YOUR THESIS TITLE

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Cale William Glisson

December 2015

© 2015
Cale William Glisson
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Your Thesis Title
AUTHOR:	Cale William Glisson
DATE SUBMITTED:	December 2015
COMMITTEE CHAIR:	Professor John Oliver, Ph.D. Department of Electrical Engineering
COMMITTEE MEMBER:	Professor Andrew Danowitz, Ph.D. Department of Electrical Engineering
COMMITTEE MEMBER:	Professor Chris Lupo, Ph.D. Department of Computer Science

ABSTRACT

Your Thesis Title

Cale William Glisson

Your abstract goes in here

ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer systems are becoming more heterogenous in their nature, including not just a typical microprocessors, but also hardware accelerators in some form or another. This allows system designers to pick and choose which portions of their program will be accelerated by the hardware, while keeping non-critical components simple.

One option in these systems is to use a Field Programmable Gate Array (FPGA) for the hardware portion, which allows for a Hardware Description Language (HDL) to be used to implement the hardware, rather than using discrete hardware accelerators. This means that in addition to the typical C or C++ code for the microprocessor, HDL code must be written for the FPGA acceleration as well as an interface between these two portions. While this gives the design incredible flexibility and control, the interface between the hardware and software is incredibly complex, and can create almost impossible to debug errors in any non trivial system. This often causes longer development cycles, and requires extremely specialized developers which means that less efficient solutions are chosen over this path.

Twill was created to simplify the development cycle, while also taking advantage of parallelization to increase performance. Twill takes a single threaded C code as an input, transforms part of that C program into hardware, and also provides the communication system between the hardware and software.

As Twill is solving such a complex problem, it in itself presents a complex tool chain, consisting of many individual portions, some automated, others not. With a lack of proper documentation of the system and the workflow, the toolchain becomes almost impossible to use properly and many issues arise.

The major contribution to the Twill project here will be providing that documentation, and elaborating more on some subtle details that were not discussed in the original Twill paper.

The remainder of this thesis is organized as follows: Chapter **??** will provide some information on the previous work done in regards to Twill. Chapter **??** will explain the intended work flow of Twill, while Chapter **??** will explain the issues that come up in that work flow, as well as any solutions to those issues that exist. Chapter **??** will discuss future work that needs to be done to Twill to get it working in it's original state, as well as suggestions on how to improve the tool.

Chapter 2

Previous Work

This section will give an overview of the design of the Twill compiler, and then go in depth into the differing components of it.

2.1 Twill

Twill was designed to take a single threaded C program and extract both Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP) in order to take advantage of the tightly coupled nature of a CPU/FPGA hybrid system.

2.1.1 Twill Dependencies

Twill takes advantage of a large amount of previous work. It uses a modified version of Distributed Software Pipelining (DSWP) [TODO: DSWP SOURCE] in order to find and extract TLP. It also relies on LegUp [TODO: LEGUP SOURCE] for finding the ILP in the threads extracted by DSWP and then to translate those threads into HDL. Both LegUp and Twill's custom DSWP implementation are implemented as extensions for the LLVM Compiler Framework [TODO: LLVM SOURCE]. Twill also uses a custome runtime system based on the hThreads project [TODO: hThreads source]. It also currently uses Xilinx ISE and SDK in order to synthesize all of the

hardware portions, and to build the C code to a Microblaze target.

The specific implementation for each of these sections within Twill can be seen in the original Twill paper [TODO: TWILL SOURCE] but a general overview will be provided here to give context.

2.1.2 Twill Compiler Architecture

The Twill Compiler was designed to simplify the DSWP implementation and the Twill compiler pass. As such, the compiler itself is a patchwork of other work along with some custom compiler passes. This can be seen in Figure ¡TODO: INSERT FIGURE 5.1 FROM DOUG¿ where each block is a different tool used to transform the input C program into two linked programs, one in C and the other in Verilog.

Chapter 3

Toolchain Usage

The twill toolchain is made up of complex pieces of software, each communicating with the others. As such the workflow involved in getting through the tool chain is not negligable just yet.

3.1 LLVM Compilation

The compiler portion of Twill is mostly all automated by a python script, that takes in only a few options.

3.1.1 Running the Python Script

Figure [[TODO: INSERT FIGURE SHOWING EXAMPLE OF PYTHON EXECUTION]] shows an example invocation of the script. The second argument is the target C file that you want to run through the toolchain. This file should contain the "main" function for the program in order for the toolchain to compile correctly. The first argument to the script is the number of partitions you want created. Currently, using 1 for this parameter will not generate any hardware, while using 2 will generate the hardware partition. In theory higher numbers will still function, generating more hardware partitions, though this functionality has not been tested, and may require

additinonal changes elsewhere.

The python script takes the input file, and then runs it through the compilation path that is described in the original Twill paper [TODO: INSERT TWILL REFERENCE HERE]. After running through all the LLVM passes, the python script will copy the generated software and hardware files into the appropriate location of a Xilinx project.

