be nice if every instruction was synchronized, and the user could step through both traces at the same time, always aware of what is happening in both domains. However, this is unfeasible for multiple reasons. One of these reasons is the substantial difference in clock frequencies between both domains. In addition, the software executes sequentially, whereas the hardware executes many instructions in parallel. Another main reason is the overhead to adding synchronization to the systems, including increased execution time, and resources in both the hardware and software. Synchronization adds extra instructions to the software, as well as memory to record the synchronization information. In the hardware, additional resources are needed to identify states that require synchronization, communicate with the software, and to record synchronization to the hardware trace buffer.

Note that synchronizing the trace buffers is not the same as synchronizing threads. Synchronizing threads is generally done when there are data dependencies between them. Because of this, thread synchronization often involves locking mechanisms to ensure validity of the synchronized data. However, synchronizing the trace buffers, as later described in this chapter, is done in hardware, and can handle multiple requests at once.

Through analyzing where synchronization may be needed, we determined that as long as the interactions between hardware and software are synchronized, then the traces will be synchronized. To further elaborate, besides explicit data transfers which are easier to follow between
domains, the only other way of data crossing domains is through shared objects. Thus if the order of memory operations on shared objects is maintained, then the user can follow the operations on these objects back and forth between domains, providing synchronization between the traces. To support this, we have developed a synchronization technique based on unique IDs representing the ordering of operations on shared objects, as well as multiple synchronization schemes in which the synchronization may be applied to reduce the overhead.

4.6.2 Synchronization Technique

The synchronization technique we have developed is based around IDs representing memory operations on shared objects. The IDs are globally unique variables generated from a counter, representing the order of writes to shared object. The synchronization technique follows two rules:

1. When writing to the memory location of a shared object, the synchronization ID is incremented, then recorded to the trace buffer of the device performing the write.

2. When reading from the memory location of a shared object, the current synchronization ID is recorded to the trace buffer of the device performing the read.

An example of this is shown in Figure 4.11. After each assignment to $X$ in hardware, the value of $X$ is recorded to the hardware trace buffer, followed by the incremented synchronization ID. After $Y$ is assigned the value of $X$ in the software, it is recorded to the software trace buffer followed by the current synchronization ID because $X$ is read. The results of this are shown in Figure 4.12. In the hardware, the incremented synchronization IDs are 1 and 2. In the software, the synchronization ID recorded after assigning the value of $X$ to $Y$ is 1, meaning the last operation on shared objects before the value of $X$ was read in the software was the first assignment to $X$ in the hardware, or the value returned from $foo$. Thus the value assigned to $Y$ came from $foo$. Note that our technique does not currently support adding timestamps to the trace buffer, but is a consideration for future work as mentioned in Section 6.2.2.

The implementation of this is slightly different in the hardware and software, as will further be discussed later in this chapter. In the hardware we synchronize on every FSM state containing memory operations on shared objects. In the software, we synchronize based on the synchronization schemes explained next in this section.
4.6.3 Synchronization Schemes

The synchronization technique allows for the user to understand the ordering of memory operations on shared objects, allowing them to follow the flow of data between the domains. However, there is some overhead incurred from synchronizing the trace buffers. To reduce this overhead, this section discusses three different synchronization schemes that can be applied to the software. Depending on the code structure, each of these schemes can still maintain 100% synchronization, or a complete ordering of memory operations on shared objects, while minimizing the impact on performance.
The code in Figure 4.13 will help to explain each of the synchronization schemes. The top box in the figure contains possible source code. The bottom box contains a pseudo-code representation of the IR code, and the corresponding loads and stores. The synchronization schemes are as follows:

1. **Scheme #1 - Memory Instructions:** The first synchronization scheme is the simplest and safest synchronization scheme, that always guarantees 100% synchronization. This scheme does exactly as the technique was previously described, inserting synchronization instructions after every memory operation on shared objects in the software. In the case of the example code in Figure 4.13, after each load instruction, the current synchronization ID would be recorded, and after the store instruction, the synchronization ID would be incremented, then recorded. Under situations where memory operations on shared objects are scattered throughout the program, then synchronizing on every memory operation on shared objects is reasonable.

2. **Scheme #2 - Basic Block of Memory Instructions:** When there are multiple memory operations on shared objects in a row, such as in the pseudo-code in Figure 4.13, and the user knows there are no race conditions or other memory operations happening in parallel, then synchronizing on every memory operation is overkill. Under these situations, the second synchronization scheme is a better approach. Under this scheme, all of the synchroniza-
tion instructions in each basic block are replaced with a single synchronization instruction at the end of the basic block. In the case of the pseudo-code, this would result in the synchronization instruction being inserted after the store instruction before the backedge of the loop. Further optimizations could be performed using complex program-wide analyses, but we are limited due to the alias analyses available in the LegUp HLS tool. Thus for further improvements, Scheme #3 relies on the support of the user.

3. **Scheme #3 - Direct Synchronization:** The third synchronization scheme is user inserted synchronization. If the user is able to manually perform data-flow and control-flow analyses, then they can insert synchronization into the program at the location that has the least impact on performance. In the case of the pseudo-code example in Figure 4.13, the user may determine that there are no other memory operations happening in either domain during the execution of this loop. As a result, the user could manually insert a synchronization instruction after the for loop. This would reduce the number of synchronization instructions from four per iteration of the loop under Scheme #1, or one per iteration of the loop for Scheme #2, to a single synchronization instruction after the loop completes.

Note these synchronization schemes only apply to the software. The hardware only has one way of synchronizing, on the states with memory operations on shared objects as will later be explained.

4.6.4 **Synchronization Implementation**

The synchronization techniques and schemes presented in this section have been implemented into LegUp on top of the support for trace-based debugging of HLS-Accelerated programs on heterogeneous systems. Synchronizing the software is done through modifications to the software IR code. Synchronizing the hardware is supported through modifying the trace debugging circuitry.

4.6.5 **Identifying Shared Objects**

As mentioned previously, the synchronization between the hardware and software traces is done through an ordering of memory operations on shared objects. Before ordering the memory
operations on shared objects, the shared objects must be determined. These shared objects are
global variables that have memory accesses in both the hardware and the software. Global variables
that only access the hardware or the software do not cross domain bounds, and do not need to be
synchronized.

To identify these objects, the LLVM IR code must be analyzed before the IR code is auto-
matically split between the hardware and software by LegUp, as shown in Figure 4.3. At that time
all load and store instructions for global variables are analyzed. All global variables with these
instructions in both the hardware and software regions of the code are determined to be shared
objects. Each of these global variables is added to a list of shared objects, and their memory op-
erations are added to lists of memory operations on shared objects in either the hardware or the
software. These lists are later used to determine when synchronization is needed in the hardware
and software.

4.6.6 Hardware Implementation

Unlike the software code, where only a single instruction can execute at a time, the HLS-
generated hardware design can execute multiple instructions in any given hardware FSM state.
As a result, supporting synchronization in hardware requires identifying which hardware states
contain the instructions with memory operations on shared objects from the source code.

Throughout the hybrid HLS-flow of the LegUp tool, a correlation is maintained between
the source code and the generated designs. This is necessary to support their debugging environ-
ment where the user can step through the trace data at the source code level. Using this correlation,
we identify the FSM states containing the source-code memory operations on shared objects previously identified. These states are then used to generate the synchronization ID module in hardware.

The purpose of the synchronization ID module is to maintain the synchronization ID, and
its interface is shown in Figure 4.14. The list of hardware states that perform memory operations
on shared objects is used by this module to determine when synchronization is needed. If the
current state (DUT State) performs a store on shared objects, then the synchronization ID is in-
cremented then recorded to the hardware trace buffer. Otherwise, the system checks if the current
state performs a load on shared objects, and if so records the current synchronization ID to the
When the hardware needs to record the synchronization ID to the hardware trace buffer, it sends the synchronization ID $\text{Sync ID}$, and sets the $\text{Record}$ signal to the LegUp debugging circuitry [15]. As both the synchronization ID and the regular trace data are likely to be added to the hardware trace buffer in the same cycle, we modified the debugging circuitry to record the trace data and synchronization IDs using dual ported memory. The first port records the regular trace data. The second port only records the synchronization ID when the $\text{Record}$ signal is set high.

As the regular trace data and the synchronization ID are recorded to the hardware trace buffer separately, we need a way to differentiate the two post-execution. To address this, an extra bit has been added to the trace-buffer width as shown in the Trace Circuit in Figure 4.14. This bit differentiates between regular trace data and synchronization IDs.

To support the software retrieving the synchronization ID, the synchronization ID module monitors the $\text{AXI Read}$ flag and the read address ($\text{Read Addr}$) on the AXI-slave connection interface integrated with LegUp. When the $\text{AXI Read}$ flag is set high, the module checks the $\text{Read Addr}$ to determine if the address represents a request from software for either the current or incremented synchronization ID. If so, the corresponding synchronization ID is sent over the AXI bus to the software.
Arguments: ID, Trace Data, Pointer to Hardware Address

```llvm
define void @record32Synchronize(i32 %0, i32 %1, i32* %2) {
  %syncID = load volatile i32* %2
  %GVIndex = load i32* @ArrayIndex
  %4 = and i32 %GVIndex, 65535
  %recordAddr = getelementptr [65536 x i32]*
      @_ZN5legup11recordArrayE, i32 0, i32 %4
  store i32 %syncID, i32* %recordAddr
  %5 = add i32 %GVIndex, 1
  %7 = and i32 %5, 65535
  %valAddr = getelementptr [65536 x i32]*
      @_ZN5legup11recordArrayE, i32 0, i32 %7
  store i32 %1, i32* %valAddr
  %8 = add i32 %5, 1
  %9 = and i32 %8, 65535
  %tagAddr = getelementptr [65536 x i32]*
      @_ZN5legup11recordArrayE, i32 0, i32 %9
  store i32 %0, i32* %tagAddr
  %10 = add i32 %8, 1
  store i32 %10, i32* @ArrayIndex
  ret void
}
```

Listing 4.8: LLVM IR Recording Function with Synchronization

### 4.6.7 Software Implementation

As the synchronization ID is managed by the hardware, synchronizing the software requires the software to retrieve the synchronization ID from hardware. This is done through a volatile load of the memory mapped AXI-slave interface. One address represents a `load` from a shared object, and the other represents a `store`. As will be noted later, because this volatile load crosses domain bounds, it has a substantial impact on software performance.
The first synchronization scheme is synchronizing after every memory operation on shared objects. To support this, modified recording function calls are inserted after each memory operation on shared objects. These modified recording functions retrieve and record the synchronization ID from the hardware addresses, record the value of the shared object, and record the ID for that shared object to the software trace buffer. An example of the modified recording functions is shown in Listing 4.8, similar to the original recording instruction shown and explained in Section 4.5.2. Only the code different from the original recording instruction is commented on.

