TWILL: A HYBRID MICROCONTROLLER-FPGA FRAMEWORK FOR PARALLELIZING SINGLE-THREADED C PROGRAMS

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 Computer Science

by

Douglas Gallatin

March 2014

COMMITTEE MEMBERSHIP

TITLE:	Twill: A Hybrid Microcontroller-FPGA Framework for Parallelizing Single-Threaded C Programs
AUTHOR:	Douglas Gallatin
DATE SUBMITTED:	March 2014
COMMITTEE CHAIR:	John Oliver, Ph.D. Professor of Electrical Engineering
COMMITTEE MEMBER:	Christopher Lupo, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science

Abstract

Twill: A Hybrid Microcontroller-FPGA Framework for Parallelizing Single-Threaded C Programs

Douglas Gallatin

Increasingly System-On-A-Chip platforms which incorporate both microprocessors and re-programmable logic are being utilized across several fields ranging from the automotive industry to network infrastructure. Unfortunately, the development tools accompanying these products leave much to be desired, requiring knowledge of both traditional embedded systems languages like C and hardware description languages like Verilog. We propose to bridge this gap with Twill, a truly automatic hybrid compiler that can take advantage of the parallelism inherent in these platforms. Twill can extract long-running threads from single threaded C code and distribute these threads across the hardware and software domains to more fully utilize the asymmetric characteristics between processors and the embedded reconfigurable logic fabric. We show that Twill provides a significant performance increase on the CHStone benchmarks with an average 1.63 times increase over the pure hardware approach and an increase of 22.2 times on average over the pure software approach while reducing the area required by the reconfigurable logic by on average 1.73 times compared to the pure hardware approach.

Keywords: Embedded Systems, Computer Architecture, Compilers

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Increasingly it is becoming common for Field Programmable Gate Array (FPGA) manufacturers to embed microprocessors within the FPGA fabric. This allows developers on such systems to pick and choose which parts of their application require the speedups achievable by being implemented in hardware while maintaining a faster development/debug cycle for the majority of the (nontime-critical) code.

The development cycle for these kinds of hybrid systems has thus been writing assembly, C or C++ code for the microcontroller and Hardware Description Language (HDL) code for the surrounding FPGA logic framework and then manually specifying the interface between the two code sections. While this paradigm gives the developer flexibility and control, the complexity of the HW/SW interface leads to many hard-to-debug errors in all but the simplest of systems. In turn this leads to longer development cycles and requires more experienced, specialized developers which often pushes many potential products to use less efficient solutions.

Twill is designed to simplify this development cycle while simultaneously exploiting latent parallelization to increase performance. In particular, Twill is a compiler that takes single-threaded C code as input, extracts long running threads from that C code, transforms some of the threads into hardware, and then pro-

vides a runtime communication system for a hybrid CPU/FPGA System-On-A-Chip.

In this way Twill is able to take advantage of both Instruction-Level Parallelism (ILP) and Thread-Level Parallelism (TLP) that may be present in the original source while not requiring any special notation or assistance from the developer. Twill attempts to efficiently utilize all parts of the hybrid system by automatically balancing and redistributing the workload in a transparent fashion.

The major contribution of Twill is to integrate algorithms for ILP and TLP parallelism into an environment suitable for small, low-powered embedded systems. It combines the strengths of previous hybrid runtime systems with the abstraction provided by High-Level Synthesis (HLS) systems while exploiting a higher degree of the parallelism inherent in the input program. Thus, it is able to give very large performance speedups for these kinds of hybrid embedded systems without requiring the programmer to have any knowledge of HDL.

The remainder of this thesis is organized as follows: Chapter 2 provides a history of hybrid systems and highlights several state-of-the-art hybrid projects. Chapter 3 then presents a broad overview which is followed by an in-depth description of the runtime architecture and the compiler architecture in Chapters 4 and 5 respectively. Twill's performance results are next discussed in Chapter 6 while future work on Twill is presented in Chapter 7.

Chapter 2

Previous Work

There are two broad research areas that this thesis attempts to bridge. First presented is the research dealing with hybrid CPU/FPGA systems. This is then followed by research in automatic thread extraction for compilers.

2.1 Hybrid Systems

One of the foundational papers for the reconfigurable computing domain is the PRISM project [24]. This project essentially connected a 10MHz M68010 processor to two Xilinx 3090 FPGAs with a 16-bit bus. At compile time the programmer selects which functions to implement in hardware and then at runtime the function arguments are passed to the FPGAs and then the result is passed back.

From the PRISM project spawned a multitude of reconfigurable-pipeline processors where the researchers attempted to remove the large data transfer costs that plagued PRISM. Recent work by Lauwereins et al. [3], Galuzzi and Bertels [11], and Vassiliadis et al. [35] represent the state-of-the-art with this approach where the processesor's pipeline itself is reconfigured for the particular problem.

In parallel, projects such as GARP [16] and OneChip [37] continued with an

architecture more closely associated with a processor attached to an FPGA co-processor. GARP, like most of the successors to the PRISM project, focused on optimizing the underlying architecture and bus layout and relied on the programmer to write code in both traditional languages such as C and HDL in order to take advantage of the platform. In particular, the GARP project was the first to put a hard processor and FPGA framework on the same silicon die. In contrast, projects such as OneChip embedded soft processors within the FPGA logic.

Once suitable hardware architectures were ironed out, several projects such as NAPA C [13] began to focus on how to compile code for these hybrid systems. NAPA C in particular created a new dialect of C designed to be easily synthesized and then used pragmas to allow the programmer to specify which variables were to be stored where and which logic blocks needed to be synthesized to the FPGA. They were able to accomplish this with a tightly integrated hardware/software platform that allowed for single-cycle latency transfers to and from the processor register file. The NAPA C compiler is also able to do limited automatic loop parallelization on the FPGA for DO-ALL loops which is the first attempt at automatically exploiting latent TLP with a hybrid architecture.

Other hybrid compilers have been created but for the most part current work on compilers in this field has moved into the runtime-reconfiguration space with projects such as work by Bergeron et al. [4], Koch et al. [20], and Purnaprajna et al. [30]. Papadimitrious et al. [28] provide a comprehensive cost-benefit analysis of these kinds of projects. In general it seems as though the current hardware platforms aren't really designed for easily partitioning and parallelizing a program across the HW/SW domains and as such these approaches tend to have extremely high overhead in exchange for allowing multiple programs unknown at system start time to run at once on the hardware or for the operating system to dynamically synthesize and execute oft-run parts of programs on the FPGA co-processor.

Still other work such as CHiMPS [31] and ROCCC [36] have attempted to utilize hybrid systems in order to allow full ANSI-C compliant code to be run on the FPGA. CHiMPS in particular utilizes the processor to take care of tradi-

tionally difficult synthesis problems such as recursion and function pointers. In contrast, other ANSI-C compliant HLS tools such as LegUp [7] will synthesize a processor in the FPGA framework to solve these problems. In general, all of these HLS tools will synthesize the entire source except for the problematic parts and while they have become very efficient at taking advantage of ILP they do nothing to attempt to extract TLP from the input programs.

Recently, work has been published about how to determine which portions of a program would be best for synthesizing into hardware given the unique area verses performance trade off that such a transformation makes. In particular, Koehler et al. [21] look at common programs where the most-executed code is prohibitively large when synthesized in an FPGA and presents a framework to maximize the performance/area product rather than solely the performance of the system. Martin et al. [25] take a different approach to solve the problem by applying past work on the NP-complete box-packing problem to determine which processor extensions to synthesize. Similarly, Curreri et al. [10] present a framework for measuring the performance and area of synthesized code from various High-Level languages in order to attempt to quantify the trade off between useful high-level language constructs and resulting code efficiency.

In the last decade several projects that attempt to provide an Operating System (OS) or Real-Time Operating System (RTOS) have come to fruition. In particular, the work by Agron and Andrews [2], FOSFOR [12], ReconOS [23], and hThreads [29] all provide operating services uniformly accessible across the HW/SW domains. The different projects have explored different tradeoffs between the HW/SW domains. For example, the ReconOS kernel is implemented entirely in software while the hThreads kernel is implemented entirely in hardware and the other two are a hybrid. None of these projects include any sort of automatic code partitioning and instead force the developer to explicitly create software threads and hardware "threads" and manage all communication between them using the OS resources.

Finally, several projects in the last several years such as LegUp [8], Spark [14], and Liquid Metal [17] have attempted to put a run-time OS-like system

together with a code-partitioning compiler in order to more automatically take advantage of these hybrid systems. Liquid Metal introduces a new Java-based object-oriented language that allows the programmer to interact with object instances across the HW/SW domains but requires the programmer to keep track of which objects are where.

LegUp and Spark both implement a compiler/translator for traditional C programs. LegUp was originally only an HDL translator but has recently added limited support for calling functions across the HW/SW domains. They have a basic automatic heuristic but encourage the programmer to annotate each function with whether that function should be implemented in HW or in SW. LegUp does not do any sort of Thread-Level Parallelism (TLP) but does implement a modulo-scheduler for Instruction-Level Parallelism (ILP). Also, LegUp does not provide any primitives other than the function call for synchronization and communication which makes it extremely difficult for the programmer to implement truly parallel code.

In contrast, Spark started with a similar system to LegUp and then focused on implementing code optimizations in order to achieve speedup. With complete control over the hardware, they were able to implement several different speculative-based optimizations with very little overhead. However, they focused almost entirely on ILP parallelization techniques at the expense of TLP parallelization.

2.2 Automatic Thread Extraction

One of the seminal papers in the computational theory of parallel programs is the Actor model [1]. This model constructs all units of computation as an actor that can do work, send messages, or spawn more actors in response to a message received. Projects such as ReconOS [23] and hThreads [29] along with Twill extend this model to essentially heterogeneous actors.

More recently, many attempts have been made to take a single-threaded pro-

gram and automatically extract long-running threads from it. Some research such as the work done by Cordes et al.[9] rely on traditional methods such as linear programming. Others such as Thies et al. [33] focus on particular applications where the problem can be easily seen as a stream of data with different modules transforming the data in some way. Some of the more ambitious projects such as the Helix project [6] control the entire stack from hardware architecture to compiler to OS. Decoupled Software Pipelining (DSWP) [27] and Kejariwal et al. [19] both focus solely on parallelizing loops; in particular they both attempt to minimize the communication overhead required to pass data around between processors. In this case, the work by Kejariwal et al. is applicable to far fewer loops than the paradigm afforded by DSWP.

