From Software Threads to Parallel Hardware with LegUp High-Level Synthesis

by

Jongsok Choi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Department of Electrical and Computer Engineering
University of Toronto

Abstract

From Software Threads to Parallel Hardware with

LegUp High-Level Synthesis

Jongsok Choi

Doctor of Philosophy

Graduate Department of Department of Electrical and Computer Engineering

University of Toronto

2016

High-level synthesis (HLS) can *automatically* synthesize software to hardware. With the design specification in software, HLS can reduce the lengthy design cycles of hardware, and make the performance and energy-efficiency benefits of hardware accessible to those without hardware skills.

Since the introduction of the first C-based HLS tools more than a decade ago [49], however, the adaption of the technology has been slow by both software and hardware engineers. We attribute this to two key factors: 1) For hardware engineers, there is still a gap between HLS-generated hardware and human-designed hardware, partly due to the inability of HLS tools to fully exploit hardware parallelism, and 2) for software engineers, HLS remains to be a difficult endeavour, as many parts of the design, such as system integration, largely remain a manual process.

This dissertation provides an HLS framework, *LegUp*, which seeks to address both issues. LegUp can compile an entire software program to hardware to produce a *hardware-only* system, or it can also automatically partition the program to generate a *processor-accelerator hybrid* system, wherein the compute-intensive program segments are accelerated by hardware, with the remaining segments executed in software on a processor. In both cases, a *complete* system is generated, including necessary memories and interconnect. To allow one to easily exploit hardware parallelism, we provide HLS support for synthesizing parallel software to parallel hardware. In particular, we support automatically compiling a multi-threaded program with Pthreads and OpenMP to parallel hardware accelerators that operate concurrently within a hardware-only or a processor-accelerator hybrid system. In the context of parallel hardware, we investigate architectural and memory optimizations that help to improve circuit performance and area, and discuss a method of using the *producer-consumer* pattern in software to infer a *streaming* circuit in hardware. With these techniques, we show that LegUp can produce high-performance hardware that can be competitive to circuits that are generated by commercial HLS tools, and demonstrate that LegUp-generated circuits can also *outperform* software executing on `x86` processors.

Acknowledgements

I would like to start by expressing my sincere gratitude to my two incredible advisors, Jason Anderson and Stephen Brown. Jason Anderson has provided invaluable guidance throughout my Ph.D studies, and has helped me to vastly improve my research abilities and communication skills. His devotion to the LegUp project has inspired many students working on the project, including myself. I admire your enthusiasm and dedication to your students, and I thank you for your mentorship on research, as well as in life in general. My co-advisor, Stephen Brown, gave me the opportunity to work on this exciting project with an amazing team, and I appreciate your high-level vision which led to inception of this project, and thank you for allowing me to flexibly choose my research paths. I would also like to thank my committee members, Vaughn Betz and Paul Chow, and my external examiner, Deming Chen, for providing valuable feedback on this work, and Joyce Poon, for chairing my Ph.D defense.

I would like to thank all of the graduate students who I have had the privilege to work with on the LegUp project: Bain Syrowik, Blair Fort, Nazanin Calagar, Marcel Gort, Mark Aldham, Joy (Yu Ting) Chen, and Julie Hsiao. In particular, I want to thank Andrew Canis, who I have worked with since my master's degree and since the very beginning of the project, and have spent numerous nights coding and debugging with. It is amazing to see what LegUp has become over the years. I also want to thank Lanny (Ruo Long) Lian, who has help me on countless number of different topics, and has provided the extra thrust that we needed to take the project further. Thanks to all the friends that I have made in graduate school: Braiden Broussau, Xander (Alexander) Chin, Safeen Huda, Jin Hee Kim, Daniel Di Matteo, Alex Rodionov, Vincent Mirian, Charles Lo, Davor Capalija, Jasmina Vasiljevic, and Henry Wong, who have all made graduate school an enjoyable place.

I am grateful for the generous scholarships that I was fortunate enough to receive, including the Natural Sciences and Engineering Research Council of Canada Alexander Graham Bell Canada Graduate Scholarship, the Ontario Graduate Scholarship, the Right Track CAD Graduate Scholarship, the Walter C. Sumner Memorial Fellowship, the Bell Graduate Scholarship, and the Rogers Scholarship, all of which made my long Ph.D years financially viable.

I am thankful to all my friends outside of school, Dave, Dylan, Kevin, Won, Chulmin, and Seyeon, who made the life of an old student still fun and enjoyable. I am grateful to my parents, who made all of this possible by moving to Canada half a lifetime ago, while leaving their careers, friends, and families back home. Thank you for always encouraging me to be my best and to pursue my goals.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The past four decades have seen tremendous growth in the computing industry. In 1971, Intel released its first ever microprocessor, the 4004 [127] – a 4-bit processor built for a calculator, with 2,300 transistors fabricated on a 10 μm (10,000 nm) process. It was capable of adding two 8-bit numbers in 850 μs, or performing $\sim$1,200 8-bit additions per second. Forty-five years later, one of the latest GPUs from NVidia, the Tesla P100 GPU built for Deep Learning, has 15.3 billion transistors built on 16 nm FinFET technology, and delivers more than 10 TFLOPS (10^{13}) of 32-bit single-precision floating-point compute power [83]. This incredible advancement in technology has been driven by the Moore's Law, an observation which stated that the number of transistors on a chip will double approximately every two years [54]. Moore's Law has become a self-fulfilling prophecy, with the industry making the technology advancements necessary to fulfill the law. With each new process node (shrink in transistor feature sizes), clock frequency increased by $\sim$50%, and transistor density increased by 100% [62]. Improvements in manufacturing technology also permitted increasing die sizes without increasing cost. However, as transistors became smaller, leakage current increased, and with more transistors per unit area, power density increased exponentially, creating the *power wall*. This has meant that processor clock frequencies, and thereby single-thread performance, has remained nearly stagnant in the past decade, as shown in Figure 1.1. A solution to this has been to increase the number of processor cores, rather than increasing the speed of a single core. The parallelism permitted improved *overall* performance, without skyrocketing power budgets. With multiple cores, parts of the chip can also be turned off to save power and prevent overheating, a phenomenon known as *dark silicon* [26]. Thus, in the past decade, as shown in