The second synchronization scheme replaces all modified recording instructions with a single instruction at the end of the basic block. To do so, each modified recording instruction is replaced with a regular recording instruction from the previous section, and the basic block containing that instruction is added to a list. After replacing all of the modified recording instructions with regular instructions, a synchronization function call is inserted at the end of the basic block before the block terminator. This synchronization function call retrieves and records the synchronization ID to the software trace buffer.

The third synchronization scheme is user-inserted or direct synchronization, where the user manually analyzes the code to identify the most efficient place to synchronize the program, and inserts a function call into the source code to synchronize at that location. In the IR, this function call is identified, and replaced with a synchronization function call.

4.6.8 Impact of Synchronization

Adding synchronization to trace-based debugging of HLS-accelerated program on heterogeneous systems impacts the amount of data preserved in the hardware and software trace buffers, as well as execution time of the program. As each of the hardware and software trace buffers are circular buffers with limited memory, each time the synchronization ID is recorded, there is the risk of older debug data being overwritten. Additionally, each synchronization instruction in software impacts execution time. To determine the quantitative impact synchronization has on the program, we measured the impact to execution time, and the reduction in the percentage of all insertions to the trace buffer that can fit in the hardware and software trace buffers using the Terasic Cyclone V DE1-SoC board.
Benchmarks

Though HLS-accelerated programs are becoming increasingly used, benchmarks which replicate their structures and properties are more difficult to find. We ended up testing two benchmarks that show more of the extremes in terms of the impact of synchronizing. These two benchmarks are Back Propagation (backprop) and Speckle Reducing Anisotropic Diffusion (SRAD) from the Rodinia benchmark suite [52]. The Rodinia benchmark suite is of particular interest, as it was design for the purpose of testing heterogeneous systems. To view the greatest impact on these benchmarks, the explicit data transfers were replaced with global variable shared objects, forcing the system to synchronize more often.

The backprop benchmark is a machine learning benchmark focused on neural networks. It consists of a forward and a backwards phase. The forward phase computes weights. The backward phase measures the error and computes new input weights for the forward phase. Throughout this benchmark there are many memory operations on shared objects, most of which are contained in loops or nested loops. For our testing, the computation was split between the hardware and software, with the forward phase executing on the software, and the backwards phase executing on the hardware. To collect additional data regarding the impact on performance, we placed an outer loop around the phases that executes until the measured error of the backward phase reach a minimum threshold.

The SRAD benchmark is a partial differential equation based diffusion algorithm. It removes speckles from images without compromising important image features. This type of algorithm is commonly used in ultrasonic and radar images to improve image clarity. In our testing, the main compute of the partial differential equation was placed in hardware. On the software the image is read to compute standard deviations used by the hardware. Overall, most of the computation resides in the hardware, with relatively few operations in the software. Additionally, the whole algorithm for both the hardware and software is contained within an outer loop.

Results and Analysis

To observe the overall impact of recording synchronization IDs to the trace buffers, we measured the percentage of all trace data that could be maintained in the trace buffers. As more
synchronization ID are recorded to the trace buffers, the total number of insertions to the trace buffers increases, resulting in a smaller portion of the total execution trace remaining in the trace buffer what execution stops.

Additionally, the execution time is impacted due to additional instructions inserted into the software. These instructions retrieve the synchronization ID from the hardware, then record it to the software trace array. Note that each of these tests maintained 100% synchronization, allowing for a total ordering of memory operations on shared objects to be reconstructed post-execution.

<table>
<thead>
<tr>
<th></th>
<th>Trace Depth</th>
<th>HW Trace Insertions</th>
<th># Times Buffer Filled</th>
<th>Insertions in Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop Baseline</td>
<td>515</td>
<td>21,552</td>
<td>41.85</td>
<td>2.39%</td>
</tr>
<tr>
<td>backprop w/ Sync</td>
<td>515</td>
<td>32,089</td>
<td>62.31</td>
<td>1.60%</td>
</tr>
<tr>
<td>srad Baseline</td>
<td>1020</td>
<td>14,910</td>
<td>14.62</td>
<td>6.84%</td>
</tr>
<tr>
<td>srad w/ Sync</td>
<td>1020</td>
<td>20,758</td>
<td>20.35</td>
<td>4.91%</td>
</tr>
</tbody>
</table>

Table 4.2: Hardware Trace Buffer Insertions

**Impact on Hardware Trace Buffer Insertions**

The impact on hardware trace buffer insertions is shown in Table 4.2. The rows of the graph represent each of the benchmarks with and without synchronization. The second column represents the number of hardware trace buffer insertions. As the amount of memory allocated to the hardware trace buffer is fixed, the trace depth is dependant upon the trace buffer width. The greater the width, the smaller the trace depth. The third column represents the total number of hardware trace buffer insertions throughout program execution. The fourth column represents the number of times the trace buffer was filled and overflowed. The last column represents the percentage of the total execution trace that fit in the trace buffer.

Overall, there is a modest reduction in the percentage of the total execution traces that fit in the trace buffers. Adding synchronization increases the number of insertions to the hardware trace buffer from 21k to 32k for *backprop*, and 14k to 20k for *srad*. With this increase in trace buffers insertions, the buffers overflow more often, 41 to 62 times for *backprop*, and 14 to 20 times for *srad*. Additionally, the percentage of the total execution trace that fit in the trace buffer was
reduced from 2.39% to 1.60% for backprop, and 6.84% to 4.91% for srad. Overall, the impact on hardware trace buffer insertions is determined by the frequency of states that synchronize versus states which record debug data to the trace buffers at any given point in execution.

Impact on Software Trace Buffer Insertions

Similar to the hardware, there is a reduction in the percentage of total software trace buffer insertions that fit in the buffer due to synchronization. The measurements are shown in Figures 4.15 and 4.16. The graphs are broken up into four sections: No Sync, Scheme #1, Scheme #2, and Scheme #3. No Sync represents the same tests run from the previous section without synchronization. Scheme #1 represents synchronizing after every memory operation on shared objects (Sync). Scheme #2 represent synchronizing at the end of basic blocks with memory operations on shared objects (Sync BB). Scheme #3 represents user-inserted or direct synchronization (Direct Sync). For all of these tests, the user inserted synchronization was inserted at junctions in the programs where the system hands-off computation to the hardware or software. For backprop, the synchronization
was inserted between the forward and backward phases. For `srad`, the synchronization was inserted before and after the main compute on the hardware. Additionally, in the graphs the bars represent which data was recorded to the software trace buffer; all control-flow (`CF`), loads (`LD`), and stores (`ST`) in the program.

The impact on software trace buffer insertions for `backprop` and `srad` are shown in Figures 4.15 and 4.16. Note the Y axis for these graphs are different. Adding synchronization when capturing control-flow has a substantial impact for both Scheme #1 and Scheme #2 because synchronization is happening more often than control-flow is being recorded. When recording all control-flow, loads, and stores in the program, the impact of adding synchronization on trace buffer insertions is minimal. The best results are seen when Scheme #3 is applied, as will usually be the case. If the user makes the effort to manually perform data-flow and control-flow analyses, then they can insert the synchronization at the locations that have the least impact on the program.
Impact on Execution Time

In addition to reducing the percentage of trace buffer insertions that fit in the trace buffers, adding synchronization increases the execution time of the program. This is primarily due to the cost of retrieving the synchronization ID from hardware. To retrieve the synchronization ID from hardware, the software must perform a volatile load from memory-mapped hardware in the FPGA fabric, which runs at a much lower clock rate versus the ARM CPU. This load crosses domain bounds, retrieves the synchronization ID from the synchronization ID module in hardware, then returns it to the software. The measured increases to execution times are shown in Figures 4.17 and 4.18. Note the difference in Y axis for these two graphs.

The backprop benchmark saw up to a 15x increase to execution time. This happened under Scheme #1, when synchronizing after every memory operation on shared objects. To improve this, users could apply Scheme #2, which results in a 5x increase to execution time. This is due to many basic blocks which contained multiple synchronization instructions. However, to further reduce the impact, we would suggest the user manually perform data-flow and control-flow analyses to manually insert synchronization at efficient locations in the program. By doing so ourselves, we
Figure 4.18: Increase in \textit{srad} execution time for various levels of observation. Lower is better.

were able to maintain 100\% synchronization, or a total ordering of memory operations on shared objects, while having a minimal impact on performance as shown under Scheme \#3.

The \textit{srad} benchmark only saw up to a 0.045x increase to execution time under Scheme \#1. This is a modest increase to execution time, and likely acceptable to the user. Scheme \#2 does not see any improvement because there were no basic blocks with multiple memory operations on shared objects. Lastly, user-inserted or direct synchronization results in the least impact to performance.