Most new work in the field revolves around the idea of speculative multi-threading such as first described without significant hardware support by Oplinger et al. [26]. This work purposefully breaks data or control dependencies in the program dependence graph by making assumptions about their values in order to more efficiently parallelize the program. The program must further keep track of these dependencies such that if any assumption is broken the program must reset itself and proceed with the new correct assumptions. Most all of "regular" thread extraction algorithms may be adapted to a speculative model with limited hardware support. Some use weighted control flow graphs and heuristics as Pan et al. [39] while still others are based off of the idea of pre-computation slices like Quiones et al. [32] or the min-cut algorithm described by Johnson et al. [18]. The DSWP algorithm was explicitly adapted for speculation by Vachharajani et al. [34]. In general, all of these methods successfully achieve performance increases under many situations where the non-speculative algorithms fall short.

Chapter 3

Twill Overview

Twill is a compiler and runtime system designed to extract latent Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP) from a single-threaded C program in order to exploit the unique characteristics of a tightly coupled CPU/FPGA hybrid system architecture.

3.1 Twill Dependencies

Twill takes advantage of a great deal of previous work: Twill uses a modified version of Distributed Software Pipelining (DSWP) as first presented by Ottoni et al. [27] in order to find and extract long running threads. Twill relies upon LegUp [7] for finding ILP in the extracted threads and for translating those threads into HDL. LegUp and Twill's custom compiler passes are both extensions for the LLVM Compiler Framework [22]. Finally, it uses a custom runtime system/RTOS heavily influenced by the hThreads [29] project.

3.1.1 DSWP

The DSWP algorithm original presented by Ottoni et al. [27] was re-implemented from scratch in LLVM. The DSWP algorithm itself consists of two parts: the generation of a Program Dependence Graph (PDG) and the splitting of the original program flow into multiple pipelined threads. The PDG is a per-function graph that keeps track of all dependencies between instructions. Each node in the PDG is an instruction in the function and each edge is a dependency originating from a "tail" instruction node and terminating on a "head" instruction node. Thus each edge designates that the tail must execute before the head. There are three different kinds of dependencies: data dependencies which designate that the head relies on the data generated from the tail, memory dependencies where the head and tail instructions read-write to the same memory locations and the execution order must be preserved, and control dependencies where the tail is a conditional branch that determines whether the head is executed or not. On top of these traditional dependencies tracked in the PDG, the DSWP algorithm requires several more edges to ensure correctness in several cases described in detail in the original paper by Ottoni et al. [27].

Once the PDG is generated, the DSWP algorithm assigns instructions to multiple threads such that all instructions in a given Strongly Connected Component (SCC) in the PDG are all assigned to the same partition. Furthermore, all cross-partition edges in the partitioned PDG must form no cycles. These two requirements essentially ensure that the threads create a pipeline or chain of threads were data only flows one direction along the chain. In this way threads are "decoupled" from each other such that data transfer latency between threads is more or less inconsequential to the runtime of the system.

The original DSWP algorithm relies on synchronized hardware queues to transfer data between threads. Two new instructions are introduced, `produce` and `consume`, that each place or remove an 8 bit unsigned integer to or from a queue. These instructions only block when the queue is full or empty and otherwise can be executed in one cycle. Ottoni et al. tested their algorithm through

simulation on an Itanium 2 architecture with the IMPACT compiler.

Overall, Ottoni et al. found that while they did get speedups with the DSWP approach, they essentially traded ILP for TLP since the Itanium processor was inherently quite efficient at exploiting ILP. However, they found that DSWP provided much better results with simpler non-VLIW processors and suggested that the benefits of DSWP would increase the simpler the processor became. Furthermore, they found that the queue latency and queue sizes did not appreciably affect runtime. Note that in their tests they only applied DSWP to a single hand-picked loop in each benchmark and did pipeline any function calls.

Details of our implementation of the DSWP algorithm can be found in Section 5.2.

3.1.2 LegUp

LegUp [7] is an open-source HLS compiler based off of LLVM that is being actively developed by the University of Toronto. In addition, LegUp provides a full hybrid design flow for FPGA designs with a soft processor. They support special pragmas to help with code partitioning along with a comprehensive profiling suite that measures area, power consumption, and performance in both the FPGA logic and soft core. They have many optimizations to exploit ILP in various ways in hardware. LegUp first builds a PDG for each function to be translated and then partitions the graph into stages of instructions where each stage contains instructions that can be potentially executed at once. Programmer-controlled heuristics are then used to adaptively control how many functional units to include for each stage which provides a trade-off between circuit size and exploiting ILP. LegUp also implements a version of iterative modulo scheduling which allows it to pipeline across loop iterations.

LegUp has two different compilation flows, one for their hybrid runtime system and one for a pure hardware HLS translation. The hybrid flow allows for communication between a number of soft processors and the hardware circuit at function boundaries but highly encourages the programmer to specify which

functions should go where. By default, LegUp will only place problematic functions on the soft cores such as functions containing recursion, function pointers, or other traditionally difficult HLS constructs. In this way, the hybrid flow of LegUp is ANSI C compliant. However, Twill makes use of the non-hybrid design flow of LegUp which does not support these constructs.

Overall, LegUp's performance and area results for the hybrid flow are very poor compared to other commercial HLS tools such as eXCite [38] while LegUp's pure hardware flow performs very similarly and in some cases vastly outperforms other commercial HLS tools.

Details of how Twill uses LegUp can be found in Section 5.4.

3.1.3 LLVM

LLVM is a popular compiler and runtime framework designed to "make lifelong program analysis and transformation available for arbitrary software in a manner that is transparent to programmers" [22]. LLVM achieves this through two major areas: a specially designed Intermediate Representation (IR) language and a comprehensive runtime and profiling framework for directly interpreting this IR in a platform agnostic manner.

LLVM's IR is very close to assembly but one key aspect that makes program transformation easier than in other compilers is that this representation is in Static Single Assignment (SSA) form which means that every variable or virtual register is assigned to only once in the entire program. Special instructions that are only allowed at the beginning of each basic block called PHI instructions are used to assign different values to a register depending on the control flow used to reach that basic block.

LLVM also provides a number of modular analysis and transform passes that make implementing custom program transformation passes easier such as the memory alias analysis, loop analysis, and dominator/postdominator tree information.

Details about how Twill uses and integrates with LLVM can be found in Section 5.1.

3.1.4 hThreads

The hThreads project [29], short for hybrid threads, is essentially an RTOS where OS primitives can be implemented in either hardware or software. The project has a software and hardware version of most primitives such as the scheduler, memory allocator, and semaphores which are explicitly switched between by the application developer depending upon his needs. hThreads uses virtual memory-mapped registers to provide communication between the hardware and software modules and provides a uniform hardware interface for implementing hardware-based "threads" in VHDL. With this setup, the application developer can write software threads in C and hardware threads in VHDL and easily interface between threads using standard operating system primitives provided by hThreads.

hThreads also provides some limited support for HLS translation by using the HIF compiler [5] which they have modified to fully support their RTOS interface. Using this tool, application developers can call the RTOS software APIs which are automatically translated into the appropriate hardware thread interface semantics in VHDL.

3.2 Twill

Twill conceptually consists of three different parts: the compiler, the software runtime system, and the hardware runtime system. An overview of how these fit together can be seen in Figure 3.1. The Twill compiler first takes as input one or more C files describing a single-threaded program. The Twill compiler then performs thread extraction and hardware translation to output two stand-alone programs: one in C to run on the processor and one in Verilog to by synthesized

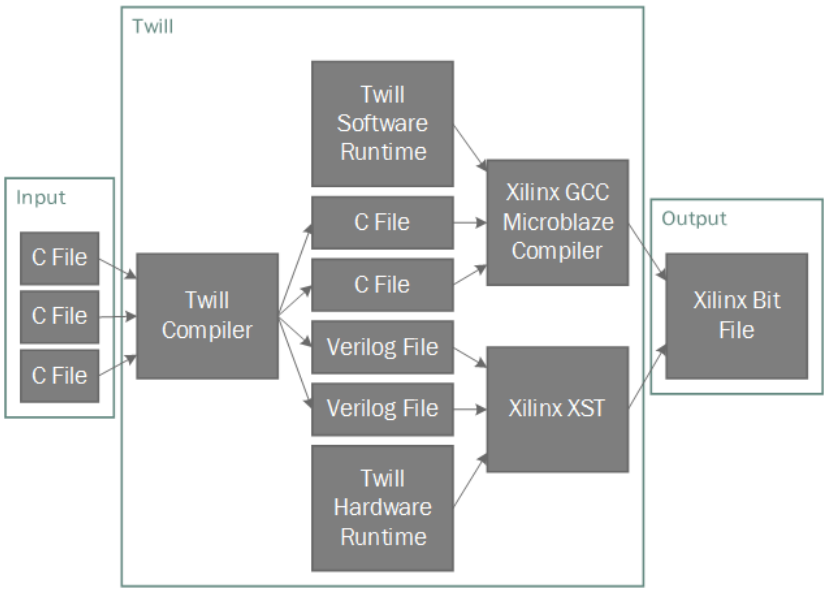


Figure 3.1: Twill Overview

onto an FPGA. These programs rely on the Twill Software and Hardware runtimes which are combined with the C and Verilog programs by Xilinx tools to produce a bitstream suitable for downloading onto an FPGA.

3.2.1 Twill Compiler

The Twill compiler is described in detail in Chapter 5. Internally it is implemented as a transform pass on top of LLVM and then uses LegUp to translate the hardware portions into Verilog. It also sets parameters for the statically defined primitives in both the software and hardware runtime systems.

While conceptually the Twill compiler could be ANSI C compliant, it currently has the same limitations on the input C files as LegUp: no recursive functions or function pointers. While this simplifies our implementation, Chapter 7 expands on how Twill could be extended to support these two constructs.

3.2.2 Twill Software Runtime

The Twill software runtime system is written in C and assembly. It contains an API for interfacing the processors with the hardware runtime. The Twill compiler generates C code for the processors with calls to these APIs in order to perform initialization, thread management, synchronization, and communication. The Twill software runtime system is described in more detail in Chapter 4.

3.2.3 Twill Hardware Runtime

The Twill hardware runtime system written in Verilog provides synchronization and communication primitives for the software and hardware threads. The generated Verilog modules from the Twill compiler include "calls" to the various hardware primitives to provide synchronization and communication with the other Verilog modules and the software threads running on the processors. Chapter 4 describes in depth the implementation details of the Twill hardware runtime system.