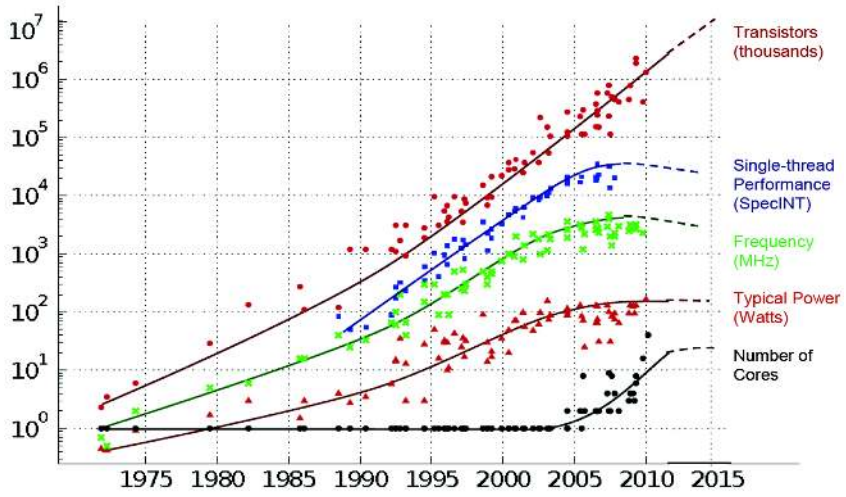


Figure 1.1: Processor trends over the last four decades [53].

the figure, the industry trend has been to increase the number of cores to improve performance. Modern state-of-the-art consumer CPUs, such as those from the Intel Xeon E7 family [128], have well over 10 cores.

In addition to multi-core CPUs, the use of specialized hardware, which is dedicated to a particular computational task, has also been on the rise. Many of these accelerators offer high performance through massively parallel architectures. For example, the Intel Xeon Phi, which is used in supercomputers, has more than 60 cores [126]. The Nvidia K80 GPU has almost 5,000 CUDA cores [73]. All of these CPU/GPU cores are generally programmed through parallel programming methodologies in software, such as Pthreads [7], OpenMP [85], OpenCL [111], and CUDA [133], where the programmer explicitly specifies the parallelism in the code. In addition to these parallel but flexible architectures, there are also application-specific accelerators, which are *dedicated* for performing a specific task, such as encryption [50], video processing [52], and deep learning [13]. This dissertation presents new methodologies to ease the design of such accelerators on a *reconfigurable* platform called a field-programmable gate array (FPGA).

FPGAs are hardware chips that can be *instantly* programmed to function as *any* digital circuit. This allows one to accelerate an application in *hardware*, which can provide orders of magnitude improvement in performance and power efficiency compared to software running on a processor [21, 45, 154]. Over the past decade, FPGAs also have continued to grow in size and complexity, allowing complex systems to be implemented on a single FPGA. State-of-the-art FPGAs have high computational capacity with abundant logic blocks, which can be programmed to function as needed, as well as dedicated hard blocks, such as memories and DSPs. For example, Xilinx announced the Virtex UltraScale XCVU440, a 20 nm

device with 4.4 million logic cells [142], containing more than 20 billion transistors, making it one of the worlds densest ICs (integrated circuits). These copious amounts of available logic cells can be tailored to *create* a custom multi-core hardware system.

Traditionally, one of the biggest barriers to using an FPGA has been its design entry, which required the use of hardware description languages (HDLs), such as VHDL or Verilog. Compared to software, HDLs are extremely complex, difficult to use, and time consuming to debug. The difficulty continues to rise, with the ever-increasing complexity of circuits implemented on FPGAs. Due to these barriers, software engineers, without hardware knowledge, simply could not use FPGAs. Despite the performance and power advantages that FPGAs offer, hardware skills are relatively rare compared to software skills (there are 10× more software engineers than hardware engineers in the United States [70]), thus limiting the broad uptake of FPGAs as a *mainstream* computing platform.

In the last two years, however, there have been important new developments in the FPGA industry. In 2014, Microsoft announced that they accelerated *Bing Search* by 2× with FPGAs, and with this success, they have started to use FPGAs in their data centres [63]. In 2015, Intel acquired Altera, one of the two largest FPGA vendors in the world, for $16.7 billion [124]. With the acquisition, Intel projected that FPGAs will be in 30% of all data centre servers by year 2020. The significance of these is that FPGAs, whose previous applications were mostly limited to niche markets in networking or telecommunications, will increasingly be used for general-purpose computing, which represents a much larger market. With millions of servers in data centres [51], and constantly changing applications, it is inconceivable to think that the hardware for these FPGAs will be designed manually, due to the lack of hardware expertise and lengthy/complex hardware design cycles.

High-level synthesis (HLS) is an up-and-coming design methodology for FPGAs. HLS raises the design abstraction of hardware to software, allowing a user to automatically generate a circuit description from a high-level software specification. The advantage of HLS is that a circuit designer can work more productively at a higher level of abstraction, reducing time-to-market compared to manual hardware design. With the input specification in software, HLS ultimately aims to bring the performance and power advantages of FPGA hardware to those with only software skills. With these advantages, and the associated promise of increased chip revenue, both Altera and Xilinx, the two largest FPGA vendors, have invested heavily in HLS, with each offering a compiler that can produce high-quality circuits. A number of academic groups have also built their own HLS tools, including [10, 23, 61, 57]. With the recent advances in both industry and academia, HLS compilers have improved significantly, such that in specific cases, they can automatically generate circuits with comparable, or even better performance and area to manually designed hardware [120], and companies have also started to use HLS to design

chips in production [36].