All three of the synchronization schemes are valid under certain code structures. However, user-inserted synchronization will always result in the least impact to performance, but requires the user to manually perform control-flow and data-flow analyses and have a good understanding of the parallelism in the program.

4.7 Chapter Summary

Much work has been done to improve trace-based debugging, but it has mostly focused on hardware executing in isolation, not for HLS-accelerated programs. To address this lack of support for HLS-accelerated programs, this chapter puts forth techniques to implement trace-based
debugging for HLS-accelerated programs on shared memory heterogeneous systems. Additionally, we demonstrate techniques to synchronize the hardware and software trace buffer, providing a temporal correlation between the two, which allows users to follow debug data across the hardware and software domain bounds. To demonstrate the feasibility of our techniques, we built a prototype system into the LegUp HLS, and measured the impact of our techniques on execution time, and hardware and software trace insertions.

The on-chip debugging environment for HLS-accelerated programs is supported as follows. The user first selects the data to record through source code annotations, then the hybrid flow of the HLS tool LegUp identifies these annotations, and uses them to determine which recording instructions should be automatically inserted throughout the program. The user is able to choose to record control-flow, loads, stores, or function arguments through the annotations. The data is recorded to the software trace buffer, which is a circular buffer automatically inserted into the software IR. The cost of supporting trace-based debugging of HLS-accelerated programs was demonstrated through measurements on execution time.

Next in this chapter, the importance of synchronizing the hardware and software traces were discussed, and a synchronization technique as well as multiple synchronization schemes were presented. Synchronizing the hardware and software traces allows users to follow the flow of data across domains to the origin of their bugs. Without synchronization, following data across domain bounds becomes very difficult if not unfeasible.

The synchronization technique provides an ordering of memory operations on shared objects. Its implementation in both the hardware and software is presented. Additionally, three synchronization schemes were presented, each with varying impact based on code structure. The first scheme is the safest and synchronizes after every memory operation on shared objects. This guarantees that the ordering of memory operations on shared objects can be reconstructed post-execution. The second scheme replaces all synchronization calls in each basic block with a single one at the end of said basic blocks. The third scheme is user-inserted synchronization that requires the user to manually perform data-flow and control-flow analyses to determine where to insert synchronization.

The impact of adding synchronization to the hardware and software trace buffers was also demonstrated. Adding synchronization results in synchronization IDs being added to the trace.
buffers. The more synchronization IDs added to the trace buffer, the less debug data is preserved. Execution time is also affected due to the need to cross domain bounds to retrieve the synchronization ID from the synchronization ID module in hardware.

Though these techniques were focused on a shared-memory system, many of these techniques could be extended to PCIe-based systems. Capturing trace-data in both the hardware and software would likely remain the same, but changes would be needed to support synchronization. Additionally, the techniques presented in this chapter have focused on modifying the open-source HLS tool LegUp. Though this is possible in some HLS tools, commercial tools are generally closed-source, preventing the implementation of these techniques. To address these types of systems, Chapter 5 presents techniques for on-chip debugging of FPGA OpenCL kernels through modifying the source code. This work is an open-source patch to LegUp 4.0, available to the public at http://goeders.net/downloads/.
CHAPTER 5. OPENCL SOURCE-CODE LEVEL ON-CHIP DEBUGGING ENVIRONMENT

As high-level synthesis tools become increasingly wide-spread, they have come to support varying types of source-code languages and systems. Each of these languages and systems introduce different debugging challenges. Chapter 4 discussed debugging techniques for shared-memory systems using the LegUp HLS tool. Another commonly used system is PCIe-based systems, which is commonly supported by commercial tools. To support PCIe-based systems, many of these commercial tools support OpenCL. OpenCL is an extension to the C source code language that provides explicit control over synchronization and data transfers in heterogeneous systems. Additionally, when using OpenCL, users implement their hardware code as OpenCL kernels, from which the HLS tools generate the hardware design. OpenCL in many ways makes utilizing PCIe-based heterogeneous systems with FPGAs simpler, however, similar to what was mentioned in previous chapters, debugging OpenCL FPGA kernels in HLS-accelerated programs can be difficult. This is especially the case for bugs that require on-chip debugging techniques. These bugs may be dependant on I/O, interactions with other hardware modules, non-deterministic timings, bugs that take too long to simulate, or do not arise until the design is executed on-chip. One of the main challenges of tackling these types of bugs is the users lack of understanding of the generated RTL, as discussed in Chapters 2 and 4. Work has been done to provide trace-based debugging techniques for HLS-generated designs by maintaining a correlation between the source code and the generated RTL, however most of these works have focused on modifying specific HLS tools, or have focused on hardware executing in isolation, not on heterogeneous systems. Providing on-chip debugging support to OpenCL HLS-accelerated programs falls outside of these categories.

Recent the work [16], [19] addressed this through techniques very similar to those presented in this chapter, in that we also rely on the Rose compiler to automatically insert recording instructions into the source code. However, their works were focused on C-to-hardware HLS-generated circuits executing in isolation, not on OpenCL or heterogeneous systems. To our knowl-
edge, no previous work has explored source-code level debugging of OpenCL FPGA kernels in HLS-accelerated programs beyond hand-modifying the source code to observe specific variables [51]. As many of the commercial HLS tools are moving towards heterogeneous systems with OpenCL, we thought it important to explore how previous techniques could be applied to an OpenCL environment. This includes the use of features more easily available in OpenCL, such as using different memory regions for the trace buffer; global memory instead of the more conventionally used local memory BRAMs. Through using global memory, the trace buffer is accessible by both the host and accelerator, allowing the accelerator to record trace data to it, and the host to retrieve it after execution. Additionally, we wanted to explore how much of the source-code level debugging environment could be automated through current open-source tools, and provide a publicly available OpenCL debug tool that could potentially be used by others in the future.

This chapter puts forth the techniques and tool-flow necessary to provide this source-code level debugging environment for OpenCL FPGA kernels in HLS-accelerated programs, through source-code modifications using open-source tools. This debugging environment allows the user to identify, select, record, and present trace data to the user for all variable assignments in OpenCL source code. First we use the ROSE source-to-source compiler to identify all variable assignment operations in the kernel source code, then we present this information to the user in the form of a GUI, allowing them to select variables assignment operations to record. The selected variables are then passed back to the ROSE compiler, and recording instructions are inserted after each of the selected variable assignment operations. After the kernel finished executing the data is retrieved and presented to the user in a debugging environment. These techniques were originally presented in "On-Chip, Source-Level FPGA Debugging for OpenCL Kernels" by Matthew B. Ashcraft and Jeffrey Goeders, which will be submitted to the International Conference on Field-Programmable Technology (FPT’2020).

5.1 Objectives and Metrics

The objective of this chapter is to develop and automate a prototype source-code level debugging environment for on-chip debugging of FPGA OpenCL kernels. Though many previous works addressed on-chip debugging of HLS-generated programs, and the previous chapter demonstrated techniques for on-chip debugging of HLS-accelerated programs, most of these techniques
were focused on hardware in isolation, or required the modification of HLS tools. Unlike those works, supporting on-chip debugging of FPGA OpenCL kernels in HLA-accelerated programs requires support for heterogeneous systems without modifying HLS tools. This is because OpenCL is only supported by closed source commercial tools. As such, the goal of this work is to develop or identify techniques which can be applied to FPGA OpenCL kernels in a closed source environment.

The only previous work we identified which targeted OpenCL directly, required users to manually modify their source code to record changes to specific variables [51]. Though this makes progress towards providing a debugging environment, it is far from a full debugging environment where users select data to record, and the tool automatically inserts recording instructions or circuitry into the program. For this purpose, one of the other goals of this work is to determine the limits of automating the debugging environment using current open-source tools.

Other previous works from which we drew inspiration, demonstrated that trace debugging support can be added through modifying the source code directly [16], [19], and synthesizing the changes into the generated design through HLS. As our techniques build off of this idea, the source code modifications, and resulting changes in the synthesized design will affect execution time, as well as hardware resources on the FPGA. Quantitatively, we will focus on these effects on execution time and hardware resources, and look into other techniques, such as trace buffer caching and memory bursting, which may reduce their impact.

5.2 Context

The techniques presented in this chapter are aimed towards heterogeneous systems, similar to Figure 5.1, with a x86 CPU and FPGA connected over PCIe. Unlike the last chapter where the CPU and FPGA had a shared memory system, the PCIe-based systems targeted in this chapter do not provide a simple shared memory. Like a GPU-based system, all data needs to be transferred through explicit PCIe data transfers between the CPU and FPGA memories. The host(CPU) can access system main memory, as well as the main memory present on the FPGA accelerator card. The FPGA memory typically consists of both DDR memory on the FPGA card, as well as local memory within the FPGA fabric.

Modern FPGA tools such as Xilinx’s Vitis and Intel’s OpenCL SDK allow designers to target these PCIe-based systems, and this chapter focuses on providing debugging support for
this class of systems. In these systems, the host code, containing all of the OpenCL directives, executes on the host, and the OpenCL kernel code is generated into a hardware design using HLS, and instantiated on the FPGA. OpenCL is an extension of the C programming language, which allows users explicit control over data transfers between the host and FPGA, as well as controlling when kernels begin executing. The vendor tools automatically synthesize all necessary logic on the FPGA to support these data transfers and control over execution.

As will be explained throughout this chapter, debugging such systems requires a unique approach, independent from the work discussed in the previous chapter. Our approach is to automatically modify the OpenCL kernel code, such that variable assignment operations of interest are recorded into FPGA memory on the FPGA card. Once execution has completed, the host initiates the transfer of this trace data back to the host, where it can be processed and presented to the user.