Chapter 4

Runtime Architecture

Twill's runtime system is heavily influenced by the hThreads project [29]. The runtime system has several primitives: semaphores, queues, software threads, hardware threads, and a simple scheduler. All of the primitives are statically configured at compile time with the exception of software threads which can be dynamically created. Semaphores, queues, the scheduler, and the hardware threads are all implemented in the FPGA logic in Verilog. Hardware threads are able to interact with semaphores, queues, and the processor's memory without interrupting the processor while software threads have minimal API wrappers to interact with the hardware primitives. The entire architecture overview is shown in Figure 4.1. Each block in Figure 4.1 represents a Verilog module while each edge represents instantiating a module. Note that the Twill module is the top-level module and does not contain any implementation details but simply defines global parameters used by most other modules.

The following subsections discuss the individual primitives after describing the bus addressing system.

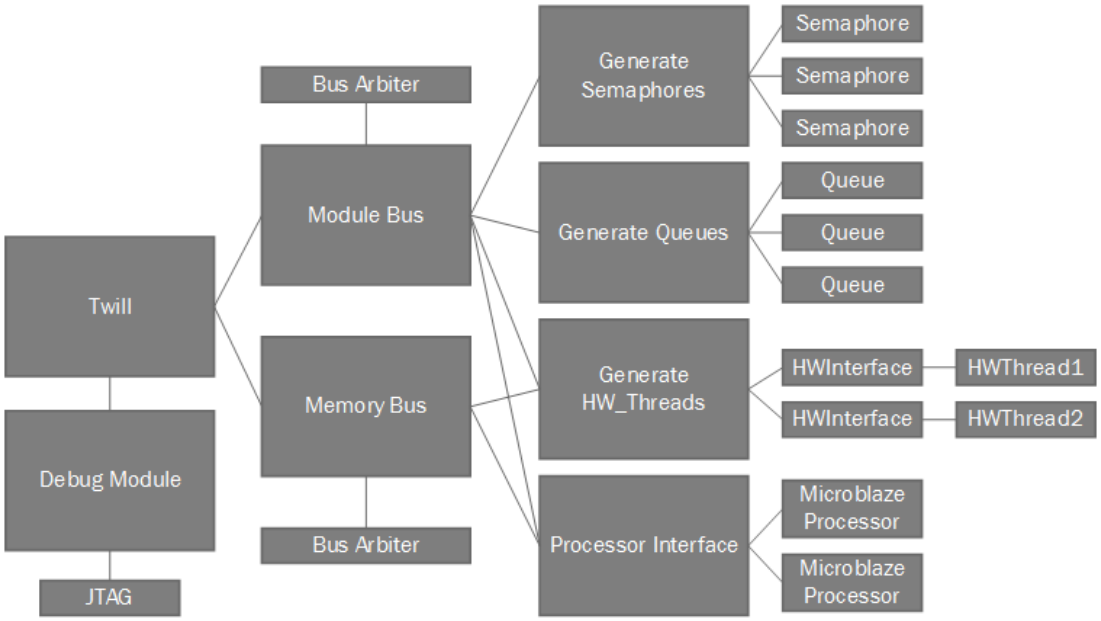


Figure 4.1: Twill Run-Time Hardware Architecture Overview

4.1 Bus Architecture

There are two main communication busses in Twill's runtime system that tie all of the primitives together. The first bus, the Module Bus, is the main communications link between all of the primitives used for passing messages. The second bus, the Memory Bus, is tied to each of the hardware threads and the processor interface module and gives the hardware threads access to the processor's memory space. The two busses are hierarchical with the Generate blocks shown in Figure 4.1 used to decrease the combinatorial logic at each stage allowing for higher clock frequencies.

Both busses work on a message passing model. Each primitive is assigned a unique address for the busses. When a primitive needs to send a message, it signals to the bus arbiter and for each clock cycle the bus arbiter will specify which primitive has control over the bus along with that primitive's bus message. This is designed in such a way that if there is no contention for the bus among the primitives, a primitive's signal will be acknowledged and its message available on the very next clock cycle. Thus, the bus has a latency of one clock cycle and

a throughput of one message per clock cycle.

The bus arbiter is implemented as a modified priority decoder which always gives priority to the processor if it is signaling and then gives priority to any primitive sending a message to the processor and finally gives priority to the primitive who has been signaling for the longest number of clock cycles. This is because the processor interface with hardware tends to be the critical path since the processor is slower at executing instructions. Furthermore, since the processor generally takes longer to perform a task than the pure hardware threads, it tends to signal the bus less frequently and when it does the system is designed such that the processor's pipeline should not be stalled at all waiting for the hardware primitive to respond.

A message on the main message bus consists of the destination address, the 3-bit message operation, and a 32-bit data field. The destination address is variably sized depending on the number of primitives. There are five operations: `give`, `take`, `start`, `stop`, and `ack`. Most primitives only accept a subset of the operations and the effect of the operations vary depending on what type of primitive is located at the destination address. The primitive specific effects from these operations are described in the following sections.

The memory bus uses the same model and timing characteristics of the main bus but is used solely to allow the hardware threads to read and write processor memory. A write takes one cycle while a read takes two cycles assuming no bus contention. One hardware thread may read/write to this bus at once completely asynchronously from what the processor is doing. Writes to memory from either the processor or from the hardware threads take two cycles to appear in the other domain.

4.2 Semaphores

The semaphore primitives are basic counting semaphores. Each may have a different max count and starting counter. A `give` message to the semaphore will

raise the semaphore while a `take` message corresponds to a lower. The data part of the message specifies how many times to raise or lower the semaphore. The semaphore will respond to the calling primitive's address with an `ack` message when that primitive has successfully taken the semaphore. When the semaphore is not locked the `ack` message will occur immediately on the next clock assuming no bus contention. If the semaphore's counter is already at zero then the semaphore will wait until a `give` message is received. The semaphore will then send `ack` messages first to the processor and then to the primitive that has been waiting the longest. In general, it is safe to send `take` messages to the semaphore from any primitive although it is not safe to send multiple `take` messages from the same primitive without receiving a corresponding `ack` message in between each message.

With the above architecture, the sending thread will be blocked for one cycle for a raise operation and a minimum of two cycles for a lower operation.

4.3 Queues

The queue primitives are first-in-first-out (FIFO) queues. Each may have a different max length and be either 1 bit, 8 bits, 16 bits, or 32 bits wide. The queues are asynchronous but assume that a single primitive is enqueueing data and a (potentially different) single primitive is dequeueing data. Thus a semaphore or other synchronization method between primitives must be used if more than one primitive is enqueueing the data at once or more than one primitive is attempting to dequeue the data at once. A `give` message to the queue enqueues the message's data field to the queue. An `ack` message will subsequently be sent back to the sender. A `take` message will cause the queue to send an `ack` message back to the sender with the dequeued value. Internally, the queues are implemented as a circular buffer with one more data element than the queue can hold. On enqueue operations, an `ack` message will be sent back to the sender immediately as long as the final data slot in the queue is empty. When the size+1 data slot is filled, an `ack` message will not be sent until a dequeue operation is

performed. In this way the sending primitive is stalled if the queue is full. Similarly, if the queue is empty then the queue will only send the `ack` message for a dequeue operation after a `give` message is received.

The synchronization overhead of enqueueing or dequeueing from a queue is thus a minimum of two cycles assuming no bus contention.

4.4 Hardware Threads

Hardware threads are user written or auto-generated HDL code that perform the desired computations. They have a simple interface to the HWInterface modules which deal with the specifics of communicating over the busses. For the hardware thread to perform any action, it sets the specified function code and the desired target along with any data parameters and then sends a pulse on a signal wire to the HWInterface. The HWInterface module then will latch in all the data and make the appropriate call. Note that the desired function code is just the equivalent to an enum where each function call has its own entry. The function code does not correspond to bus operation but uniquely specifies whether to perform an enqueue, dequeue, raise, lower, load, store, etc. operation. Furthermore, the desired target is not the same as the address but rather an index into a virtual array of the OS primitive implied by the function code. For example, passing zero as the desired resource to a raise call will raise the first semaphore while passing zero as the desired resource to an enqueue call will enqueue to the first queue. Multiple calls to different primitives may be made at once; the only constraint is that only one call may be initiated per cycle.

Each call will "return" to the hardware thread by the hardware interface specifying the code and resource on the return wires along with any data that might have been returned on the incoming data wires. In this way one function call per cycle may return to the hardware thread. The operations that "return" immediately on the next clock cycle assuming no bus contention are memory store, semaphore raise, start thread, and stop thread. Operations that take

multiple cycles are memory load, semaphore lower, enqueue, and dequeue. The HWInterface can also signal to the hardware thread that another thread started or stopped it asynchronously to any pending requests.

The HWInterface module connecting the HWThread modules to the Generate HW Threads block in Figure 4.1 is responsible for managing all of the simultaneous requests and their response states. It is designed in such a way as to not add latency between the hardware thread's operation request and sending messages out on the bus and thus the hardware thread has the minimum cycles listed in the other sections of synchronization overhead.

There are several special system hardware threads that handle some system-related tasks. The first is the I/O manager which is connected to the serial port and all of the external interrupt pins, reset signals, LED's, and switches. Other threads can send messages to this thread to interact with the I/O ports. Interrupts are forwarded with one clock cycle latency to the appropriate handler either in hardware or on the processor.

The second special system hardware thread is the timing thread which is used to time all of the cycle counts referenced in Chapter 6. The final special hardware thread is the scheduler. The scheduler is a simple round-robin scheduler for the software threads which can handle threads in both blocked and waiting states. Every period it will interrupt the processor with the new SW thread ID to switch to. It also snoops on the message bus looking for the active thread to become blocked in order to switch out threads. Since all of this logic is in hardware, the only critical-path cost on the processor is a single context-switch unlike traditional schedulers which require two context switches in addition to running the actual scheduling algorithm.

4.5 Processor Interface

The processor interface provides the method of connecting a variable number of Microblaze processors to the two busses. It is split into two parts: Verilog

code that creates the actual connections and a C library that runs on each of the processors.

The C library provides function APIs such as Enqueue(), RaiseSemaphore(), and StartThread(). It also provides an interrupt controller that interfaces with the I/O hardware thread to pass interrupt sources to the proper SW thread's interrupt routine.