Despite the advantages of HLS, there is still a gap between HLS-generated hardware and human-designed hardware. This is partly due to the inability of HLS to fully exploit the parallelism available in the FPGA fabric. Hardware parallelism with HLS is typically achieved in the two following ways: 1) Instruction-level parallelism, where multiple basic operations, such as a multiplication and an addition are performed in the *same* clock cycle, and 2) pipelining, where a single piece of hardware can *overlap* multiple iterations of computations to execute them at the same time, much like a pipelined processor. We consider both such methods as *fine-grained* parallelism, where a set of instructions, or loops in a function, are executed in parallel to improve performance. Coarse-grained parallelism, such as those at the *thread*-level, are not addressed with these techniques. Thread-level parallelism is increasingly common in the software domain, as it is essential to take advantage of modern multi-core CPUs/GPUs. Thus we believe a *key* missing feature for HLS is a methodology for automated synthesis of a *multi-core* hardware system, which takes advantage of the millions of available logic cells to maximize performance, where the specification to create the multi-core hardware is as *easy* as using a multi-core CPU. So far, creating a multi-core hardware system in HLS typically requires one to first generate the core with HLS, then either stitch the cores together manually using HDL, or use an FPGA-vendor-provided system integration tool like Qsys [103]. To create a complete system, it also frequently involves bringing up off-chip interfaces such as DDR3 memory. This can be cumbersome for a hardware engineer, but may not even be possible for a software engineer. Some HLS tools have begun to support the use of vendor-specific pragmas to replicate hardware modules [144]. However, vendor-specific pragmas are unintuitive, not cross-platform, and most importantly, are not standard software. They are used to create hardware behaviour that is *different* from software behaviour (software executes sequentially whereas hardware executes in parallel), which can introduce hardware bugs that are *invisible* from software.

My Ph.D. research addresses this challenge by presenting a methodology where one can use well-known standard techniques in *multi-threading* to specify parallelism in software, where the multiple software threads are *automatically* compiled to concurrent accelerators in hardware. This brings *parallel software* to *parallel hardware*. The complete system, including on-chip/off-chip memories and interconnect, is also automatically generated, making the process as streamlined as using a multi-core processor. We believe this is an important step towards improving the usability of HLS, to truly allow software engineers to create high-performance multi-core hardware systems, using standard parallel programming methodologies that they are already likely familiar with. We implement the work in this research within the LegUp High-level Synthesis Framework from the University of Toronto [10].

1.2 Thesis Contributions

The objective of my Ph.D. research is to answer the following questions:

- How can we use software methodologies in HLS to exploit the spatial parallelism available on FPGA hardware?
- How can we create efficient concurrently operating hardware in HLS using these software methodologies?

To answer these questions, we make several contributions, as outlined below:

Chapter 3 describes our HLS support for synthesizing parallel software threads (specified using the Pthreads or OpenMP standards) to parallel-operating hardware modules, in a process-accelerator hybrid system. Each software thread is synthesized into a concurrently operating hardware accelerator, with the remaining program segments executed on a *soft* MIPS processor. Such a hybrid approach is attractive as accelerators deliver the compute power of hardware, while the MIPS processor offers the flexibility of software. The generation of the complete System-on-Chip (SoC), including the processor, accelerators, on-chip caches, as well as off-chip memory, is entirely automatic. Parallel software often requires synchronization across threads, hence we also provide HLS support for two key synchronization constructs: `mutexes` and `barriers`. This has been published in the 2013 IEEE International Conference on Field-Programmable Technology (FPT) [14].

Chapter 4 extends the work in Chapter 3 by adding support for the *hard* ARM processor (dual-core Cortex-A9) on the Altera Arria V SoC FPGA [107]. In recent years, FPGA vendors have introduced SoC chips with a *hard* ARM processor integrated on the same die as the FPGA [106, 140]. The ARM processor, typically running at speeds between 800MHz and 1.5GHz, runs much faster than the FPGA fabric, and can execute software programs significantly faster than what was previously possible with *soft* processors, such as the MicroBlaze [135] or the NIOS II [105]. In this chapter, we describe the creation of an ARM hybrid system on the SoC FPGA, where Pthread functions are accelerated to hardware, and the remaining software segments are executed on the ARM processor. The ARM processor can run with an OS, or in bare metal (no OS). We also provide direct memory access (DMA) support for memory transfers between hardware accelerators and off-chip DDR3 memory, which significantly improves memory bandwidth and performance. We show that our ARM hybrid systems can show significant performance and energy benefits compared to software executing on the *soft* MIPS, the *hard* ARM, and two different `x86` processors, as well as the MIPS hybrid systems from Chapter 3. This work is to be submitted to the IEEE Transactions on Very Large Scale Integration Systems (TVLSI).

Chapter 5 outlines a different HLS flow, where parallel software threads can be compiled to concurrently operating hardware modules without the need for a processor to be present in the system. We note that for some applications, it can be beneficial to compile the entire program to hardware, instead of using the processor-accelerator hybrid system. ARM SoC FPGAs are still relatively nascent, with only a handful of SoC FPGAs on the market, and soft processors can add significant area/power overheads. We describe the *hardware-only* flow of LegUp, where the entire multi-threaded software program is compiled to hardware, with parallel-threaded modules executing concurrently within a larger hardware system. We show that this methodology can produce parallel circuits that bring significant benefits in speed, power, and area-delay product, when compared to sequential hardware. Along with the work in Chapter 4, this work is to be submitted to the IEEE Transactions on Very Large Scale Integration Systems (TVLSI).

Chapter 6 investigates two crucial aspects that affect circuit performance and area of parallel hardware: *circuit topology* and *memory architecture*. HLS automatically compiles a software program to a hardware circuit, generally comprised of multiple hardware modules. The hardware modules can be connected within the overall circuit in various ways, defining different circuit topologies. We investigate two different circuit topologies, 1) the *nested* topology, and 2) the *flat* topology. In the nested topology, hardware modules are created in a hierarchical manner: modules are instantiated *within* the modules that use them. Conversely, the flat topology instantiates all hardware modules at the *same* level of hierarchy. We also explore methods to reduce memory contention among parallel hardware units by investigating three different memory architectures which use: 1) a *global* memory controller, 2) *local* memories, and 3) *shared-local* memories. Local and shared-local memories are dedicated RAM blocks for a single or a set of hardware modules. Lastly, we also consider *memory replication* to localize memories to each hardware module that uses them to reduce memory contention, as well as converting small memories to registers to reduce memory usage. This was has been published in the 2015 IEEE International Conference on Field-Programmable Technology (FPT) [15].