To support and implement the techniques presented in this chapter, two open-source tools are needed: the source-to-source compiler ROSE, and custom Python scripts. The ROSE compiler will be introduced here, and the Python scripts will be addressed in Section 5.4.1.

Figure 5.1: Execution Environment
5.2.1 ROSE

The ROSE compiler [61] is a source-to-source compiler used for translating one high-level language to another, or for optimizing a specific high-level language. Because ROSE is a source-to-source compiler, it uses an IR that maintains many of the high-level source code constructs. The IR it generates and performs analyses and optimization on is an abstract syntax tree (AST). The AST is a graph based representation where nodes represent constructs in the source code, and edges represent contents of the child node. An example of an AST is shown in Listing 5.1 and Figure 5.2. Listing 5.1 represents possible source code. Figure 5.2 represents a simplified AST for that source code.

\[
X = Y \ast Z; \\
Y = W \ast V;
\]

Listing 5.1: AST Source Code

```
X = Y * Z;
Y = W * V;
```

Figure 5.2: AST Example

The AST shown in Figure 5.2 is only a portion of the AST graph. The highest node in the original graph would represent all of the code, with edges to the Functions and the Globals. Descended from the Function nodes are the BasicBlock nodes where this graph begins. This BasicBlock node contains the two instructions shown in Listing 5.1. Each of the instructions in the
basic block are given their own branch in the tree. In this case, there are *Assignment* nodes representing each of the assignments to $X$ and $Y$. The edges of the *Assignment* nodes represent the LHS and RHS arguments, $X/Y$ and *Multiply*. The *Multiply* nodes are further broken up into their arguments.

When performing analyses and optimizations to the AST, the nodes are used to identify specific instructions and operands, and nodes are added or modified to change the code. If the user was inserting extra instructions between the two instructions shown in Listing 5.1, they would insert an edge and node between the two *Assignment* nodes currently in the graph. These modifications are performed using a ROSE transformation, which consist of custom analyses, optimizations, and transformations users can develop.

### 5.3 On-Chip Debugging of OpenCL Programs

To provide a source-code level debugging environment for on-chip FPGA debugging of OpenCL kernels, this chapter puts forth techniques to identify, select, record, and present the trace data to the user for variable assignments in OpenCL kernel source code. In developing techniques to handle each of these steps, we have learned from techniques presented in Chapter 4, and in recent works [16], [18], [19], [51]. These techniques will be demonstrated in this section, as well as a tool flow we used to implement them, shown in Figure 5.3. Note that due to the current limitations of the Rose compiler, we use Python to automatically change the code from OpenCL to C before passing it to Rose, and translate it back to OpenCL after Rose is finished (discussed further in Section 5.4.1). The tool flow is broken up into 5 steps:

1. Identify assignment operations in the kernel code,
2. Users select which operations to record,
3. Insert recording instructions into the kernel code,
4. Execute the design in hardware, and retrieve the data recorded in the trace buffer,
5. Present the execution trace data to the user.
Steps 1-3 rely on the Rose compiler. After step 3, the design is synthesized using commercial HLS tools. Step 4 involves executing on the heterogeneous system. Step 5 take place post-execution.

5.3.1 Step 1: Identifying Variable Assignments

An important part of any trace-based debugging environment is the ability to select variables to record during execution. This is due to 1) limited memory available on-chip, and 2) tracing signals requires extra hardware resources. During execution the limited memories are shared between all variables being recorded to the trace buffer, meaning the more variables selected for execution, the less data for each variable can be saved. Additionally, the more variables selected to record, the more hardware resources are needed, which can be challenging in congested designs [41].

In order to identify all of the variable assignment operations we used the ROSE compiler, as shown in Step 1 of Figure 5.3. As discussed in Section 5.2.1, the ROSE compiler transformations operate on an AST, whose nodes represent functions, operations, arguments, operands, variables, etc. We traverse the AST to identify all Assignment nodes contained in each of the kernels, and
sort them by function. Each of the Assignment nodes, and corresponding variable assignment operations, are assigned globally unique identifiers (IDs), which will be used to correlate the trace data with the source code post-execution.

5.3.2 Step 2: Selecting Variable Assignments for Recording

After the variable assignment operations in the kernel source code have been identified, the user needs to be able to select which variables to record, as shown in Step 2 of Figure 5.3. The ROSE transformation calls a GUI we prototyped, as shown in Figure 5.4. This GUI shows the function, checkable assignment operations, and the source code, so the user may accurately select
variable assignment operations to record. Once the user has selected the variables to record the
GUI returns a list of selected variable assignment operations to the ROSE transformation.

5.3.3 Step 3: Modifying the Source Code to Record Selected Variable Assignments

Many of the previous works on debugging HLS generated designs have relied on modifying
HLS tools directly, or on hardware executing in isolation. Unfortunately, many of these techniques
are not applicable to HLS-generated designs from OpenCL kernels and commercial HLS tools. To
provide a debugging environment that is applicable to all HLS tools, we adopted ideas from recent
works to provide debugging support through adding recording instructions into the source code
directly [16], [51], and rely on the HLS tools to synthesize the changes into trace buffer insertion
logic.
To modify the source code we used similar techniques to the software recording techniques presented in Chapters 2 and 4. In these techniques, instructions are inserted into the source code to record values to a trace-buffer, as well as globally unique IDs that represent where the data was recorded. This is demonstrated in the code listings in Listings 5.2 and 5.3. Listing 5.2 represents possible source code where the user wants to record all assignments to the value $X$, and Listing 5.3 represents the recording instructions inserted to record $X$. After the assignment to $X$ in Listing 5.3, two instructions are inserted to record data to the trace buffer, $X_{ID}$ and $X$. When later parsing the trace buffer, the globally unique ID, $X_{ID}$, represents that the next entry in the trace buffer is that value of $X$ on line 1 of the source code.

At the end of the kernel source code a final ID and value are recorded to the trace buffer, as shown on lines 6 and 7 of Listing 5.3. The final ID marks the final entry into the trace buffer for this thread. The final value contains the final offset into the trace buffer, and is used to find the oldest entry into the trace buffer. This will be explained later in this section.

![Figure 5.5: Trace Buffer with Entries from Listing 5.4](image-url)
Trace Buffer

To provide access to the trace buffer for both the host and FPGA, the trace buffer resides in global memory. Similar with other OpenCL buffers, a trace buffer is declared and allocated by the host code for each FPGA kernel and device in use. If the kernel is broken up into multiple threads, the trace buffer is divided between the threads equally. When this happens, instead of one large trace buffer, there are multiple smaller thread trace buffers contained in the device trace buffer. The trace buffers layout is shown in Figure 5.5.

In this trace buffer example, the buffer is split up into multiple threads, with each thread allotted eight entries. In the buffer, indices 0-7, 8-15, and 16-23 represent the thread trace buffers for Thread 1, Thread 2, and Thread 3 respectively. The size of each thread’s trace buffer is set by the user, and each thread’s trace buffer is treated as a circular buffer. When determining the size of the thread trace buffers, the user should consider the resources available on the device, the number of kernel threads executing on the device, and the amount of data they expect to record in each thread. Additionally, the size of each thread’s trace buffer must be a power of two due to the computation of the indices into the buffers. This will be discussed later in this section.

The size of each devices trace buffer is traditionally based off of the number of entries in the trace buffer, multiplied by the bit-width of each entry. For our trace buffer, the bit-width is determined by the data type of the trace buffer, which can be chosen by the user. Additionally, the trace buffer can be broken up into multiple thread trace buffers. As a result, the overall size of the trace buffer is the data type of the trace buffer, multiplied by the size of each thread’s trace buffer, multiplied by the number of threads.

Implementation

The list of selected variable assignment operations from the GUI is returned to the ROSE transformation, where upon it inserts recording instructions after each selected variable assignment operation. The instructions record the value and the globally unique ID, representing the location of the value, to the trace buffer. The index into the thread’s trace buffer must be calculated based upon the thread, thread trace buffer size, and the current offset into the buffer. Additionally, since
each thread’s trace buffer is a circular buffer, the offset into the thread’s trace buffer must masked to keep it within the entries allotted to the thread. The computation is:

\[
ThreadID \times TBSIZE + (offset \& (TBSIZE - 1))
\]

Where \(ThreadID\) represents the current thread ID. \(TBSIZE\) is the per-thread trace buffer size. The bitwise AND is performed with the offset into the thread’s trace buffer and the max index in order to have the index continually wrap back to the start of the thread’s trace buffer as a circular buffer. An example of this is shown in Listing 5.4, and the entries in Thread 2 of Figure 5.5.

In the listing, the size of the trace buffer is eight entries, meaning every eight entries in the buffer represents a different thread’s trace buffer. The offset is the index into the thread’s trace buffer, and there is a bitwise AND operation between it and the max offset (thread buffer size minus one). The entries to the trace buffer on lines 3-4 of Listing 5.4 are shown on indices 8-9 if Figure 5.5.

After the recording instructions for the selected variable operations are inserted, the two final recording instructions are inserted at the end of the kernel to mark the end of entries into the trace buffer, and to record the final offset into the thread’s trace buffer. This is shown on lines 7-8 of Listing 5.4 and indices 12-13 of Figure 5.5. The final offset is used in determining where the oldest entry in the trace buffer is. If the final offset is less than the size of the thread’s trace buffer, the the oldest entry into the thread’s trace buffer is the first entry. If the final offset is greater than the size of the thread’s trace buffer, then the buffer has overflowed, and looped back to the initial index at least once. In this case the oldest entry in the trace buffer is the entry immediately after the final offset. In the entries in Figure 5.5, the final offset in index 13 contains a value of 13. Since this is larger than the thread buffer size of eight, this means the oldest entries into the thread’s trace buffer are contained in indices 14-15.