The communication between the C library and the hardware module is implemented using a single Microblaze Stream. Streams are built into the Microblaze processor and act very similarly to the hardware queues described above. There are two instructions in the Microblaze ISA, `put` and `get`, that each take two cycles for their data to be transfered into or out of the FPGA logic. When the streams are full or empty they will stall the processor if the corresponding `put` or `get` instruction is executed. It takes two `put`/`get` instructions to pass a message to or from the processor interface. Thus since the processor interface is designed to mask as much of the hardware overhead as possible it takes five cycles for the processor to complete any operation with any of the hardware primitives. Because of the way the message bus priority works, the worst latency possible with processor messages is $4 + n$ cycles where n is the number of processors attached to the system.

The hardware processor interface module has only one address on the main bus no matter how many processors there are. It internally queues and interleaves the processor operations, simulating any multiple requests to the same primitive from the processors. This was done to reduce the already large overhead of having the processor communicate with the hardware primitives.

The processor interface also manages the memory between the processors and the hardware threads. Each processor has its own copy of the memory and the hardware threads share another copy. A simple write-update coherency scheme is used simply because of the small size of the memories used in the project. If the memories were larger a more sophisticated coherency scheme could be used if needed with little adverse effect on the overall architecture.

Chapter 5

Compiler Architecture

Most of the design decisions in the runtime system were made in order to optimize and simplify our DSWP implementation and the Twill compiler pass. The Twill compiler itself is a multi-stage patchwork of other work along with custom compiler passes. This can be seen in Figure 5.1 where each block is a different tool used to transform the input C program into two programs, one in C and one in Verilog.

5.1 LLVM

The first step is the standard LLVM tool-flow [22]. LLVM 2.9 is used in order to have the LLVM IR directly compatible with the LegUp toolchain. LLVM's front-end is Clang which is responsible for generating LLVM IR from the input C code. Twill calls Clang with the "-O2", "-ffreestanding", and "-fno-builtin" flags. The freestanding and no-builtin flags ensure that the resulting code does not infer LLVM intrinsics such as memset, memcpy, variatic functions, return address manipulation, stack address manipulation, and other built-in LLVM features that are not explicitly present in the original C source. These features are all difficult or slow to implement in hardware and thus any optimizations Clang thinks it can

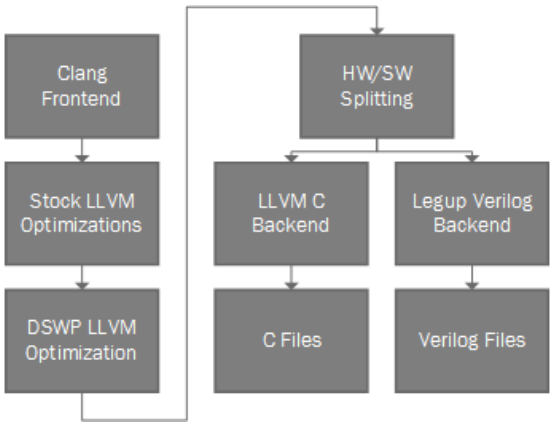


Figure 5.1: Twill Compiler Tool Flow

achieve by inferring these primitives will in general result in slowdowns in the resulting code.

After Clang is run and generates as output an LLVM IR assembly file, several builtin LLVM analysis and transform passes are run to further optimize the program and to massage it into an easier to work with form. In particular, Twill runs in order the LLVM passes "basicaa", "mem2reg", "mergereturn", "lowerswitch", "indvars", "inline", "always-inline", "simplifycfg", "gvn", "adce", and "loop-simplify" to prepare the LLVM IR for the DSWP transform passes.

5.2 DSWP

The DSWP algorithm conceptually pipelines loops by building a complete Program Dependence Graph (PDG) of the loop and then partitioning it into separate threads such that data is forwarded in only one direction between the threads. This technique was chosen as the main source of TLP parallelism because the original authors discovered that it became more efficient as the simplicity of the processing cores increased and because the required low-level and low synchronization-cost queues were relatively easy to create with control over the hardware.

We implemented the DSWP algorithm as three separate custom LLVM passes.

The first pass "fixes" globals by passing their addresses to all functions as parameters. Thus after this pass executes the only uses of globals in the entire program are guaranteed to be the very first instructions in the main function that essentially take the address of each global. This pass is done since the way LegUp synthesizes globals is to create new memory blocks on the FPGA containing the global data. These memories do not update between different hardware threads or between the hardware threads and the processor and thus any write to a global variable would cause incorrect behavior. Furthermore, this simplified the modified memory load/store operations in the LegUp source to always reference the unified address space rather than trying to figure out which memory space to address into. For programs with many globals, it may be more efficient to implement a cache-coherency scheme between the distributed memories but in practical applications it appears that the number globals actually used by a given function are quite limited and thus the overhead involved in passing globals by argument is relatively small.

After the custom global pass, the "deadargelim", "argpromotion", and "constprop" stock LLVM passes are run in order to clean up any dead code or global arguments being passed to a function that are unused. These passes also will identify any constant globals and replace their memory lookups with constant expressions.

The second pass builds the Program Dependence Graph (PDG) for each function. This pass relies on LLVM's builtin "basicaa" and "loops" analysis information. We found that LLVM's higher level dependence analysis framework handled call sites and arguments wrong inside of nested loops and thus re-implemented a similar framework from scratch. Once the PDG is built this pass assigns a weight to each instruction node in the PDG denoting how many estimated cycles each instruction is expected to take along with how much area the instruction is expected to take if transformed into hardware. For example, load and store instructions are both expected to take two cycles in software while store takes one cycle in hardware. Both load and store instructions take the minimum area possible for an instruction in hardware since they simply call out to the hard-

ware runtime system. Another example is the division instruction which takes 34 cycles to complete in software and only 13 in hardware. However, it is assigned a large penalty in hardware area since it requires a dedicated DSP block on the FPGA or an inordinate amount of LUT blocks.

Once the PDG is generated and properly annotated, the actual DSWP pass is run. The DSWP pass as input takes the number of initial partitions to build; it will output at least that many independent long running threads. The DSWP pass relies on the "postdomtree", "domtree", "domfrontier", "postdomfrontier", and "loops" stock LLVM analysis passes. The DSWP pass runs on the entire program rather than on each function.

For each function in the program, the DSWP pass runs a simple heuristic-based partitioning algorithm to divide the SCCs of the PDG between the partitions. The partitioning algorithm takes as input a targeted percentage of work to be assigned to each partition. The developer specifies an initial percentage of work to be done in the software domain as opposed to the hardware domain and this percentage is used to generate the initial targeted percentages for each partition. Furthermore, the partition percentages are adjusted accordingly as the DSWP algorithm is iterated upon as a result of the function call logic described below.

Each SCC is assigned two different estimations of its weight. The hardware weight consists of the sum of the estimated cycle·area products that would result by translating each instruction into hardware. The software weight consists of the estimated number of cycles required to execute the instruction on a Microblaze processor.

A sorted list of SCCs is maintained such that all SCCs that are valid to place onto the current partition according to the rules described in Section 3.1.1 are in the list. Essentially this means that all SCCs on the list do not have dependencies upon SCCs that have not been placed into a partition yet. Every time a new partition is started, the total hardware weight and total software weight of the SCCs currently on the sorted list are compared to determine whether this partition will be a software or hardware partition. Once this decision is made,

the list is resorted according to the appropriate weight and the smallest SCCs are added to the partition (and the list of available SCCs is updated) until the weight surpasses the targeted percentage for this SCC at which point a new partition is started and the process repeats.

Once the partitioning is complete, each partition is generated in a separate function named "⟨function_name⟩_dswp_⟨partition_number⟩". The functions are filled in with their appropriate instructions according to the SCCs assigned to that partition. Branch instructions are ignored during this step. In this step, dependencies between SCCs are also added in the form of pairs of calls to the special functions "Enqueue" and "Dequeue". The dequeues are inserted into the basic block where the dependent instruction would have been placed (this is described more in Section 5.2.1). The enqueues are not added during this step but are tracked in order to be added later once all partitions have been created.

Once all of a partition's basic blocks and instructions are filled in, care is taken to ensure that all basic blocks from the original function that contain a call site are included in the partition's function along with all control dependencies that call site may have outside of the partition. At this point the newly generated function contains all of the basic blocks and most of the final instructions minus branch instructions and call site instructions.

At this point, all of the branch instructions are added to each basic block and their branch targets are adjusted to the appropriate basic blocks in the new partition function. If the basic block the branch instruction would have targeted is not present in the new partition, the closest block that post-dominates the missing block is branched to instead. Any conditional branches that branch only to one block are replaced with unconditional branches.

After the branch instructions are inserted, all of the instructions' arguments are fixed to reference the new instructions in the partition function. Special care is taken with PHI nodes to ensure that there is an entry for each predecessor block and that each predecessor block is matched with the proper original predecessor block.

At this point it is possible for PHI instructions to rely on data from instructions not in this partition. In these cases a dequeue function call will precede the PHI node in the basic block which is illegal in LLVM IR definition. New basic blocks are inserted in these cases between the specified predecessor block and the block with the PHI node. The dequeues are then pushed into these new blocks so that the required ordering of the instructions is preserved while the PHI instructions remain as the first instructions in their basic block.

The above steps are repeated for each partition for the given function. Once all of the partition functions have been created, the enqueues function calls are placed into the partitions in the proper place. As part of this step, data conversion instructions are added to both the enqueue and the corresponding dequeue function calls in order to satisfy LLVM's type system.

All of the partition functions for each function in the source program are built using the above steps. Once all of the functions have been created, every single call site in the generated partition functions are visited and adjusted to call the proper partition's function. At this point an analysis of the dependencies between functions is used to determine whether it is possible to reuse queues. As queues are reused semaphore function calls are inserted where necessary to ensure that the assumptions about being able to reuse queues is correct.

The above entire process is repeated with different partitions assigned different target percentages at the partitioning step since the partitions can be swapped at function call boundaries (more information on how this happens in Section 5.2.1). In these cases when the partition is switched the percentages assigned to those functions will be switched and only the affected functions will be recomputed. This can cause multiple DSWP versions of a single original function if the original function is called in different locations in the original program. Currently this adjustment-and-recompute step is capped at happening a maximum of two times.

The output of the DSWP pass is an LLVM IR file that contains both the original functions and the new DSWP functions intermixed.

5.2.1 DSWP Differences

There are several algorithmic differences between our implementation and the implementation described in Ottoni et al.'s original paper [27] that we discuss below.