Chapter 7 details our work on providing Pthreads support to *infer* streaming hardware in HLS. Streaming hardware, which allows multiple data elements to be processed at the same time in a pipelined fashion, permits high hardware throughput. However, most HLS tools require that streaming functionality be specified using vendor-specific pragmas. This results in non-standard software that generates hardware behaviour that is different from software. To address this, we propose using a well-known software technique to infer streaming parallel hardware in HLS. Specifically, we use the producer-consumer pattern, commonly used in multi-threaded programming, to infer the generation of hardware that can exploit both *pipeline* and *spatial* parallelism on FPGAs. Our proposed methodology allows one to cre-

ate a design in software, using only standard software methodologies, that can not only synthesize to streaming hardware, but also model the generated hardware more accurately than existing solutions from other state-of-the-art C-based HLS tools. This work has been published in the 2016 IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) [16].

1.3 Thesis Organization

The Ph.D thesis is organized as follows: Chapter 2 gives a survey of currently available HLS tools, discusses a number of different parallel programming languages that are supported in the HLS tools, and examines a few case studies comparing HLS-generated hardware to human-designed hardware. It also describes the LegUp High-level Synthesis Framework wherein the thesis research is implemented. The research contributions are presented in Chapters 3, 4, 5, 6, and 7. Chapter 8 provides conclusions and gives suggestions for future work.

Chapter 2

Background

This chapter provides an overview of a number of currently available HLS tools, discusses some of the different parallel programming languages which are supported in those HLS tools, and examines a few case studies that have compared HLS-generated hardware to human-designed hardware. Lastly, it gives a high-level overview of LegUp HLS, describes some of its most important features, and highlights a few studies comparing the quality of LegUp-generated hardware to those of other HLS tools.

2.1 An Overview of Active High-Level Synthesis Tools

Table 2.1 lists a number of academic and commercial HLS tools that are currently available, along with their input/output languages, released years, and target domains. For some HLS tools, their output languages are not clearly specified publicly, hence for those tool, we have listed their outputs as RTL (register-transfer level).

A++ [94] is Altera's newest HLS tool, and is generally intended for use by hardware engineers to create IP cores from C/C++ code. The generated core can then be manually integrated into a larger system by the user. On the contrary, Altera's OpenCL SDK [100], which is targeted for software engineers, generates a complete system, with C/C++ host code executing on a processor (ARM or `x86`), and the HLS-generated hardware from OpenCL executing on an FPGA. It is typically used for massively parallel applications, where multiple threads execute on deeply pipelined hardware. The PCIe link that is used by an `x86` processor to communicate with an FPGA, as well as off-chip memory interfaces used by FPGA hardware to access memory, are all set up automatically by the tool.

Xilinx also offers a number of HLS tools, which target different types of users and application domains. Xilinx acquired AutoESL in 2011 [119], for their HLS tool, AutoPilot [30], which was re-branded as

Table 2.1: An overview of high-level synthesis tools [56].

Name	Developed By	Type	Input Language	Output Language	Year	Target Domain
A++	Altera	Commercial	C/C++	RTL	2015	Generic
Bambu	Politecnico di Milano	Academic	C	Verilog	2012	Generic
Bluespec Compiler	BlueSpec Inc.	Commercial	Bluespec System Verilog	Verilog	2007	Generic
Catapult HLS	Mentor Graphics	Commercial	C/C++/SystemC	Verilog/VHDL/SystemC	2004	Generic
CoDeveloper	Impulse Accelerated	Commercial	Impulse C	Verilog/VHDL	2003	Streaming
CyberWorkBench	NEC	Commercial	C/SystemC	Verilog/VHDL	2011	Generic
DK Design Suite	Mentor Graphics	Commercial	Handel-C	Verilog/VHDL	2009	Embedded
DWARV	TU Delft	Academic	C subset	VHDL	2012	Generic
eXCite	Y Explorations	Commercial	C	Verilog/VHDL	2001	Generic
FCUDA	UIUC	Academic	CUDA	C	2009	Massively parallel
GAUT	University of Bretagne	Academic	C/C++	VHDL	2010	Digital signal processing
LegUp	University of Toronto	Academic	C	Verilog	2011	Generic
MaxCompiler	Maxeler	Commercial	MaxJ	RTL	2010	Dataflow
SDAccel	Xilinx	Commercial	C/C++/OpenCL	RTL	2015	Massively parallel
SDSoC	Xilinx	Commercial	C/C++	Verilog/VHDL	2015	Embedded
SDK for OpenCL	Altera	Commercial	OpenCL	Verilog	2013	Massively parallel
Stratus	Cadence	Commercial	C/C++/SystemC	RTL	2015	Generic
Synphony C	Synopsys	Commercial	C/C++	Verilog/VHDL/SystemC	2010	Generic
Vivado HLS	Xilinx	Commercial	C/C++/SystemC	Verilog/VHDL	2013	Generic

Vivado HLS after the acquisition. Vivado HLS is one of the most widely used commercial HLS tools for FPGAs, and is typically used by hardware engineers to generate IP cores from software. More recently, in 2015, Xilinx announced SDAccel [136] and SDSoC [137], both of which are targeted for software engineers. SDAccel takes as input OpenCL kernels and targets massively parallel applications that are typically used in data centres (similar to Altera's OpenCL SDK). SDSoC, on the other hand, is mainly used for embedded environments, where an SoC, comprising an ARM processor and hardware accelerators, is implemented on an FPGA. SDSoC is very similar to LegUp, in that it can automatically generate a complete SoC from software.

There are also an ample number of other commercial HLS tools. Bluespec Compiler [121] takes in Bluespec System Verilog (BSV) to generate hardware described in Verilog HDL. BSV is a high-level functional HDL based on Verilog, where modules are implemented as a set of rules. The rules are then used to express behaviour in the form of concurrently operating FSMs [58]. Catapult HLS [114], formally from Calypto Design Systems which was acquired by Mentor Graphics in 2015 [113], uses C/C++/SystemC code to generate hardware. This commercial tool has been used in the past by companies such as Google and Qualcomm [115, 36]. The Impulse CoDeveloper [123] uses Impulse-C, which is a C subset, to target image processing and streaming applications. CyberWorkBench [132] from NEC [131], uses C/SystemC as input, and can generate hardware for both Altera and Xilinx FPGAs, as well as for ASICs. DK Design Suite [116] uses Handel-C, a subset of the C which has been extended with hardware-specific constructs, to provide a software flow for compiling of software algorithms onto FPGAs. eXCite [150] takes an input C code and can generate either Verilog or VHDL. This commercial tool was used in a comparative study with LegUp [10], which is presented in Section 2.4.2. MaxCompiler [112] accepts MaxJ, a Java-based language, to generate synthesizable code for the data-flow engines on their

proprietary hardware platforms. Cadence's Stratus [110] integrates Cynthesizer, which was acquired from Forte [86], and Cadence's C-to-Silicon Compiler, into one HLS tool [134]. It uses C/C++/SystemC descriptions to generate hardware, targeting ASICs, SoCs, and FPGAs. The Synopsys Synphony C Compiler [117], which was acquired from Synfora [82], takes as input C/C++ code, and can also generate hardware for both ASICs and FPGAs.