5.3.4 Step 4: Modifying the Host Program, Executing Hardware, and Retrieving Data

In addition to the automatic insertion of recording instructions into the kernel source code, modifications are needed in the software source code to handle the trace buffer kernel argument. These modifications are done manually due to the variation in creating and adding kernel argu-
ments in the software source code. The main modification needed is to declare the trace buffer in the host and pass it to the kernel as an extra argument. This will allow the trace buffer to reside in global memory in the hardware, where the hardware can write to it, and the software can read from it.

In declaring and allocating the trace buffer, the user needs to determine the data type, and the size of the trace buffer. By default, the type of the kernel trace buffer is set to `long`, but the user may manually change this in the kernel source code. The size of the data type, as mentioned previously, is:

\[
\text{sizeof}(\text{DataType}) \times \text{EntriesPerThread} \times \text{NumThreads}
\]

where `DataType` is the trace buffer’s data type (int, long, float, etc.), `EntriesPerThread` is the number of entries in each thread’s trace buffer, and `NumThreads` is the number of threads executing on the device. If the kernel is executing on multiple devices, or if there are multiple kernels on the device, then a trace buffer needs to be declared for each device and/or kernel. For example, if the datatype was of type `long`, there were 4096 entries in each thread’s trace buffer, and there were 8 threads executing on the device for the kernel, then the calculation would be `sizeof(long) * 4096 * 8`.

The other modification needed in the source code is to record the trace buffer to a file. The contents of each trace buffer are separated by spaces, and the different trace buffers are separated by unique IDs.

### 5.3.5 Step 5: Displaying the Trace Data

Once the trace data is recorded to a file, the user can analyze the data at any time. To assist in this, we created a templated HTML page, using the Python Jinja2 framework. The page is dynamically generated with the selected variable assignment operations, and the contents of the trace buffer. Since the page is dynamic, the user can generate a trace timeline for any of the selected variable assignment operations they previously selected. The trace timeline is ordered from the oldest entry to the newest. Our prototyped trace viewer is shown in Figure 5.6. In the viewer, the data is organized by device, thread, and variable assignment. In the example shown,
the assignments to variables $y$ on line 76 and $x$ on line 77 of `mandelbrot_kernel.c`, for Device 0, Thread 0 are shown.

![OpenCL Debug Trace Viewer](image)

Figure 5.6: OpenCL Debug Trace Viewer
An important aspect of note is that this approach only allows users to observe a relative ordering of events. The “timescale” is an ordering in which data was written to the trace buffer. Obtaining a actual wall-clock times might be possible through adding a high-precision counter to the hardware via RTL kernel, and recording this to the trace buffer in addition to the data and ID. However, this would introduce other overheads, and likely be tool-chain specific.

5.4 Challenges and Optimizations

5.4.1 Workarounds to Handle OpenCL Files in ROSE

At the time of our work, OpenCL support for the ROSE compiler was still in development. As a result, we had to modify the OpenCL source code to make it acceptable to ROSE. As OpenCL is an extension to the C programming language, which is supported by ROSE, we needed to remove the unsupported OpenCL constructs. To do this we used an automated Python script to change the file type from `.cl` to `.c`, and commented out all of the OpenCL directives. After the ROSE transformation finished executing, the Python script changed the file type back to `.cl`, removed the comments around the directives, and added directives to the new trace buffer kernel arguments.

5.4.2 Triggering

An important part of trace based debugging is allowing users to set triggers to stop recording data at any point in execution. This is due to the limited memories available on the device, and the nature of trace buffers as circular buffers, which continually overwrite older data. Without a means to stop execution, important data could be overwritten. We considered adding support for this to the GUI, which would in turn add it to the source code to be synthesized into the hardware design, but felt that it would limit the types of triggers available.

Instead of supporting the insertion of triggers through the GUI, we elected to have users insert triggers into the OpenCL kernel source code using an `if` statement with a `return` instruction to terminate kernel execution. This allows users to create triggers of arbitrary complexity. Note that this only allows users to halt execution with a trigger. Many debugging tools allow triggers to halt execution after a certain number of cycles. This could probably be done through further kernel
source code modifications with some sort of counter that is checked before data is recorded to the trace buffer, but this could affect scheduling in the HLS tools. However, as this is not the focus of this work, we have not explored this further.

The downside of this is that setting a new trigger requires a recompile of the system, as is also the case for selecting new variables to record. It might be possible to get around this through adding conditional triggers based upon kernel arguments. This is something future work may address.

5.4.3 Trace Buffer Caching

The techniques we have presented rely on writing and retrieving trace data to/from the trace buffer in global memory, however, other techniques might be beneficial. One of these techniques is trace buffer caching. Trace buffer caching involves allocating a local memory array, recording data to that array, then periodically writing that array to the globally allocated array. This allows users to take advantage of memory bursting, a technique often used in FPGA OpenCL kernels to more efficiently copy data back to the host.

To quantify the results of trace buffer caching versus our technique of recording straight to global memory, we modified some of the tests to perform trace buffer caching, similar to the pseudo code in Listing 5.5. After trace data and the globally unique IDs are recorded to the local trace buffer, the index is compared to the max address to determine if the local buffer is full and needs to be copied to the global trace buffer. If so, the program then calculates the base address in the global trace buffer to record data, then increments the bufferCnt, and copies the data over. The bufferCnt represents how many times the local buffer has been copied over to the global trace buffer.
buffer. Next if the global trace buffer has reached its max index, the program resets the bufferCnt to zero like a circular buffer. The quantitative results of implementing this will be explained in the next section.

### 5.5 Experiments and Results

The techniques put forth in this chapter were prototyped and tested on five Intel OpenCL benchmarks [62] using Quartus 18.1 and Intel FPGA SDK for OpenCL 18.1. The tests were run on a Xeon E6-2699 processor with a Stratix 10 SX D5005 FPGA connected over PCIe. The benchmarks tested were Compute Score (compute), Finite Difference Computation 3D (fd3d), FFT 1D (fft), Multithread Vector Operation (multiop), Time-Domain FIR Filter (tdfir), and Vector Addition (vector_add). The benchmarks are partitioned as originally set by Intel, with most of the main computation for the benchmark residing in the OpenCL kernel source code. The thread trace buffer sizes for each kernel were eight for multiop and vector_add because each thread only had a single instruction and many threads, and 4096 for compute, fd3d, fft, and tfir, as they are more complex benchmarks with many instructions.

Adding source-code level debugging to OpenCL kernels increases the resource usage on the FPGA, and can affect execution time of the kernels. The quantitative results were gathered from running the benchmarks under four scenarios:

1. **Baseline**: original benchmark.

2. **Framework**: the bare-bones debug framework added to the original benchmark, but with no variables traced. The global trace buffer is added as a kernel argument, and the kernel records the final index and offset to the thread’s trace buffer at the end of the kernel execution.

3. **Recording**: Variable assignment operations are recorded within the main algorithms. These are variables within the main loop or algorithm of the benchmark that could be beneficial in debugging the program. The selected variable assignment operations are mentioned in the next subsection.

4. **Caching**: Two benchmarks, compute and tfir, were run with trace buffer caching to measure its impact. The caching tests record the same variable assignment operations as the previous
scenario, but included a local memory buffer with 512 entries, that is copied to the global trace buffer whenever it is full.

5.5.1 Benchmarks

The benchmarks used and variables recorded in the benchmarks are as follows:

compute is a document filtering benchmark that looks at the frequency of certain words in a document. Its algorithm is based around a Bloom filter. The benchmark has two kernels, reduction and compute_score. In reduction the changes to the total value within the main loop are recorded. In compute_score the results of the function calls in each of the unrolled loops were recorded.

fd3d is a 3D finite-difference stencil-only computation. In its kernel, the results of part of the computation for the next computation point, xtile, is recorded.

fft is a 1D radix-4 complex FFT and inverse FFT performed sequentially with two kernels, fetch and fft1d. In fetch, the assignments to the local memory buffer in the first loop are recorded, and the updating buffer address in the second loop is recorded. In fft1d, the results of the data after each step of the FFT are recorded.

multiop is a simple benchmark with two kernels, each containing one vector operation. The results of each of these vector operations are recorded.

tdfir is a time-domain finite impulse response (FIR) filter benchmark. In the kernel the real and imaginary coefficients in each iteration of the outer loop are recorded.

vector_add contains a single vector addition operation. This is the recorded variable. Though this kernel is simple, this benchmark can support millions of concurrent kernels on multiple devices.