Function Calls

Probably the biggest difference is that our implementation of DSWP operates on the function level rather than on the loop-level. While pipelining code outside of loops is of questionable benefit, it allows us to implement a key extension to the original DSWP algorithm. The original algorithm treated function calls as a single large-latency instruction and thus would not pipeline any functions outside of the function containing the manually designated loop to pipeline. By extending the pipeline to the function level, our implementation treats function calls as zero latency instructions and then sets up a special dependence so that a sub-tree of threads will pipeline the called function. This sub-tree of threads will reuse the existing threads in the current pipelining when there is no recursion involved.

Therefore in our implementation, each function contains a "master" thread and zero or more "slave" threads. The thread that the call instruction is partitioned into becomes the master thread for the new function and is responsible for passing the arguments and receiving the returned result value. The other threads call the remaining slave versions of the function. All of the slave threads for that function do not accept any arguments for the function and instead will create standard `enqueue`/`dequeue` instruction pairs with the master thread only if the partitioner gives instructions to the slave thread requiring those arguments.

Thus when a function call is found, the pipeline is rebuilt for that function based off of the thread with the call instruction and then the old pipeline resumes once the function call has finished. This does create situations where data must flow against the direction of the original pipeline which puts the queue la-

tency on the critical path of the execution. It also potentially causes multiple versions of the same function to have to be translated into each hardware thread which increases the FPGA area required. To solve both problems, we move each function's master and slave threads into separate threads as long as the various call-sites to each function cannot execute at the same time. Within a single function, this is determined by a simple conservative heuristic which requires all call sites to have an unbroken chain of dependencies between them in order to be considered non-overlapping. Semaphores are used to ensure the function is indeed non-overlapping if the function has call-sites in multiple functions. In practice most of the time functions that do have overlapping calls tend to be simple functions that the partitioner will not partition anyways and thus the above two problems are avoided a majority of the time.

Furthermore, this method of resolving function calls potentially switches which partitions of a function are placed into software and hardware. Thus the function calls are resolved as the last step in the custom DSWP algorithm and the entire algorithm is iterated upon with different partitioning target percentage and roles for the partitions of the particular function that is called.

Conditional Control Dependencies

Since LLVM IR is in Single-Static-Assignment (SSA) form, some of the additional artificial conditional control dependencies introduced in the original paper are not implemented. The SSA form and its PHI nodes ensures that these scenarios cannot occur. However, there is an additional problem that LLVM's implementation of PHI nodes introduces. In LLVM, the PHI nodes may assign a constant based off of the control flow entering the block. An example of this problem is illustrated in Figure 5.2. The problem occurs when the partition that contains the PHI node does not have any instructions in one of the preceding basic blocks: BB2, BB3, or BB4. In this case according to [27] those basic blocks would not be present in the partition and thus the resulting threads will not be correct. Intuitively, the PHI node is control dependent on the branches in BB1

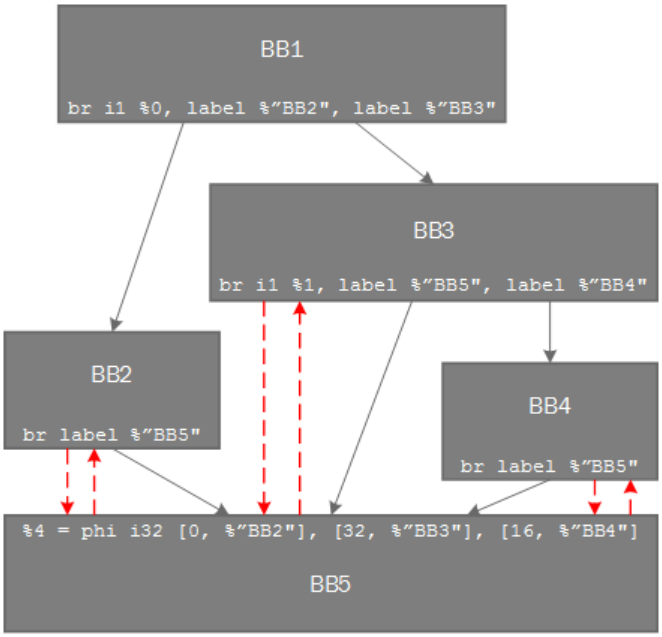
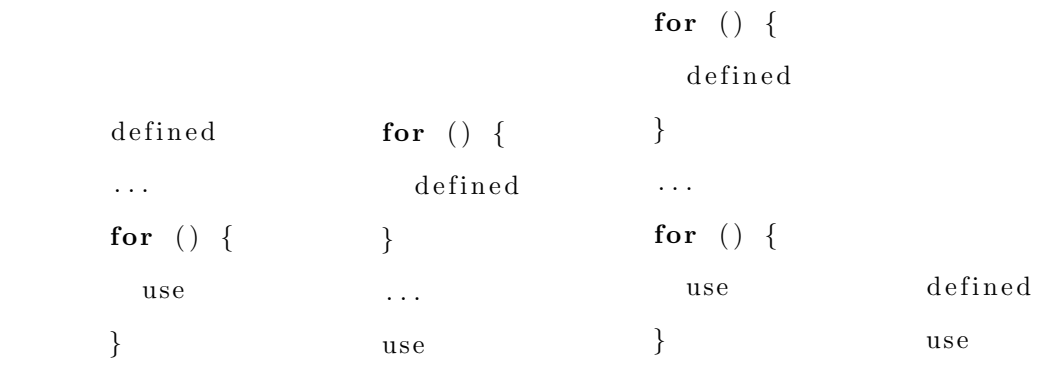


Figure 5.2: PHI Node Example Control Flow Graph: Gray edges represent the control flow while dotted red edges represent the fake dependencies

and BB3 but because of how LLVM handles PHI nodes it is not possible to forward the result of the branches using `enqueue`/`dequeue` instructions. Instead, we create a pair of fake dependencies between the PHI node and the branch instruction of every block that is associated with a constant. These dependency pairs can be seen in dotted red in Figure 5.2. This essentially forces the problematic branches and the PHI node to be on the same partition.

Loop Matching

Another difference in our implementation is how loops are handled. In the original implementation only one loop was handled in each program. Since functions can have an arbitrary number of loops arranged in an arbitrary fashion, care must be taken to ensure the `enqueue` and `dequeue` instructions are matched between loops properly. For each `enqueue`/`dequeue` pair we look at the loop structure and find the lowest loop in the original function that contains both the instruction whose result needs to be enqueued in the master thread (`defined`)



(a) use in a sub-loop of defined

(b) define in a sub-loop of use

(c) define and use in distinct loops

(d) define and use in same loop

Figure 5.3: Enqueue/Dequeue Loop Matching Cases

and the instruction that uses the defined instruction in the slave thread (`use`). At this point there are four cases shown in Figure 5.3. Figure 5.3 (d) shows the basic case where the loops are well matched. Trivially, the `enqueue` instruction is inserted directly after the `defined` instruction while the `dequeue` instruction is inserted directly before the `use` instruction.

For the case shown in Figure 5.3 (a), the `enqueue` instruction is inserted after the `defined` instruction while the `enqueue` instruction is inserted at the end of all of the `use` instruction's loop preheader blocks. Similarly, for the case in Figure 5.3 (b) the `dequeue` instruction is inserted directly before the `use` instruction while the `enqueue` instruction is inserted at the beginning of all of the `defined` instruction loop's exit blocks. In the case shown in Figure 5.3 (c) the `enqueue` instruction is inserted in all of the exit blocks while the `dequeue` instruction is inserted in all of the preheader blocks. Note that this will create asymmetric numbers of `enqueue`/`dequeue` instructions but will ensure that for any given control flow each loop iteration will have matching instruction numbers.

Furthermore, for every `enqueue`/`dequeue` pair a simple flow algorithm is run

on the lattice formed by the common dominator and post-dominator nodes to ensure that every `enqueue` is matched with a corresponding `dequeue`. The flow algorithm places dummy `enqueue` and `dequeue` instructions as required such that `enqueue` instructions are as close to the dominator node as possible while `dequeue` instructions are as close to the post-dominator node as possible.

Even after doing a flow adjustment this leads to some edge cases where naively doing the above will break the code. Whenever the preheader blocks have successors other than the loop header or when the exit blocks have predecessors other than blocks within the loop control flow is broken. In these cases, special basic blocks not present in the original function must be created between the block outside of the loop and the blocks inside the loop. The `dequeue`/`enqueue` instruction is then placed into this block and the branches are adjusted accordingly.

Another case where doing the above will break the code is if the `use` instruction is a PHI node and the `dequeue` instruction would be placed directly before the PHI node. In this case a new basic block not present in the original function is created on the control path between the basic block the PHI node is in and the basic block the `defined` instruction is in. The `dequeue` instruction is then placed in this basic block.

Homogeneous Threads

The final major difference between the original DSWP implementation and our modified implementation is that since the threads are not going to be run on homogeneous cores, the thread partitioner creates uneven partitions. It also ensures that all allocations and deallocations across all of the function calls are on a single special thread since a single thread must be in charge in order to keep the heap in sync.

5.3 HW/SW Splitting

After the DSWP transformation is finished, the generated threads must be split from the single LLVM IR file into HW and SW components. This stage generates a different set of stand-alone LLVM IR for each individual HW thread and SW thread based off of the results from the DSWP partitioner. Currently the special memory management thread is forced to be in software to take advantage of the standard C library's malloc/free although it would be straightforward to implement these two functions in hardware to allow hardware threads to manage the memory and to relax the requirement that all memory allocations must be on one thread. In practice, for media applications there are very few memory allocations inside the main computation loop which makes this limitation less problematic.

The only other special requirement for the split is that the master for the main function is always implemented in the software so the processor drives the entire program execution which is required for many SOC systems. After these two threads have been assigned, the larger partition sizes are prepared for the hardware translation while the smaller partitions are put onto any remaining processor cores. Only one thread for each processor is assigned unless the threads can be demonstrated not to overlap in execution time so that context switches are avoided.

Once the individual stand-alone LLVM IR files for each thread are generated, they are passed into the LLVM C backend for the software threads or into the LegUp Verilog backend for the hardware threads.

5.4 LegUp Modifications

We modified LegUp in several areas to interface with the Twill hardware runtime. First, the signals needed to interface with Twill's hardware runtime system were added to all generated LegUp Verilog modules. The output signals

for this interface are driven by a priority decoder and multiplexer combination that allows the signals to be sourced from whichever sub-module is currently active in the generated LegUp state machine.

All calls sites to the special functions of "Enqueue", "Dequeue", "Raise", and "Lower" are replaced with the equivalent Twill runtime hardware signaling. Furthermore, all load and store instructions are replaced with the appropriate signaling for interfacing with the Twill runtime hardware memory operations.