On the academic front, there are also many HLS tools; we highlight a few which are being actively worked on. Bambu [61], from Politecnico di Milano, leverages the `GCC` compiler. It takes as input C code and generates hardware in Verilog. DWARV [57], from the Delft University of Technology, uses the CoSy commercial compiler infrastructure [118]. It supports a subset of C and can generate hardware described in VHDL. Bambu, DWARV, and LegUp were used in a comparative study [56], where the summary of results is presented in Section 2.4.3. FCUDA [28] is a source-to-source compiler that can translate CUDA to C, which can then be compiled to hardware by AutoPilot [30]. Lastly, GAUT [23] is tool from the University of Bretagne that uses C/C++ code to target digital signal processing applications.

2.2 Parallel Programming Languages in High-Level Synthesis

As previously mentioned, a number of different HLS tools have started to support parallel programming languages as input to HLS to produce parallel hardware. Namely, Altera's SDK for OpenCL and Xilinx's SDAccel take as input OpenCL code, FCUDA can receive CUDA code, and LegUp supports compiling Pthreads and OpenMP into parallel hardware. The different programming languages, which have different programming models, can have their own strengths and advantages. OpenCL and CUDA are well-suited for designing massively parallel applications where threads operate on multiple data elements in parallel. These languages provide support for vector data types, which can be used to express accessing multiple data elements in a concise and a convenient manner. However, both OpenCL and CUDA require the user to explicitly specify operations such as queuing kernels, creating buffers, and transferring data between host and device, as well as require an understanding of GPU programming concepts such as work-groups (for OpenCL) and blocks (for CUDA). For non-GPU programmers, this can be a challenging task. OpenMP, on the other hand, allows one to parallelize a section of code, such as a loop, with a single line of pragma. Thus OpenMP truly provides the ease-of-use for parallel programming. As for the Pthreads standard, it requires explicit thread forks and joins, but one can also specify task-level parallelism fairly easily, and Pthreads can give more control to the user, with synchronization mechanisms such as semaphores [80]. For some applications, any of OpenCL, CUDA, OpenMP, or Pthreads can be used to implement functionally equivalent designs, however, one should

consider the type of computations an application entails, as well as the ease of implementation associated with using a particular programming language, and choose the one which is the best suited for the case.

2.3 High-Level Synthesis vs. Hand-coded RTL

HLS tools provide the ability to automatically synthesize hardware from software. While this provides an easier design methodology, and offers a shorter time-to-market, many hardware engineers question the quality of hardware generated by HLS. To this end, several works have compared the quality of HLS-generated hardware to human-designed hardware. AutoPilot was used in [120] to implement a DQPSK (Differential Quadrature Phase-Shift Keying) Receiver Workload. This workload has a fixed required throughput to process modulated data at 18.75 Msamples/s at 75 MHz. AutoESL was used compile the C implementation of the application to hardware, which was compared to a manually designed RTL implementation. In terms of performance, both designs were able to meet the required throughput, and in terms of area, the HLS-generated hardware consumed 0.3% *less* area than the hand-coded RTL. Hence the HLS approach was able to produce a smaller circuit, while matching the performance.

In [1], Altera's OpenCL SDK was used to compile a Gzip algorithm [65] described in OpenCL to FPGA hardware. This HLS-generated hardware was compared to a handed-coded RTL implementation done by IBM researchers [48], which was considered to have the *best* publicly known results. The comparison showed that the OpenCL implementation was 5.3% lower in terms of performance (2.84 GB/s compression rate at 193 MHz vs. 3.0 GB/s at 200 MHz), and used 2% more logic and 25% more RAMs. Nevertheless, the complete OpenCL implementation was done in a month (one week for the initial kernel implementation, and three weeks for optimizations). With this, the authors state that designing hardware with OpenCL can be as easy as writing software code, where its performance can also be as good as hand-coded RTL.

Overall, several HLS tools have shown promising results, in terms of both performance and area, when comparing HLS-generated hardware to human-designed hardware. With continuous developments in the HLS domain, we think that there will soon be more application domains where HLS can be used to generate hardware that can match, or outperform, manually designed hardware.

2.4 LegUp High-Level Synthesis Framework

LegUp is an open-source HLS compiler framework under active development at the University of Toronto since 2009. First released in 2011, the tool is currently on its fourth public release, and is freely down-

loadable by the research community. LegUp has been the subject of 20 publications and has been downloaded over 4,000 times from around the world (`http://legup.eecg.toronto.edu`).

LegUp accepts a C program as input, which can be compiled to either: 1) A *hardware-only* system where the entire program is compiled to a hardware circuit, or 2) a processor-accelerator *hybrid* system where one or more functions are accelerated to hardware, with the remainder of the program executed in software on an embedded processor. The embedded processor can either be a *soft* MIPS processor, or a *hard* ARM processor. The software/hardware partitioning, as well as the generation of the complete SoC, including the processor, on-chip/off-chip memories, and interconnect, are automatically handled by LegUp.

LegUp is implemented within the LLVM compiler framework, an open-source compiler that was initially started as a research project at the University of Illinois at UrbanaChampaign (UIUC) [77], and is now being developed at Apple. Many companies, including Sony, NVidia, and Google, are using the compiler commercially. Other HLS tools, such as Xilinx's Vivado HLS [138], and Altera's OpenCL SDK [100], are also constructed within LLVM. Using LLVM allows one to leverage a robust, powerful and vetted compiler framework, which by default ships with more than 50 compiler optimization passes [38].