5.5.2 Results

The results are shown in Table 5.1. The table is organized according to the benchmark and scenario run. Each benchmark was executed without any modifications, with the debugging
Table 5.1: Kernel Debug Runtime and Resource Overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel runtime (ms) vs base</th>
<th>ALUT vs base</th>
<th>FFs vs base</th>
<th>ALMs vs base</th>
<th>RAMs vs base</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>2208.67</td>
<td>202803</td>
<td>384819</td>
<td>91276</td>
<td>2109</td>
</tr>
<tr>
<td>compute w/ Framework</td>
<td>2049.87  -7%</td>
<td>212429</td>
<td>404025</td>
<td>94571</td>
<td>2091 -1%</td>
</tr>
<tr>
<td>compute w/ Recorded</td>
<td>2134.52  -3%</td>
<td>221890</td>
<td>415005</td>
<td>98896</td>
<td>2132 1%</td>
</tr>
<tr>
<td>compute w/ Caching</td>
<td>2390.60  8%</td>
<td>243465</td>
<td>415249</td>
<td>109062</td>
<td>4838 129%</td>
</tr>
<tr>
<td>fd3d</td>
<td>199.03</td>
<td>143905</td>
<td>319809</td>
<td>74880</td>
<td>1894</td>
</tr>
<tr>
<td>fd3dw/ Framework</td>
<td>207.72  4%</td>
<td>149118</td>
<td>325578</td>
<td>76350</td>
<td>1894 0%</td>
</tr>
<tr>
<td>fd3dw/ Recorded</td>
<td>219.37  10%</td>
<td>159051</td>
<td>345363</td>
<td>79945</td>
<td>1880 -1%</td>
</tr>
<tr>
<td>fft</td>
<td>7.24</td>
<td>125425</td>
<td>294047</td>
<td>76762</td>
<td>1084</td>
</tr>
<tr>
<td>fft w/ Framework</td>
<td>7.84   8%</td>
<td>134077</td>
<td>308195</td>
<td>70861</td>
<td>1084 0%</td>
</tr>
<tr>
<td>fft w/ Recorded</td>
<td>8.83   22%</td>
<td>144125</td>
<td>340401</td>
<td>74794</td>
<td>1089 0%</td>
</tr>
<tr>
<td>mutiop</td>
<td>1.03</td>
<td>121140</td>
<td>268675</td>
<td>65620</td>
<td>783</td>
</tr>
<tr>
<td>mutiop w/ Framework</td>
<td>1.12   9%</td>
<td>129773</td>
<td>291114</td>
<td>68605</td>
<td>819 5%</td>
</tr>
<tr>
<td>mutiop w/ Recorded</td>
<td>1.13  10%</td>
<td>129773</td>
<td>295102</td>
<td>68567</td>
<td>819 5%</td>
</tr>
<tr>
<td>tdfir</td>
<td>1.10</td>
<td>157996</td>
<td>418395</td>
<td>86107</td>
<td>956</td>
</tr>
<tr>
<td>tdfir w/ Framework</td>
<td>1.03  -6%</td>
<td>153447</td>
<td>361718</td>
<td>75860 -9%</td>
<td>802 -16%</td>
</tr>
<tr>
<td>tdfir w/ Recorded</td>
<td>1.26  15%</td>
<td>167366</td>
<td>439685</td>
<td>89904</td>
<td>1001 5%</td>
</tr>
<tr>
<td>tdfir w/ Caching</td>
<td>851.91  77347%</td>
<td>385809 144%</td>
<td>1009517</td>
<td>187611 118%</td>
<td>2178 128%</td>
</tr>
<tr>
<td>vector_add</td>
<td>3.37</td>
<td>107905</td>
<td>249100</td>
<td>60166</td>
<td>624</td>
</tr>
<tr>
<td>vector_add w/ Framework</td>
<td>3.71  10%</td>
<td>112035</td>
<td>251785</td>
<td>61584</td>
<td>642 3%</td>
</tr>
<tr>
<td>vector_add w/ Recorded</td>
<td>3.75  11%</td>
<td>112107</td>
<td>249732 0%</td>
<td>61990 3%</td>
<td>648 4%</td>
</tr>
</tbody>
</table>

framework in place (w/ Framework), with variables recorded during execution (w/ Recorded), and with trace buffer caching (w/ Caching). The second column represents the total execution time for all kernels in the given benchmark. The remaining columns represent resource utilization on the FPGA; ALUTs, FFs, ALMs, and RAMs. Note that the hardware resource utilization results represent the complete hardware design, including the full OpenCL shell added by default to all OpenCL SDK designs, specific to the FPGA used.

The impacts to execution time range from a slight improvement to execution time to a 22% increase in execution time for all tests except tdfir with caching. These changes in execution time are due to the need to record data to global memory, and the impact that has on scheduling. By inserting extra instructions that record to memory, the scheduler may have increased or reduced flexibility to perform pipelining or other operations. The exception to these modest changes to execution time is tdfir with caching, as will be discussed later in this section.

With the exception to the tests with caching, the changes to resource utilization were modest across the board, ranging from -16% to 16%. These changes are due to the the additional trace buffer insertion logic, and the effect that has on the scheduler and optimizations.
The most informative results are those from the caching tests. Trace buffer caching is commonly used to improve performance of HLS-accelerated programs through the use of memory bursting. As such one might hypothesize that it should have resulted in a decrease in execution time, and a small increase in resource utilization due to the extra local memory buffer and logic to copy the local trace buffer to the global trace buffer. However, the opposite is shown. Trace buffer caching resulted in a 8% to 144% increase in hardware resource utilization, and an 8% to 77347% increase to execution time. Additionally, the _fft_ benchmark was run with caching, but it was unable to finish the scheduling portion of synthesis. This is likely due to the conditional logic inserted for trace buffer caching after each recorded variable. This conditional logic, which ended up inside the main loops of the algorithms in the benchmarks, performed large memory operations between local and global memory. Because of this, the scheduler would have no way of optimizing or pipelining the loops, as it might have to wait for an unknown amount of time per loop iteration. This likely resulted in the removal of most optimizations from the main loops in the algorithms of the benchmarks.

Overall, the quantitative results show a modest impact on execution time and hardware resources when writing directly to global memory. Additionally, they demonstrate the effect of adding conditional logic and trace buffer caching in the main loops on program execution time and hardware resources.

### 5.6 Chapter Summary

High-level synthesis tools have come to support varying source-code languages and systems, including OpenCL on PCIe based systems. OpenCL allows for explicit control over kernel generation, and allows users to write their programs in the C source code language. Because of this, the code can be debugged at the source-code level with relative ease. However, there are some bugs that do not arise until execution time, such as those dependant on I/O, interactions with other hardware modules, non-deterministic timings, or bugs that take too long to simulate. These types of bugs require the use of on-chip debugging techniques such as trace-based debugging. Though work has been done to improved trace-based debugging for HLS-accelerated programs, it has mostly focused on providing support for specific HLS tools, or on hardware executing in isolation.
This chapter put forth techniques to provide a source-code level debugging environment for on-chip debug of HLS-accelerated designs from OpenCL kernels. These techniques allow the user to identify, select, record, and view the trace results of variable assignment operations in the kernels through source code modifications. By applying these techniques at the source-code level, these techniques are applicable to all HLS tools. Additionally, we explored the effect of using trace buffer caching to improve program performance.

To quantify our results, we developed a prototype and tool-flow that allowed us to test our techniques on five Intel OpenCL FPGA benchmarks. The quantitative results showed a modest impact on execution time when following our techniques, and a substantial impact on execution time when using trace buffer caching. The substantial impact of trace buffer caching on execution time and hardware resources is likely due to the conditional logic and memory operations added to the main loops of the algorithms in the benchmarks.
CHAPTER 6. CONCLUSION AND FUTURE WORK

This chapter provides a summary of this dissertation, reviews the contributions put forth as well as their impact, and discusses future research opportunities based on this work.

6.1 Dissertation Summary

Programmers and developers have continually looked for ways to achieve year-over-year performance improvements through microprocessor architectural advancements and compiler support. Unfortunately, the year-over-year performance improvements have slowed down in recent years due to the power wall and scaling challenges [7], [8]. To get around this, programmers and developers have begun to focus more on non-traditional computing devices, such as GPUs and FPGAs. GPUs excel at performing thousands of computations in parallel, but are restricted by excessive control-flow or data dependencies. FPGAs consist of reprogrammable logic that can implement arbitrary digital circuits or hardware designs. Both of these non-traditional computing devices can execute in conjunction with general purpose microprocessors in heterogeneous systems, and provide large performance improvements to program execution, but they each introduce their own challenges.

To take full advantage of GPUs, users must identify highly parallel code in their program, properly annotate it with optimization directives, and handle the data transfers between the software on the host microprocessor, and the hardware kernel code on the GPU. Efficiently scheduling the data transfers is of particular importance due to the excessive overhead it imposes. The overhead can even dwarf the improvement of using the GPU at all [10]. Under simple code structures, scheduling data transfers is as simple as transferring data used on the GPU to the GPU immediately before the kernel invocation, and transferring data modified on the GPU to the host immediately after the kernel invocation. However, under more complex code structures, such as branches, loops, nested call structures, and multiple kernel invocations scattered throughout the program, efficiently
scheduling the data transfers becomes very difficult. Efficiently scheduling data transfers in these scenarios can require program-wide control-flow and data-flow analyses, which are difficult to perform manually.

Fully utilizing FPGAs introduces many other challenges including designing and debugging the hardware design. Traditionally, utilizing FPGAs required hardware designers to create hardware designs using RTL languages. The RTL languages allow users to describe hardware circuits in terms of registers, wires, and state machines. In recent years, with the growing popularity of high-level synthesis (HLS) tools, FPGAs have become more widely available to programmers and developers. HLS tools translate high-level source code languages, such as C, to RTL languages, reducing the need for programmers to learn RTL languages. Many HLS tools can also provide the ability to generate complex programs for heterogeneous systems, allowing programmers to use FPGAs to accelerate their programs. Among the various benefits of using HLS tools is the ability to debug the to-be-generated designs and programs at the source code level using traditional source-code level debugging techniques. However, some bugs are made more challenging due to HLS tools, particularly those that require on-chip debugging techniques. These types of bugs usually require users to familiarize themselves with the generated design before they can debug it, which can take a significant amount of time. This challenge is further exacerbated in HLS-accelerated programs, where in addition to understanding the HLS-generated design, the user must familiarize themselves with the interface between the domains, and find ways to follow debug data across domain bounds.

Additionally, many of the techniques developed to improve on-chip debugging for HLS-generated designs require modifications of HLS tools, or they target hardware designs executing in isolation. Unfortunately, some commonly used source code languages and systems, such as OpenCL on PCIe based heterogeneous systems, are only supported by commercial tools. Thus applying the techniques developed for on-chip debugging of FPGA OpenCL kernels in heterogeneous systems becomes very difficult.