Several small modifications were made to how LegUp handles multiplies, division, memory blocks, and PLL blocks in order to use LegUp on Xilinx FPGAs rather than the originally supported Altera FPGAs. Thus, even though Twill has only been tested on Xilinx FPGAs the Twill tool-chain does support programming for Altera based FPGAs.

5.5 Final Steps

Once the output C program and Verilog modules are generated, they are imported into a Xilinx project that performs the final compilation and synthesis. The C program is compiled with Xilinx's version of GCC set to optimize for performance ("-O2"). The FPGA project is setup to optimize for speed but with no extra effort in the map and place & route algorithms. Finally, Xilinx's data2mem utility is used to generate a bitstream file to download to the FPGA once the elf and bit files are generated from their respective C and Verilog code.

Chapter 6

Results

All of the results presented were measured on a Xilinx XUPV5 board with a Virtex 5 FPGA. The runtime system has also been run on a Nexys 2 board with a Spartan 3E FPGA and a ZedBoard with a Zync-7000 SOC. All of the tests were run with only 8x32 sized queues and with one Microblaze processor. The Microblaze processor is configured to minimize its area according to the Xilinx tools to better simulate a constrained embedded system. All hardware modules including Microblaze are clocked at 100MHz. All HDL code for both LegUp and Twill was synthesized with the "optimize for performance" setting in the Xilinx ISE Project Navigator version 14.6.

The CHStone benchmarks from [15] were used to compare Twill to both the pure software solution and the pure hardware solution. These benchmarks are relatively parallelizable and also are fully supported by LegUp so a baseline could be established. Note that DFAdd, DFDiv, DFMul, and DFSine CHStone benchmarks all utilize 64-bit values and thus were not included since Twill currently does not support larger than 32-bit values.

Benchmark	# Queues	# Semaphores	#HWThreads
MIPS	12	0	1
ADPCM	328	0	5
AES	100	0	3
Blowfish	104	2	2
GSM	65	0	3
JPEG	576	3	6
MPEG-2	47	0	4
SHA	82	0	1

Table 6.1: DSWP Results

6.1 Twill DSWP Results

A summary of the number of hardware threads, queues and semaphores created can be found in Table 6.1. Across all of the benchmarks, the partitioner generated a workload split of about 75%-25% between the hardware threads and the software thread. The MIPS benchmark and SHA benchmarks both had all of their functions inlined and thus had no function calls to generate new threads. In contrast, the Blowfish benchmark had the largest number of functions that couldn't be extracted into their own thread due to the nature of its optimized call graph.

6.2 Area Analysis

The runtime system is quite small, using on average across all of the tests 2-4% of the FPGA. Each HWInterface module takes up 44 Look Up Tables (LUTs). An 8x32 queue uses 65 LUTs and one DSP block. Semaphores take up 70 LUTs with 100 primitives on the bus. The processor interface takes up 24 LUTs. The scheduler takes up 98 LUTs and two DSP blocks. Each of the two bus arbiters utilize 15 LUTs apiece.

Table 6.2 shows the total number of FPGA blocks used by Twill compared against the same benchmark purely translated by LegUp. The Twill HWThreads

Benchmark	LegUp	Twill HWThreads	Twill	Twill + Microblaze
MIPS	2101	1830	2318	3752
ADPCM	16893	7182	28682	30116
AES	16488	8302	15338	16772
Blowfish	5872	3293	10493	11927
GSM	7397	5888	11983	13417
JPEG	31084	18443	56101	57535
MPEG-2	16295	8116	13467	14901
SHA	12956	7856	13352	14768

Table 6.2: Number of LUTs used in FPGA logic for pure HW translation by LegUp and hybrid Twill implementation

column consists of only the number of LUTs that the LegUp translated HW threads take up. The Twill column includes the LUTs that the HW threads use along with the runtime system queues, semaphores, busses, and memory cache update system. Finally, the Twill + Microblaze column includes everything from the prior columns along with the LUTs used for the Microblaze soft processor. As can be seen the pure hardware size is always smaller than LegUp's translation mainly due to less functionality existing in the hardware. Adding in the overhead of the runtime system puts Twill's size on par with LegUp's results which is reasonable particularly if a hard processor is being used rather than a soft one. On average, we see a modest 1.73 times area decrease in the space required by the HW Threads and a slight increase of 1.35 area increase when including the Twill runtime system.

Aside from LUTs, LegUp makes use of BRAM memory blocks to pass arguments to functions and to handle arrays. Very few BRAM blocks are used in Twill's HW threads while most benchmarks used 10-15 BRAM blocks with the pure LegUp synthesis. Microblaze uses 16 BRAM blocks regardless of what code is running which provides 32kB of instruction and data memory for the Microblaze processor. In addition, with the way that Twill's memory management works almost all of the HW thread data is stored in the processor's data memory segment instead of creating new blocks. This gives all benchmarks comparable numbers of BRAM blocks between LegUp's pure HW translation and Twill's

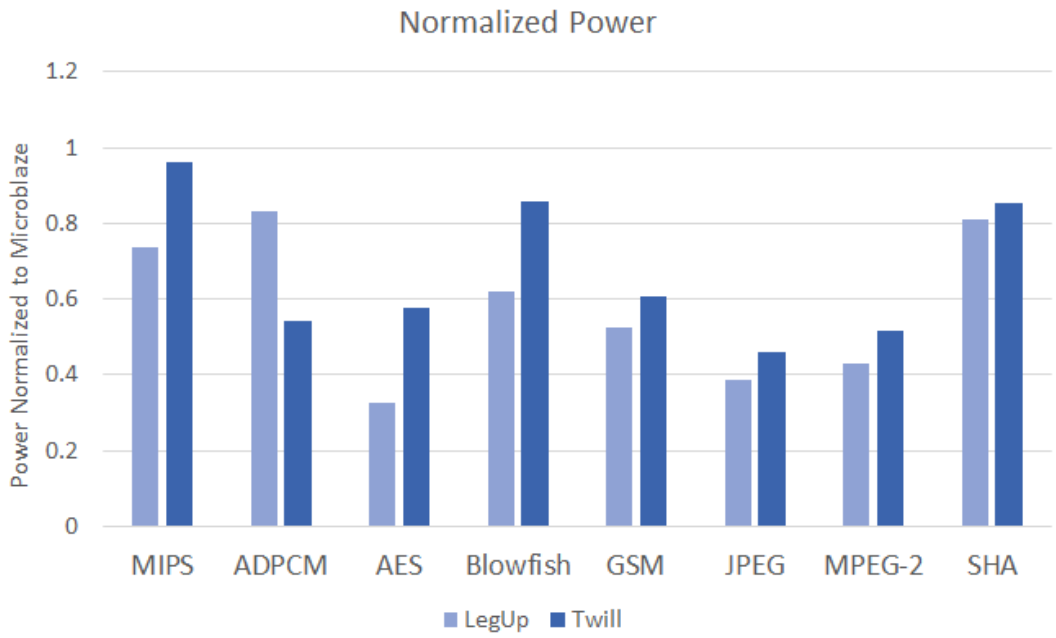


Figure 6.1: Power consumption normalized to the pure Microblaze SW implementation measured using Xilinx's power simulation tools

hybrid translation.

6.3 Power Analysis

Figure 6.1 shows the power characteristics obtained through Xilinx's power simulation tools. Twill is compared to LegUp's pure HW translation normalized to the pure software implementation running on Microblaze. As expected, the pure HW translation has the best power performance followed by Twill and then the pure Microblaze implementation. This is because Microblaze is really power inefficient compared to a direct hardware implementation. With a hard processor it could be expected that Twill's power consumption would be less than LegUp's since it has to synthesize less hardware. On examining why Microblaze is so inefficient it appears that the majority of the power consumption comes from the multiple Phase-Lock Loops (PLLs) used internally.

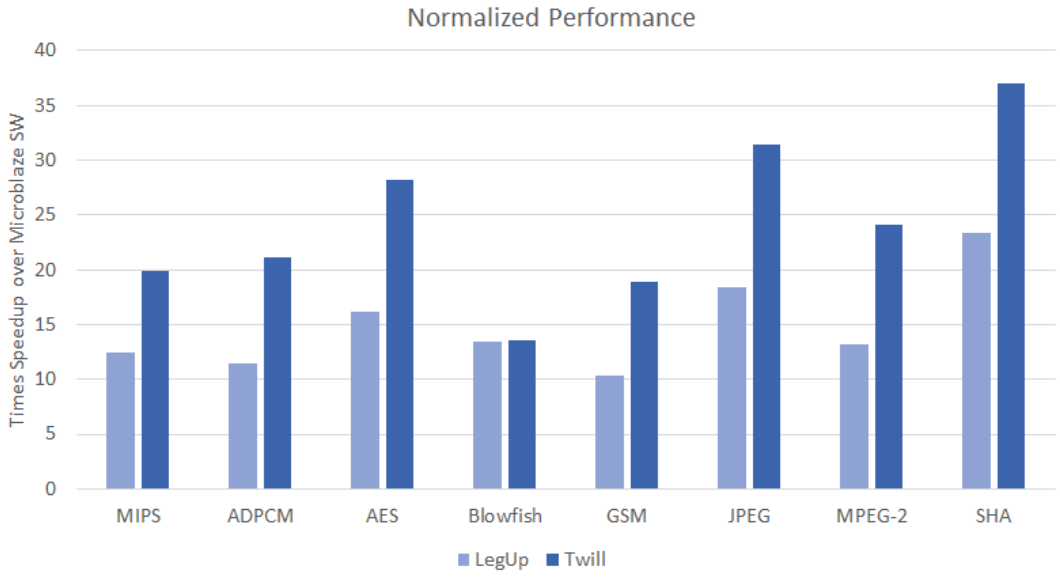


Figure 6.2: Performance speedups normalized to the pure SW implementation

6.4 Performance Analysis

Figure 6.2 shows the performance characteristics of Twill compared to LegUp's pure HW translation normalized against running the benchmark directly on the Microblaze processor. In general Twill outperforms the pure hardware implementation since it can take advantage of TLP as well as ILP. Twill on average achieves a 1.63 times speedup over the pure hardware implementation on these benchmarks which are designed to be easily translatable into pure hardware. Twill also vastly outperforms a pure SW implementation on the Microblaze processor as expected by on average 22.2 times. This speedup comes from multiple sources: arithmetic operations such as multiply and divide are much faster in hardware, LegUp will schedule as many instructions as possible at the same time to exploit ILP, and Twill will run instructions on the processor at the same time as LegUp is executing its state machine in order to exploit TLP.