With respect to the language support, LegUp supports a large subset of `ANSI C`. It can synthesize to hardware most C constructs, including pointers, arrays, structs, functions, integer and floating point operations. It does not support, however, recursive functions or dynamic memory allocation/deallocation for hardware synthesis. However, LegUp is not unique in this regard as these features are typically not supported in HLS tools for compilation to hardware. The benefit of having the hybrid flow, however, is that program segments containing unsynthesizable constructs can be executed in software on the processor.

Figure 2.1 shows the hardware-only flow of LegUp. It comprises three major steps, *Frontend*, *IR transformations*, and *Hardware Backend*. The Frontend step invokes `Clang`, LLVM's front-end compiler, which receives a C program as input and produces LLVM IR (intermediate representation). The IR, which is LLVM's internal representation of the program, is target-independent assembly-like code, which can later be converted to machine-dependent assembly by the LLVM compiler. Clang can also be used to directly compile a program to an executable, like `gcc`. After converting C code to the LLVM IR, all HLS operations are performed on the IR. In the IR Transformations step, the program goes through a series of compiler optimization *passes*, each of which attempts to optimize the program. These passes include standard optimization passes included in LLVM (such as dead code elimination, or constant propagation), as well as custom passes that we have written ourselves to optimize for the output hardware. Finally, the optimized IR is input to the Hardware Backend step, which executes the HLS steps of *allocation*,

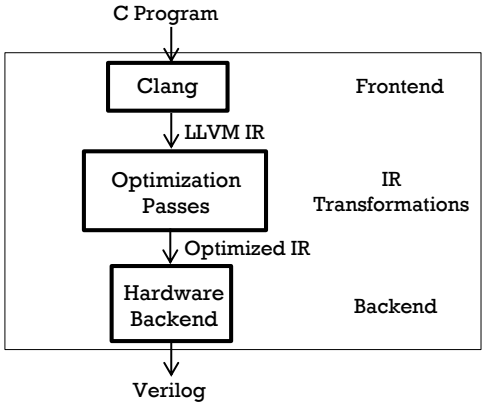


Figure 2.1: Hardware-only flow in LegUp HLS.

scheduling, and *binding*, to generate hardware specified in the Verilog hardware description language.

Allocation lays out the constraints on the HLS problem, for example by determining the amount of hardware that may be used to implement the circuit. LegUp reads in device-specific data from a configuration `Tcl` file, which specifies the target FPGA and the resource limits for the device. Scheduling assigns operations to specific clock cycles, using the target clock period constraint given by the user, as well as pre-characterized data specifying the delay for each type of operation on the target FPGA device. Given the clock period constraint, scheduling can also *chain* multiple instructions into a single cycle. For example, if a multiplication was characterized to take 7 ns, and a dependant addition was characterized to take 3 ns, with the given clock period constraint of 10 ns, both operations can be scheduled to execute in the *same* cycle, without violating the clock period constraint. After scheduling, binding performs the task of assigning operations to specific hardware units. When multiple operations are assigned to the same hardware unit, multiplexers are created to facilitate sharing. Finally, using the information gathered from allocation, scheduling, and binding, hardware is generated in Verilog.

Figure 2.2 shows the processor-accelerator hybrid flow of LegUp. In this flow, which was initially established in my M.A.Sc. thesis research [18, 17], LegUp automatically generates an *entire* SoC system, including an embedded processor, one or more hardware accelerators, memories, and interconnect. There have been a number of improvements to the flow since the prior work, including adding support for an ARM processor (initially the flow only had MIPS processor support), migrating to a newer version of Altera's system integration tool, Qsys [103], adding more optimization passes, as well as other general improvements. To use the hybrid flow, the user first designates one or more C functions for acceleration, then runs LegUp. Note that the first part of the flow (shown in the top left of the figure), where the C program goes through a series of optimization passes, and is given to the Hardware Backend, remains the same as the hardware-only flow shown in Figure 2.1. This allows us to perform the *identical* set of

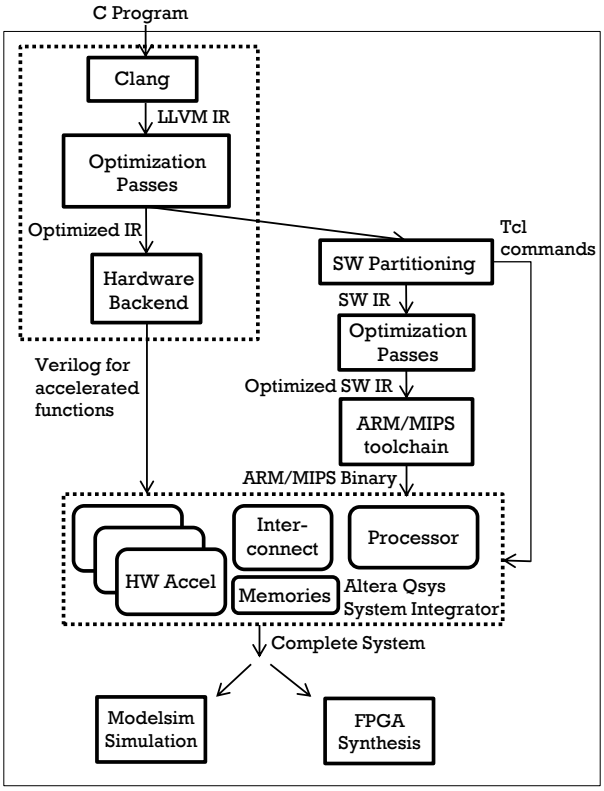


Figure 2.2: Processor-accelerator hybrid flow in LegUp.

compiler optimizations to the entire program (i.e. to both its software and hardware segments), with the steps specific to the hybrid flow simply added on as an *additional* procedure, allowing for easier maintainability. In the hybrid flow however, the Hardware Backend only compiles to hardware those functions designated for hardware acceleration. It also generates a wrapper module for each hardware accelerator, which contains Avalon interfaces (Altera's on-chip interface) to allow an accelerator to communicate with the processor and shared memories over the Avalon interconnect [4]. In the hybrid flow, the software program must also be modified to execute the accelerated functions in hardware instead of in software. To do this, we run the `SW Partitioning` pass on the Optimized IR. This pass partitions the software portion to be executed on the processor by removing the hardware-designated functions and replacing them with wrapper functions, which provides the means for the processor to communicate with the hardware accelerators over the interconnection fabric. The wrapper functions perform memory-mapped reads/writes over the Avalon Interconnect to transfer function arguments, start accelerators, and retrieve any return values.