Fortunately, many of the challenges introduced by non-traditional computing devices can be uniformly addressed by compilers. Compilers already perform a majority of analyses, optimizations, and transformations on users code, and allow for users to develop and implement custom techniques. Additionally, most HLS tools are built as part of compilers, allowing users
to take advantage of their features. The objective of this dissertation was to use compiler analyses, optimizations, and transformations to address the challenges of efficiently scheduling data transfers, on-chip debugging of hardware and software in HLS-accelerated programs, and developing a source-code level debugging environment for FPGA OpenCL kernels in HLS-accelerated programs.

In Chapter 3 we demonstrated the importance of scheduling data transfers, and presented novel techniques to efficiently schedule data transfers based off of dominator-tree and modified liveness analyses. On-chip debugging techniques for heterogeneous systems were discussed in Chapter 4. These techniques relied on recording both software and hardware execution, and synchronizing the hardware and software traces through tracking memory operations on shared objects. Chapter 5 demonstrated techniques to develop a source-code level debugging environment for OpenCL FPGA kernels in HLS-accelerated programs through source-code modifications. The modifications, automatically inserted through the source-to-source compiler ROSE, are synthesized into the hardware design using any HLS tool. The remainder of this chapter will summarize the contributions of this dissertations in more detail, and outline future research possibilities.

6.1.1 Summary of Contributions

Efficiently Scheduling Data Transfers for Non-traditional Compute Devices

The first contributions presented in this dissertations, in Chapter 3, addressed the need for efficient scheduling of data transfers for accelerators. Data transfers are expensive, and in some cases can outweigh the benefit of using an accelerator, so we thought, how can compilers reduce this overhead? Prior to this work, data transfers manually inserted by the user would not be optimized by the compiler under the assumption that the user knows best. However, we demonstrated that efficiently scheduling data transfers in programs with complex code structures and multiple kernel invocations requires program-wide control-flow and data-flow analyses. These types of analyses can be very difficult and time consuming for users to perform, however, they are the types of analyses typically performed by compilers. Because of this, our approach to reducing the impact of data transfers was to use compilers to automatically schedule data transfers in efficient locations. In the chapter, we introduced techniques to automatically efficiently schedule data
transfers between the hardware and software, and demonstrated the substantial impact it had on execution time, and the amount of data transferred.

The process of efficiently scheduling data transfers was broken up into five steps. 1) Alias analysis was used to identify memory locations read from and written to in the kernels. 2) The alias analysis was then used to identify all instructions accessing memory locations in the software code that alias with those used in the kernels. 3) A scheduling graph was created for the portion of the program containing those instructions. The scheduling graph was a combination of a control-flow, data-flow, and call graph. 4) Dominator-tree and a modified liveness analysis were performed on the graph. The modified liveness analysis we presented was a novel analysis used to determine where the data transfers could be scheduled in the program. Using these results we developed an algorithm to schedule the data transfers outside of loops and control-flow, and eliminated unnecessary data transfers. 5) We inserted markers into the code for the back-end to automatically insert the data transfer instructions.

The effect of these techniques resulted in orders-of-magnitude improvement to execution time, number of data transfers, and the number of bytes transferred. At the time of our publication [1], we presented a novel scheduling graph, modified liveness analysis, and scheduling algorithm for efficient scheduling of data transfers. These techniques could be adopted and applied to other tools or compilers, to reduce the burden placed on users to identify efficient locations to transfer data, and reduce the impact of data transfers on program execution.

On-chip hardware and software debug of HLS-accelerated programs

The next contributions of this dissertation, presented in Chapter 4, were focused on debugging HLS-accelerated programs on heterogeneous systems. Though much work has been done in the past regarding trace-based debug of HLS-generated programs, almost all of the work focused on hardware executing in isolation. However, most HLS tools are moving to support HLS-accelerated programs for heterogeneous systems, allowing programmers to accelerate their programs without having to understand how the FPGA or communication with the FPGA work. Unfortunately, this makes on-chip debugging of these programs much more difficult, as the bug could arise in the hardware, software, or in the communication between the two. This led to the idea of how can we debug heterogeneous systems in a unified debugging environment? To address this, we presented...
the first unified on-chip debugging environment for both the hardware and software, allowing pro-
grammers to perform trace-based debugging, and view debug data from both domains in a unified
debugging environment.

Support for the unified source-code level debugging environment is broken up into the
hardware, software, and synchronization between the two. For the hardware, we took advantage of
previous work that implemented trace-based debugging circuitry into the LegUp HLS tool [15]. For
the software, we provided a user driven approach, where the user could insert annotations into the
source code to select variables, control-flow, and/or function arguments to record during execution.
We modified the LegUp HLS tool to identify the annotations, automatically inserted a trace buffer,
and recording instructions into the software code. These instructions record the trace data and
globally unique identifiers (IDs) to the trace buffer. The IDs are used to correlate the trace data
with the source code post-execution. The addition of the software recording instructions allowed
us to collect trace data from both the hardware and software, and present it to the user in a unified
debugging environment.

In addition to demonstrating techniques to record and view trace-data from both the hard-
ware and software domains, we addressed the problem of how we can follow debug data across
domain bounds. Trace-based debugging relies on following the effects of a bug back to its origin
to determine its cause. However, when the effects of the bug cross domain bounds through shared
objects, it becomes much more difficult to follow the effects back to the origin. To address this
we initially thought that we needed to align the hardware and software traces, so that we could
determine what both were doing at the same time. However, we realized that the only data we can
not follow across domain bounds were memory operations on shared objects. Because of this, we
realized that we only needed to determine the order of operations on shared objects to determine
where data came from, allowing us to follow it across domain bounds back to its origin. To address
this, we developed a synchronization technique based on a global counter, which keeps track of the
most recent modifications to shared objects in the program. Our synchronization technique relied
on identifying all shared objects in the program through analyzing memory operations on global
variables in both the software and the hardware. Global variables accessed by both domains were
considered shared objects. The list of memory operations on shared objects was used to identify
all hardware states that should be synchronized. A hardware module was designed to manage the
synchronization ID, handling requests for synchronizing from the software, and synchronizing the hardware when the current state matched states with memory operations on shared objects.

In the software, one of the big questions we addressed with synchronization was how often we needed to synchronize. Synchronization adds extra overhead in terms of trace buffer insertions and execution time, so it was important to find the balance between being able to follow debug data across domain bounds, and reducing the impact to these overheads. The most basic approach was to synchronize after every memory operation on shared objects, however in some situations this was overkill. To address this, we proposed three synchronization schemes that could be used depending on the code structure to reduce the overheads mentioned. The first synchronization scheme did as previously mentioned, as synchronized after every memory operation on shared objects. This guarantees that the user will be able to follow trace data across domain bounds. The second synchronization scheme applied to scenarios where there were multiple memory operations on shared objects in a single basic block. In these situations, the synchronization can be applied at the end of the basic block, instead of after each memory operation on shared objects, if there are no other memory operations on shared objects executing at the same time elsewhere in the program. Though this can drastically reduce the impact on the program, there are still further improvements that could be made. Unfortunately, these synchronization schemes are at the limit of what was available through analyses in the LegUp HLS tool. Because of this, the last synchronization scheme relies on the user manually analyzing the program to determine the best place to synchronize, such as before or after a kernel invocation. This usually will result in the least impact to execution time and software trace buffer entries, but it requires the most work out of the user.

To quantify the effect on the generated program, we measured the impact to execution time when recording data to the software trace buffer, and the impact on execution time, and insertions into the hardware and software trace buffers when synchronizing under the different synchronization schemes. When recording data to the software trace buffer, our measurements showed an average increase to execution time of 97% under the worst possible scenarios. When synchronizing the hardware and software traces, our quantitative results showed an average decrease of 33% in the total execution trace remaining in the trace buffer, and varying results in the software depending on which data was being recorded. The impact synchronization had on execution time varied from 1X to 15X depending on which synchronization scheme was used, and the structure of
the code. The data showed that programs with few memory operations on shared objects resulted in minimal impact on execution time regardless of the synchronization scheme used. However, programs with many memory operations on shared objects, including within nested loop structures, had much greater impacts on execution time under the first synchronization scheme, and we would recommend using the second or third synchronization scheme if possible.

At the time of our publications [2], [3] we presented these novel techniques, demonstrating the feasibility of a unified debugging environment for HLS-accelerated programs on heterogeneous systems. These techniques could be applied to any HLS tool, provided that it allows for compiler analyses of the program as a whole before splitting it up into hardware and software, and maintains a correlation between the source code and generated RTL.

**OpenCL Source-Code Level Debugging Environment for On-chip debugging**

The contributions put forth in Chapter 5, we focused on exploring on-chip debugging of OpenCL FPGA kernels in HLS-accelerated programs. As most commercial tools are moving more towards supporting heterogeneous systems, particularly with OpenCL, it is important to explore how previous techniques could be applied to an OpenCL environment. Additionally, we explored how much of a source-code level debugging environment for on-chip debug of OpenCL FPGA kernels could be automated using current open-source technologies. To address this, we developed techniques and a tool-flow which allows users to develop a source-code level debugging environment through modifying the source code directly, and synthesizing those changes into the generated hardware circuit, allowing it to run on any HLS tool.