Twill manages to only match the pure hardware speedup on the Blowfish benchmark. On closer inspection, it appears that Twill chose poor partitions for the hardware and software threads with each function call in the main loop

transferring the master control between the hardware and software. This causes the function argument data to be sent back and forth several times between the hardware and software threads before any computation on the data is performed. Similarly, the return value alternates back and forth before finally being used in the next iteration of the loop. We modified the heuristic specifically for this benchmark to prevent this behavior and found a 1.89 times speedup between the modified Twill implementation and the pure hardware implementation. This modified heuristic also decreased the number of queues from 92 to 34 which shows that our original heuristic for partitioning instructions into separate threads could use some improvement.

LegUp appears to do a poor job at synthesizing the ADPCM benchmark compared to the other benchmarks. This interpretation is consistent among the area, power, and performance results. Some of the constructs in this program appear to be quite difficult to synthesize which gives an advantage to Twill when it puts these parts on the processor. This is the only benchmark shown that utilizes division extensively which might be one of the contributing factors since LegUp was set up to use a simple serial divider for these tests.

6.5 Partitioning Heuristic Effects on Performance

We explored the effects of changing the targeted percentage of instructions to be placed into the partitions. Figures 6.3 and 6.4 show the changes in performance and queue count modifying where this split point lies. As can be seen most clearly in Figure 6.3 there is a negative correlation between the number of queues required and the performance of Twill for a given benchmark. Furthermore, it seems that the even splits between the HW/SW domains perform the worst. This is probably because when the first half of most computations are computed in SW and then the intermediate results are passed to the HW in order to finish the computation the communication costs skyrocket while the amount of TLP exploited remains about the same.

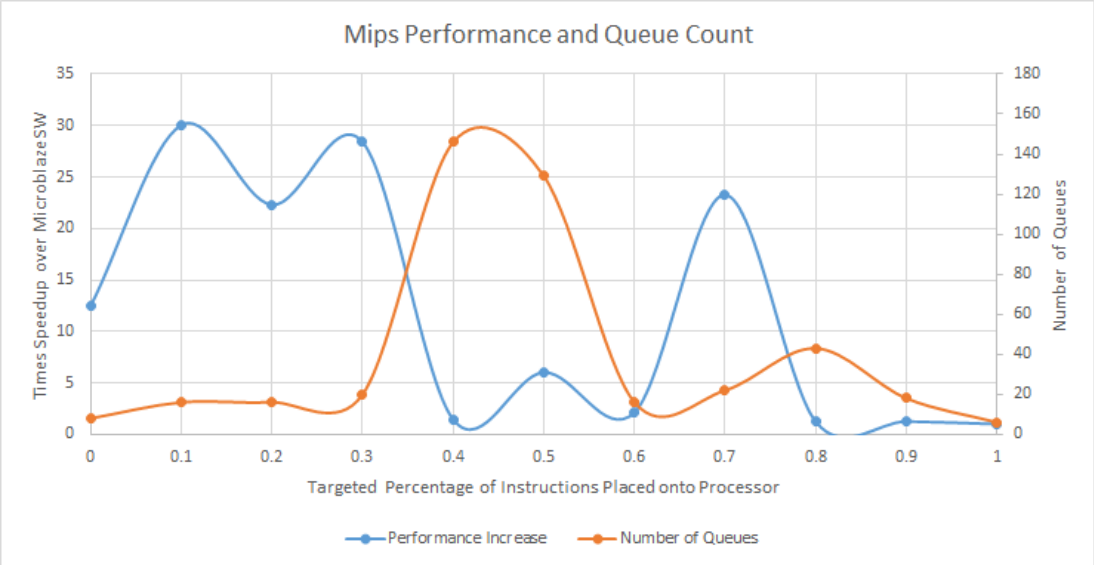


Figure 6.3: Mips benchmark performance with various targeted partition split points

Ottoni et al. found very similar results when they were experimenting with finding the optimal partitioning for a given loop. While they were very focused on balancing the work across threads in an optimal manner since they assumed homogeneous threads, they found that the greedy heuristic algorithm for partitioning is not particularly good at finding the optimal partition but often works "well enough". That seems to be the case with Twill as well. While perhaps a more complicated heuristic could be used to achieve better results, Twill's results show that its automatic thread extraction through partitioning can result in a significant performance increase without any programmer intervention.

6.6 Queue Size and Latency

One important result from the DSWP implementation described Ottoni et al. [7] is that the algorithm was very resilient to large queue latencies and short queue sizes regardless of the benchmark run. This was achieved by never having the pipeline "flushed" except for at the very end of program execution. Our implementation of DSWP potentially flushes the pipeline much more frequently

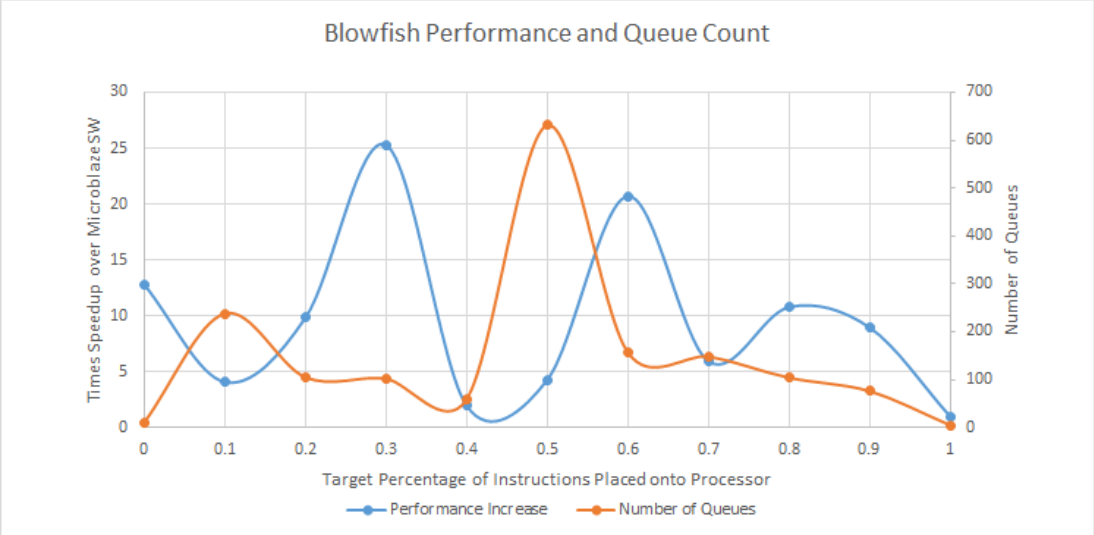


Figure 6.4: Blowfish benchmark performance with various targeted partition split points

on function boundaries and so a similar experiment was conducted to determine the resiliency of Twill to hardware queue latencies and sizes.

Figure 6.5 shows that while Twill's resiliency depends upon the application, overall Twill is still fairly resilient. Compared to Ottoni et al.'s original implementation of the DSWP algorithm, we have found a much bigger performance degradation as the queue latencies are increased. On average Ottoni et al. report a 10% slowdown with a queue latency of 100 while we found a 27% performance decrease on average with a queue latency of 128. As noted above, this is probably because of how Twill flushes the pipeline fairly frequently. In addition, the original paper only optimized a single long-running loop out of the entire program and thus any performance increase or slowdown effect will be magnified in our full program implementation. Thus we believe that our performance decrease is much closer to the original results than the data suggests.

Figure 6.6 shows similar results for the queue sizes. Note that for the JPEG benchmark the 32 queue size did not fit on the FPGA. Ottoni et al. found that they received a slowdown of 6% when reducing the queue length from 32 to 8. We found a comparable 9.7% slowdown when comparing our queue lengths of 32 and 8. As mentioned above, our slowdown/speedup results are probably exaggerated

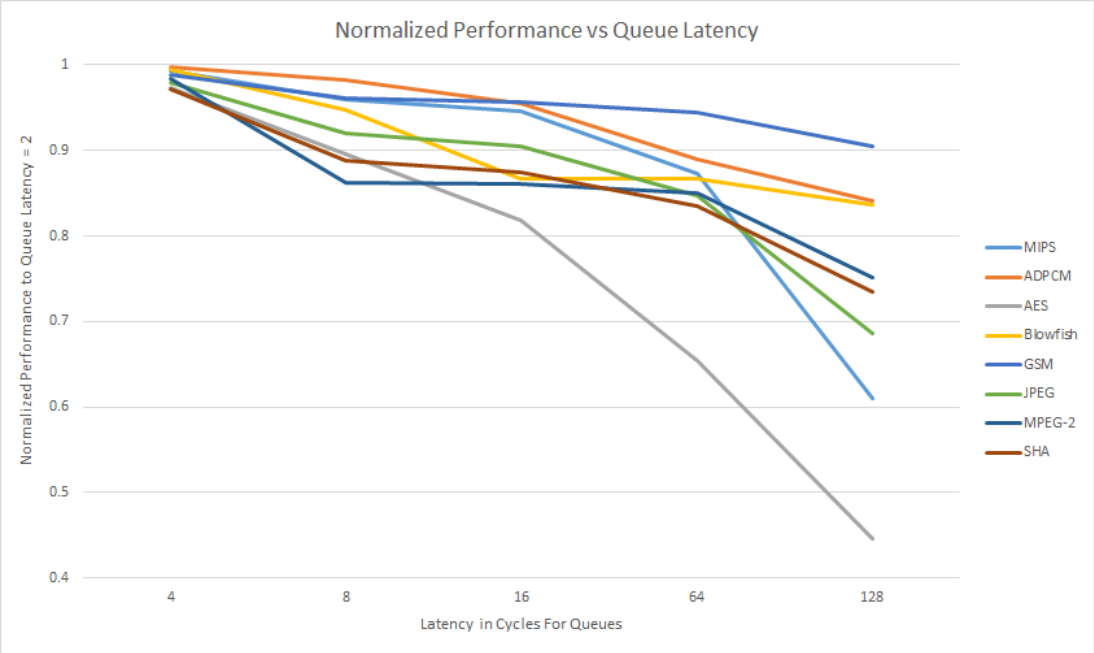


Figure 6.5: Twill performance speedups normalized to runtime with 2 cycle queue latency