As the `SW Partitioning` pass has knowledge of which functions are designated for hardware, it also generates `Tcl` commands, which are subsequently used by Qsys to generate the system. The generated SW IR from the `SW Partitioning` pass goes through more optimizations, such as *strength reduction* and

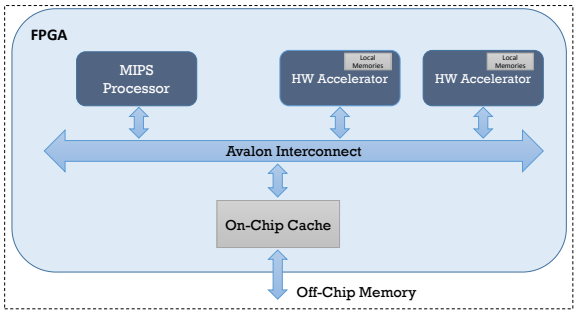


Figure 2.3: MIPS hybrid architecture.

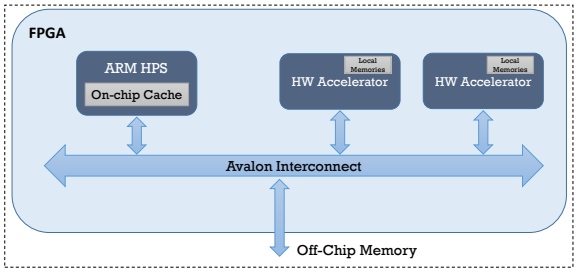


Figure 2.4: ARM hybrid architecture.

inlining, which can further optimize the software program, and the final Optimized SW IR is compiled with the ARM or the MIPS compiler toolchain (depending on the selected processor architecture) to generate the software binary. Once both the software and hardware partitions have been processed, Qsys is automatically invoked, using the previously generated `Tcl` commands as input, to generate the *complete* SoC system. Qsys instantiates the processor, memories, and hardware accelerators, and creates the Avalon Interconnect to tie all of the components together. Finally, the generated system can then either be simulated with ModelSim, with the testbench and its test vectors (compiled from the software binary) automatically generated to initialize the off-chip memory, or it can be synthesized with Altera's FPGA CAD tool, Quartus II, to produce the FPGA programming bitstream.

Figs. 2.3 and 2.4 show the general architectures of the MIPS and the ARM hybrid systems. In both systems, the processor communicates with the hardware accelerators over the Avalon Interconnect, where they also *share* an on-chip cache, backed by off-chip memory. An accelerator can also have *local* memories which are not shared with the processor, or with other accelerators. For the MIPS hybrid system, shown in Figure 2.3, the on-chip cache is implemented on the FPGA fabric, and can be customized as needed. For the ARM hybrid system, shown in Figure 2.4, the cache resides within the ARM Hard Processor System (HPS), which is a fixed architecture.

2.4.1 Key Features

It is worth highlighting several key features of the LegUp HLS tool. Although LegUp has many other features, including those used to visualize/debug the generated circuit, we only discuss the most notable and widely used features of the tool.

Push-Button SoC Generation

As described above, LegUp can generate a complete SoC from software with a *single* command, `make hybrid`. This permits a software engineer without hardware knowledge to create an entire processor-

accelerator system. Off-chip memory interfaces are automatically set up with correct DDR specs, and the testbench is generated so that input data can be loaded from DDR memory. One can choose to use a soft MIPS processor, or a hard ARM processor based on a `Tcl` parameter. The MIPS is an open-source processor [71] implemented on the FPGA fabric, and can be customized as needed. We support using the MIPS system on seven different Altera FPGA boards: DE2, DE2-115, DE1-SoC, DE4-230, DE4-530, DE5-Net, and the Arria V SoC Development Board. The hard ARM processor, on the other hand, runs at a much higher frequency than a soft processor and is therefore able to execute software much faster. Use of this processor is limited to those SoC FPGAs with the ARM HPS, and we currently support three Altera SoC FPGAs: DE1-SoC, SoCKit, as well as the Arria V SoC Development Board. The ARM processor on these SoC FPGAs has been configured so that it can be programmed via USB (described in Chapter 4). The user can also choose to run the ARM processor with an OS, or in bare metal. The push-button SoC generation is used extensively in the work described in Chapters 3 and 4. Recently, other commercial HLS tools, such as Xilinx's SDSoC [137], have also followed suit, by providing their own automatic SoC generation feature.

Loop Pipelining

Loop pipelining is an HLS technique for the synthesis of pipelined hardware that can execute multiple iterations of a loop concurrently, by commencing a new loop iteration before the preceding iteration is complete. By overlapping the execution of multiple loop iterations, loop pipelining significantly increases hardware utilization and performance. For high circuit performance, loop pipelining is a crucial HLS technique, because in many C applications, the *hot spots* are in loops. LegUp uses a loop pipelining scheduling algorithm with *backtracking* [9], developed by Andrew Canis. This scheduler, based on the SDC scheduling formulation [20], produces loop-pipelined hardware competitive to (and in some cases, superior to) that produced by a commercial HLS tool. Loop pipelining is used in Chapters 3, 4, 5 and 7.

Multi-Threading

In software, multi-threading is generally used to execute a program on multiple cores. As part of this thesis research, we have built an analogy to this behaviour in hardware, wherein parallel software threads can be used to *specify* multiple concurrently executing hardware cores. Our multi-threading feature in HLS allows a software engineer to use the parallelization techniques they are already familiar with to specify hardware parallelism. This is the overarching theme of the research in this thesis. Multi-threading support is described and used in Chapters 3, 4, 5, 6 and 7. We believe the capability to

synthesize Pthreads and OpenMP kernels into parallel hardware, in either a hardware-only system, or a processor-accelerator system, is unique to LegUp HLS[1]. Our multi-threading technique allows one to easily exploit hardware spatial parallelism on an FPGA purely from software. Multi-threading can also be combined with loop pipelining, to create multiple *pipelined* cores.