3.2 Xilinx

After running the python script runs, the rest of the tool chain lies within Xilinx ISE, which is responsible for synthesizing the hardware, and uploading and running the code on the actual hardware.

3.2.1 Xilinx ISE

After running the python script, Xilinx ISE can be opened up to the "uHTheadsCustom" project. This project contains a pre-generated MicroBlaze processor, the hardware software interface HDL, and should already include the generated hw file. The next step in the process is to synthesize the top level module, the uHThreads object. This will synthesize the generated hardware, the processor, and the glue logic between the two generating a bitstream. After synthesis is complete, the hardware design should be exported to Xilinx SDK with the bitstream.

3.2.2 Xilinx SDK

Exporting the bitstream will open up the Xilinx SDK, which is based on Eclipse. There should be 2 projects open here, one that contains the BSP for the target FPGA device, and the uHThreadsCode project, which is where the software portion from the python script was copied earlier. In main.c, the main function initializes the Twill runtime, starts up the hardware threads, and then executes the program. After execution the number of cycles taken to execute the program and the return value

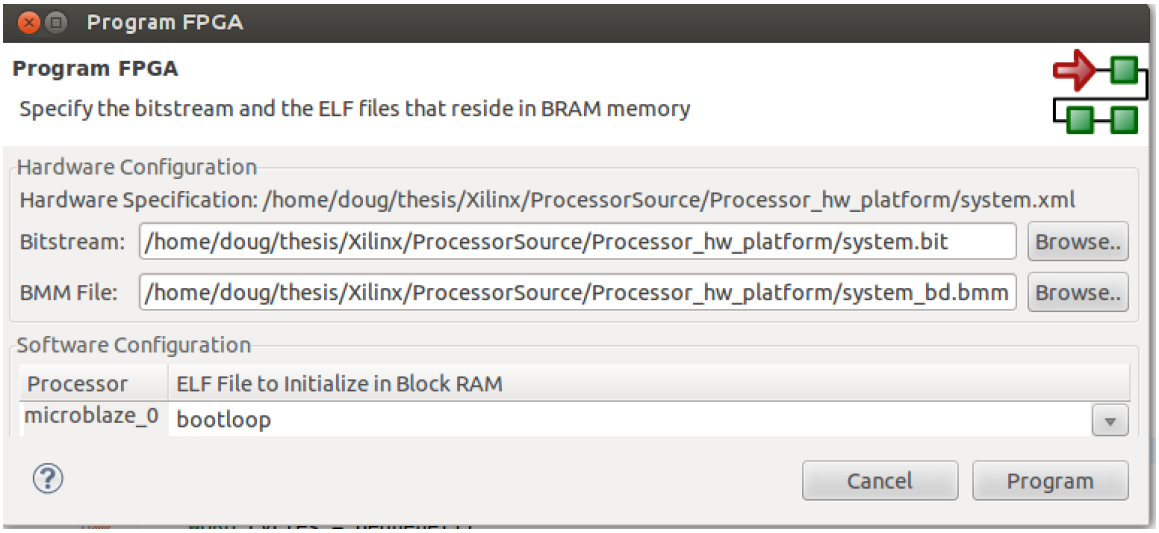

Figure 3.1: Program FPGA Dialog Box

from the "main" function in the original program will print over a Serial connection.

To actually program the FPGA, ensure that the hardware is connected to the host, typically through a JTAG connection. Click the "Program FPGA" which will display the dialog shown in Figure **??**.

Ensure that the ".bit" file and ".bmm" file correspond to the correct files in the board support project, and then click program to upload the program to the FPGA. After the program is finished uploading, you should be able to click the debug button which will launch the debug view and allow you to debug, or run through the program as necessary.

Chapter 4

Toolchain Issues

While the toolchain should be used in the way described in the previous chapter, there are some issues that exist in the current implementation of the toolchain that will be outlined here.

4.1 LLVM Compilation

When running the python script, a number of errors that prevent compilation do come up. First, the LLVM compilation itself can fail due to a bug in the DSWP algorithm that causes loop headers to be used as exit blocks, an example of this error can be seen in Figure **??**. This is likely due to missed cases when performing the partitioning of the program graph into hardware and software. The exact location of this oversight is still unknown.

In order to progress past this issue in the toolchain, the amount of the program going into hardware can be configured to be a different amount, which moves where

```
Assert(basicBlock->getTerminator()->getNumSuccessors() <= 1 && "Exit block is a
loop header...") failed in file /home/doug/thesis/llvm-2.9/lib/Transforms/DSWP/D
SWP.cpp, line 877
```

Figure 4.1: Assertions Failing Within DSWP Algorithm

the partitioning occurs and can allow the compilation to complete successfully.

Chapter 5

Conclusion

This is some concluding text