The tool-flow was broken up into five steps, and techniques were presented to handle each of the steps. 1) The ROSE source-to-source compiler was used to identify all variable assignment operations in the kernels, and grouped them by function. 2) The list of variable assignment operations was passed to a variable selection GUI that allowed users to select variable assignment operations to record during execution. 3) The list of selected variable assignment operations was passed back to the ROSE compiler, where recording instructions were automatically inserted after each of the selected operations. The recording instructions record a globally unique ID, and the trace data. The globally unique ID is used to correlate the trace data with the source code post-execution. 4) The user modified the software source code to retrieve the trace buffer through a
kernel argument, and recorded it to a file. 5) The trace buffer was parsed and presented to the user in a user-friendly format that allowed users to dynamically create trace timelines for the selected trace data.

In addition to these steps we discussed the limitations and workarounds necessary in using the current version of the ROSE compiler, and addressed basic triggering and trace buffer caching. The ROSE compilers support for OpenCL is still in development, so we demonstrated how ROSE could support OpenCL through using Python scripts to remove OpenCL directives before passing it to ROSE, and re-inserting the directives after ROSE has finished. We then discussed how users could insert triggers of arbitrary complexity into their code to halt execution or the recording of data. Lastly, we discussed how trace buffer caching and memory bursting techniques would be used in our techniques.

To quantify the impact of these techniques on the OpenCL FPGA kernels, we measured and compared the execution time and hardware resource utilization of the baseline benchmark, the benchmark modified with our trace-based debugging framework, the benchmark modified to record variables during execution, and the benchmark modified to support trace buffer caching to record variables during execution. Under all measurements, with the exception of trace buffer caching, there was a modest impact to execution time and hardware resource utilization, 22% increase or less. Trace buffer caching had the opposite effect of what we expected. This is likely due to the added control-flow to support it in the main loops of the algorithm, which negatively affected the scheduling portion of the HLS tool.

The tool-flow and techniques presented demonstrated that trace-based debugging techniques for heterogeneous systems can be applied to a source-code level debugging environment for OpenCL FPGA kernels in heterogeneous systems, though there are still many limitations that arise. These limitations are centered around the lack of full support for OpenCL in source-to-source compilers, as previously mentioned, and the nature of synthesizing the recording instructions and triggers into the source code. Because the recording instructions and triggers are synthesized into the source code, they cannot be changed without re-synthesizing the hardware design. This is an open research question, and Section 6.2 will discuss it further. However, by inserting the recording instructions and the triggers at the source code level, and synthesizing them into the design, the techniques presented in Chapter 5 are agnostic to the HLS Vendor tool used in synthesis, and the
system the program is executed on. In addition to these limitations, we demonstrated that using
global memory to store the trace data did not result in excessively large overheads to execution
time and hardware resources. By using global memory we were able to retrieve the trace buffer as
a kernel argument at the end of program execution, rather than having to rely on external hardware
tools to retrieve the trace data.

6.2 Future Work

There are many possible research directions in developing techniques to support non-
traditional compute devices and accelerated programs using compilers, as they already are respon-
sible for a majority of the analyses, optimizations, and transformations users rely on. This sections
will focus on possible research opportunities we identified while developing and prototyping our
techniques.

6.2.1 Efficient Data Transfers

There are two areas of exploration in more efficiently scheduling data transfers:

The first is question is, are there more aggressive compiler analyses that can be applied
to our scheduling algorithms to further reduce the impact of data transfers? For example, one
possibility would be applying polyhedral analysis to the modified liveness analysis and scheduling
algorithms presented in Chapter 3. Polyhedral analysis is used to determine which portions of
arrays are accessed in iterations of a loop. This would allow for portions of the data arrays to be
transferred, rather than the entire array, when only part of the array is used. Researchers could look
into this, and other aggressive analyses to reduce the impact on data transfers.

The second direction we considered was a way to determine if transferring data and exe-
cuting on the accelerator is more beneficial then executing on the host. Due to the expensive cost
of data transfers, it is possible that the cost of transferring data, and executing on the accelerator is
less efficient than executing on the host. If we were able to determine this, compilers could repli-
cate the kernel code in the software, and execute the accelerated algorithm in either the accelerator
or host, depending on which is more efficient for the current system.
6.2.2 Unified Debugging Environment for Heterogeneous Systems

There are two research directions we identified for our unified debugging environment for heterogeneous systems. The first is focused on how we can record more data with less impact on execution and hardware and software resources, and the second is focused on how we can reduce the impact of synchronization on execution time.

Unlike supporting trace-based debugging in the hardware where it usually only requires extra resources, recording trace data in the software directly impacts execution time. This is due to the extra sequential instructions required to record data to the software trace buffer. How can we reduce this impact to software execution without reducing the amount of debug data recorded? There are a many directions this could go, but there are could we have thought to be more viable:

First is to find locations to record data that have less of an impact on execution time. Normal compiler optimizations move instructions around to reduce their overhead. For example, instructions that load values from memory are often moved to earlier locations in the program to prevent the instructions that use the values from having to wait for them to load. Similar to this, it might be possible to identify places in the program where instructions are waiting for others to finish to record data in. This could also include waiting for threads, memory operations, or I/O to finish. Hypothetically, by identifying these locations, and moving recording instructions to them, more data could be recorded with reduced impact on execution time.

The second direction we though of was can the hardware be used to reduce the impact on software. It might be possible to have the hardware retrieve data from global memory, and store it in a safe location. This would allow the software to continually execute without being affected by the recorded data. It might also be possible to reduce the effect on software by sending trace data from the software to the hardware, and have the hardware record it in global memory, rather than recording it directly in software.

In addition to reducing the impact on execution time, reducing the impact on hardware resources or recording more data with less resources is important. This has been an area research many have looked into [39], [41]. However, our unified debugging environment provides a distinct advantage over these, specifically the software is already recording data to a trace buffer. Because of this, researchers could explore the effects of regularly transferring debug data from the hardware to the software, preserving more of the hardware trace data.
Synchronization has a substantial impact on program execution time due to the cost of the software retrieving the synchronization ID from hardware. Requiring the software to cross domain bounds to retrieve the synchronization ID is acceptable if it is happening occasionally, but if it is happening repeatedly, it quickly becomes expensive. Researchers could look into ways to reduce this overhead. There are a couple of ideas we had in this regard:

The first idea was, can the management of the synchronization ID be shifted between the hardware and software to reduce the need to cross domain bounds? This would be highly beneficial when the software or hardware halt execution, such as in programs where the software pauses after a kernel is invoked, and does not continue until the kernel finished executing. By shifting management of the synchronization ID between the devices in these types of systems, the impact of crossing domain bounds to retrieve the synchronization ID could be virtually negated.

The second area of investigation is identifying other synchronization schemes that may be more beneficial to code structures. This could include using natural thread synchronization points to aid in synchronization, such as barriers, mutexes, explicit data transfers, or kernel invocation calls. Relying on other synchronization points, many synchronization calls could be eliminated, reducing the impact on execution time.

### 6.2.3 Source-code Level Debugging Environment for FPGA OpenCL kernels

There are multiple areas of research and engineering that could be addressed in our source-code level debugging environment for OpenCL FPGA Kernels in heterogeneous systems. As far as engineering, our techniques could be implemented in a source-to-source compiler that fully supports OpenCL. This would remove the workarounds needed to support it with ROSE, and allow for greater adoption. In terms of research, there are four areas of research that could be applied to this work: how trace data can be more accurately collected and organized, how can triggers be implemented at the source code level without the need to re-synthesize them, how can trace buffer caching be applied without the negative impact on scheduling, and exploring different recording techniques and their synthesized results to determine the best implementation.

Our techniques allow users to view a trace timeline of the selected variable assignment operations, however the "timescale" of this timeline is based upon the order in which they were written to the trace buffer, not on the cycle they were written to the trace buffer. The main draw-
back to this is it does not provide a cycle-by-cycle view of execution like traditional trace-based debugging tools. To address this, researchers could look how to provide a cycle-by-cycle view of the trace data. This could possibly done through adding high-precision timers to the hardware design in the HLS tool, or maybe through automatically modifying the generated RTL to add timer data to entries to the global trace buffer.

The approach we took in relation to triggers was to have users manually insert triggers into the kernel source code using if statements with return instructions. Though this works for stopping execution at specific times, is does not provide other common triggering features, such as allowing users to begin capturing data when the trigger is set, stopping data capture after a set number of cycles, or changing the trigger without having to re-synthesize the design. Researchers could look into how to add various types to triggers to the source code, how they are synthesized into RTL, and determine the most efficient implementations. Additionally, they could look into techniques that do not require the re-synthesis of the hardware design to update the trigger. One thought we had to support his was through using kernel arguments to determine which of many triggers should be used.

Trace buffer caching and memory bursting are techniques commonly used in OpenCL FPGA kernels to more efficiently transfer data to global memory. Traditionally, memory bursting is performed at set locations in the program where the buffer is known to be full. However, with the local trace buffer, we do not necessarily know at compile time when it will be full, resulting in control-flow added after each recording instruction to check if it is time to transfer the local trace buffer to the global trace buffer. This control-flow contains a branch that could execute for an unknown amount of time while data is copied from the local trace buffer to the global trace buffer, which prevents the synthesis scheduler from properly optimizing loops containing them. One area of research is to look for ways to efficiently use trace buffer caching in this type of environment without negatively impacting the synthesis scheduler. One option might be to have the complete thread’s trace buffer reside in the local buffer, and copy it over at the end of kernel execution. This would likely require smaller trace buffers per thread, but could substantially reduce the impact to execution time.

The last area of research is analysing the synthesis of the recording instructions into trace insertion logic to develop better source code modifications. The recording instructions inserted into
the source code are synthesized into RTL using the desired HLS tool, and impact both execution
time and hardware resources. Researchers could analyze the generated RTL using different types of
recording instructions to identify those which have the least overhead after synthesis, allowing the
synthesized recording instructions to have overhead similar to traditional trace-based debugging
circuitry.
REFERENCES