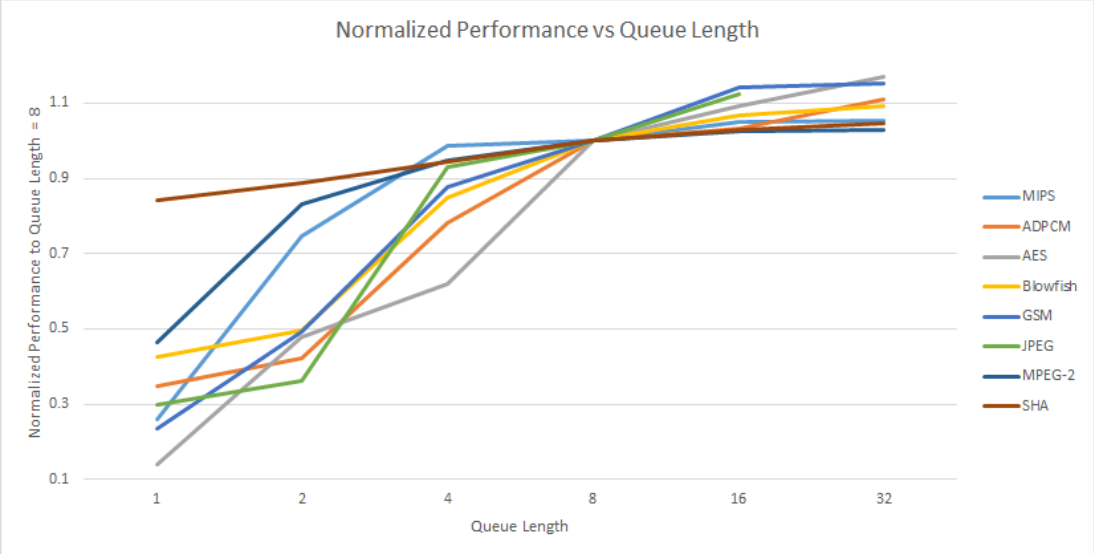


Figure 6.6: Twill performance speedups normalized to runtime with length 8 queues

compared to the original results; in addition, we used 32-bit queues while the original paper used 8-bit queues.

6.7 Results Overview

Overall, we found that Twill performed very favorably across several benchmarks compared to the LegUp pure HW implementation and the pure SW implementation. We found an average 1.63 times speedup over the LegUp implementation and an average 22.2 times speedup over the pure SW implementation. Furthermore, we found that the average FPGA area required for the Twill HW Threads decreased by 73% and that we only had an average FPGA area increase of 35% when factoring in the Twill runtime system overhead. Twill's partitioning heuristic is fairly inconsistent in finding the optimal partition and reinforces what Ottoni et al. found with their implementation of DSWP. Overall, Twill is resilient to changes in the queue latencies and sizes which provides some implementation flexibility when implementing queues and other runtime primitives.

Chapter 7

Conclusion

In this paper we presented a new hybrid SOC compiler and corresponding run-time system called Twill. Twill takes advantage of TLP and ILP in order to achieve an average performance speedup of 1.63 times over LegUp's pure hardware translation even while reducing the amount of area needed for the reconfigurable logic. Twill achieves this by utilizing a modified version of DSWP to extract long-running threads from the input C source and then distributing these threads across the hardware/software divide in a hybrid CPU-FPGA SOC.

7.0.1 Future Work

As mentioned in Chapter 3, Twill currently supports only a subset of the C language. Notably, recursion and function pointers are currently not supported. There is no conceptual reason preventing their implementation and we propose several methods to deal with them. Recursion is only a problem in hardware since there is no stack. The Twill DSWP implementation could be extended to support the concept of barriers. At each barrier point all threads would come to the same execution state such that all queues are empty. The recursive function calls represented by backedges in the call graph would then be protected by these barriers on either side with the master function call always being in software. In

this way, the recursive functions or chain of functions could be parallelized as normal and then only at the recursion point would the pipeline be flushed and restarted. This would be slower than the equivalent code written as a loop but should still give reasonable speedups over the pure hardware implementation.

A similar system could be used to handle function pointers as well. Everything up to the actual call instruction with the function pointer could be parallelized. Anytime a function pointer is assigned to a new function the code must be changed to assign the master DSWP function. The call could be protected with barriers with the software always having master control of the called function. Furthermore, the way Twill handles function calls would have to change slightly. Instead of having the calling function call all of the slave functions each master DSWP function would be responsible to start the slave functions. This would increase the overhead of function calls slightly but potentially could be limited with points-to analysis to only the functions that could be called through a function pointer.

Another shortcoming of Twill is that it does not support larger than 32 bit data values to be passed inside of queues. This means that 64 bit data types and structures that are bigger than 32 bits are not supported currently by Twill. This shortcoming is relatively easy to overcome; one option is to enqueue/dequeue two or more values at a time and rebuild the resulting data structure or to simply use multiple queues to pass the data.

Another aspect of Twill that can be improved is the partitioning heuristic. As mentioned in Section 6.4, the partitioning heuristic can have a huge impact on the final performance of the program. More research is needed into how different heuristics affect this performance and what the best heuristic is for various program types.

Finally, Vachharajani et al. [34] extended the DSWP algorithm to be speculative. This allowed them to greatly increase the speedup gained by the original algorithm with a little hardware support. Since Twill has a large control over the hardware through the reconfigurable logic, it seems relatively straightforward to extend Twill's DSWP algorithm to be speculative which should allow Twill to

extract even more long-running threads and increase the amount of TLP parallelization that it can utilize.

Bibliography

[1] G. Agha. ACTORS: A model of concurrent computation in distributed systems. *Artificial Intelligence*, 1986.

[2] J. Agron and D. L. Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *International Conference on Hardware Software Codesign*, pages 393–402, 2009.

[3] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. *IEEE Transactions on Software Engineering*, 28:847–862, 2002.

[4] E. Bergeron, L.-D. Perron, M. Feeley, and J.-P. David. Logarithmic-Time FPGA Bitstream Analysis: A Step Towards JIT Hardware Compilation. *ACM Transactions on Reconfigurable Technology and Systems*, 4:12–27, 2011.

[5] N. Bombieri, G. D. Guglielmo, L. D. Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli. HIFSuite: Tools for HDL code conversion and manipulation. In *IEEE International High-Level Design Validation and Test Workshop*, pages 40–41, 2010.

[6] S. Campanoni, T. Jones, G. Holloway, G.-Y. Wei, and D. Brooks. The helix project: Overview and directions. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 277–282, New York, NY, USA, 2012. ACM.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. D. Brown, and T. S. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Symposium on Field Programmable Gate Arrays*, pages 33–36, 2011.

[8] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. From Software to Accelerators with LegUp High-Level Synthesis. In *Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013.

[9] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *International Conference on Hardware Software Codesign*, pages 267–276, 2010.

[10] J. Curreri, S. Koehler, B. Holland, and A. George. Performance analysis with high-level languages for high-performance reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 23–30, April 2008.

[11] C. Galuzzi and K. Bertels. The Instruction-Set Extension Problem: A Survey. *ACM Transactions on Reconfigurable Technology and Systems*, 4:18–28, 2011.

[12] L. Gantel, A. Khiar, B. Miramond, M. E. A. Benkhelifa, L. Kessal, F. Lemonnier, and J. Le Rhun. Enhancing reconfigurable platforms programmability for synchronous data-flow applications. *ACM Trans. Reconfigurable Technol. Syst.*, 5(3):14:1–14:16, Oct. 2012.

[13] M. Gokhale and J. M. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *Field-Programmable Custom Computing Machines*, pages 126–135, 1998.

[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design*, pages 461–466, 2003.

[15] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *IEEE International Symposium on Circuits and Systems*, pages 1192–1195, 2008.

[16] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Field-Programmable Custom Computing Machines*, pages 12–21, 1997.

[17] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *European Conference on Object-Oriented Programming*, pages 76–103, 2008.

[18] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. *Sigplan Notices*, 39:59–70, 2004.

[19] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian, and H. Saito. On the exploitation of loop-level parallelism in embedded applications. *ACM Transactions in Embedded Computing Systems*, 8:1–34, 2009.

[20] D. Koch, C. Beckhoff, and J. Torrison. Fine-grained partial runtime reconfiguration on virtex-5 fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 69–72, May 2010.

[21] S. Koehler, G. Stitt, and A. D. George. Platform-aware bottleneck detection for reconfigurable computing applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(3):30, 2011.

[22] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[23] E. Lubbers and M. Platzner. ReconOS: An RTOS supporting Hard and Software Threads. In *Field-Programmable Logic and Applications*, pages 441–446, 2007.

[24] P. m. Athanas and H. f. Silverman. Processor Reconfiguration Through Instruction Set Metamorphosis: Compiler and Architecture. *IEEE Computer*, 1993.

[25] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation. *ACM Transactions on Reconfigurable Technology and Systems*, pages 1–38, 2012.

[26] J. T. Oplinger, D. L. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, 1999.

[27] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *International Symposium on Microarchitecture*, pages 105–118, 2005.

[28] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems*, pages 1–24, 2011.

[29] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews. Hthreads: A Computational Model for Reconfigurable Devices. In *Field-Programmable Logic and Applications*, pages 1–4, 2006.

[30] M. Purnaprajna, M. Porrmann, U. Rueckert, M. Hussmann, M. Thies, and U. Kastens. Runtime reconfiguration of multiprocessors based on compile-time analysis. *ACM Trans. Reconfigurable Technol. Syst.*, 3(3):17:1–17:25, Sept. 2010.

[31] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. J. Eggers. CHiMPS: a high-level compilation flow for hybrid CPU-FPGA architectures. In *Symposium on Field Programmable Gate Arrays*, 2008.

[32] C. G. Quiones, C. Madriles, F. J. Snchez, P. Marcuello, A. Gonzlez, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading

based on pre-computation slices. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, 2005.

[33] W. Thies, V. Chandrasekhar, and S. P. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *International Symposium on Microarchitecture*, pages 356–369, 2007.

[34] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative Decoupled Software Pipelining. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, 2007.

[35] N. Vassiliadis, G. Theodoridis, and S. Nikolaidis. An automated development framework for a RISC processor with reconfigurable instruction set extensions. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2006.

[36] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines*, pages 127–134, 2010.

[37] R. D. Wittig. OneChip: An FPGA Processor With Reconfigurable Logic. In *Field-Programmable Custom Computing Machines*, 1995.

[38] I. Y Explorations. Ansi-c to rtl automatically. Accessed: 2014-2-20.

[39] X. yu Pan, Y. Zhao, Z. Chen, X. Wang, Y. Wei, and Y. Du. A Thread Partitioning Method for Speculative Multithreading. In *Scalable Computing and Communications*, pages 285–290, 2009.