Bit-width minimization

Software variables utilize standard data types of predefined widths, such as `short` or `int`, regardless of whether all of the bits of the particular data type are used or not. This is acceptable for executing on a processor, where the data-path widths are fixed. However, for HLS-generated hardware, where the data-path can be customized as needed, using the pre-defined widths may unnecessarily result in high area consumption or poor performance. To address this issue, Marcel Gort, implemented automatic bit-width minimization [33], which analyzes the program at compile-time, leveraging program constants and the program's dataflow graph to automatically minimize data-paths widths of the generated hardware. This feature is not used for the work in this thesis; nevertheless, it is an important area-reduction feature in HLS and thus is mentioned here for readers who may wish to use LegUp in the future.

2.4.2 Software-to-Hardware Results

To give a sense of the quality of circuits produced by LegUp, and also to entice readers' interest in HLS, we discuss a few experiments that analyze the performance and area of hardware-only and hybrid systems generated by LegUp, as well as compare them to other HLS tools.

In 2011, with our *first* release of LegUp (LegUp 1.0), we conducted a study [10], where we measured the performance, area, and energy consumption of 13 benchmarks (12 CHStone benchmarks [37] and Dhrystone), as we successively moved portions of the program from software to hardware, until the entire program was compiled to hardware. For this experiment, we created five different architectures as shown below:

1. A software-only implementation where the entire program is executed in software on the soft MIPS processor (denoted as MIPS-SW in Figure 2.5).

2. A hybrid processor-accelerator hybrid implementation where the *second most* compute-intensive function (and its descendants) in the benchmark is implemented as a hardware accelerator, with

[1]Other HLS tools, such as Altera's OpenCL SDK, also allow multiple threads to be compiled to parallel hardware. However, it does not support creating a hardware standalone system, as it needs to create a full system with PCIe/off-chip memory interfaces that are *locked* down. This can be useful for data centre applications, but can be difficult for use in embedded/IoT domains.

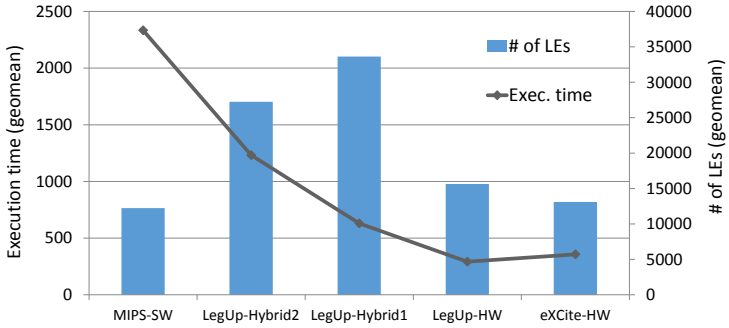


Figure 2.5: Performance and area results of hardware-only and hybrid systems generated by LegUp 1.0 and eXCite.

the balance of the program running in software on the soft-core MIPS processor (denoted as LegUp-Hybrid2).

3. Same as 2, but with the *most* compute-intensive function (and its descendants) compiled to a hardware accelerator (denoted as LegUp-Hybrid1).

4. A hardware-only implementation where the entire program is compiled to hardware by LegUp (denoted as LegUp-HW).

5. Same as 4, but with the program compiled with a *commercial* HLS tool, eXCite (denoted as eXCite-HW).

Note that eXCite [150] was the only commercial tool we had access to at the time that could compile the benchmark programs.

Performance and area results, in geometric mean across all benchmarks, are shown in Figure 2.5. The x-axis shows the five different scenarios described above; the left y-axis represents geomean execution time (for the *line* graph); the right y-axis displays area (for the *bar* graph), in terms of logic elements (LEs). The general trend is that, as more computations are mapped to hardware, the execution time improves. Comparing the hardware-only architecture of LegUp to eXCite, LegUp produced circuits with moderately better performance. In terms of area consumption, area increased as more computations are compiled to hardware, and abruptly dropped for the hardware-only architectures, as the processor system is no longer in the system. LegUp-HW consumed roughly 20% more area than eXCite-HW. The area-delay products of the two tools, which measures the *efficiency* of hardware, were nearly identical.

The geometric energy consumption results, shown in Figure 2.6, also show a similar trend to the performance results. Energy consumption is reduced drastically as computations are increasingly implemented in hardware. The LegUp-Hybrid2 and LegUp-Hybrid1 flows used 47% and 76% less energy

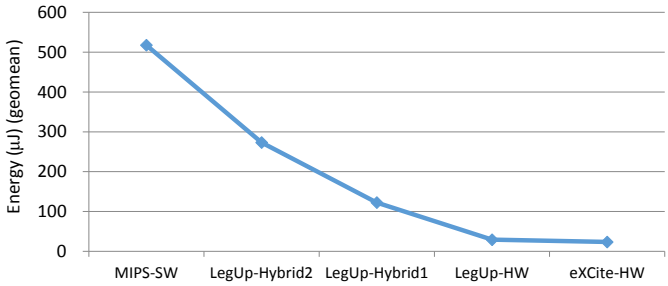


Figure 2.6: Energy consumption results of hardware-only and hybrid systems generated by LegUp 1.0 and eXCite.

than the MIPS-SW flow, respectively, representing 1.9× and 4.2× better energy-efficiency. LegUp-HW used 94% less energy and eXCite-HW also showed similar results to LegUp-HW (95% less energy).

These early results demonstrated that the first release of LegUp could produce circuits comparable to those generated by a commercial tool, and also showcased LegUp's usefulness as a tool for exploring the hardware/software co-design space.

2.4.3 Comparison to Other HLS tools

More recently, there has been ample activity in the HLS research community, with a variety of HLS tools available from both industry and academia. Although all HLS tools carry a common goal – making hardware design easier – many of these tools differ in terms of their input languages, and the types of features/optimizations they provide. This makes it challenging to compare HLS tools with one another, and for this reason, there had been no prior work on evaluating the performance of different HLS tools. Recognizing this, we conducted a study [56] in cooperation with two other academic institutions, the Delft University of Technology and the Politecnico di Milano, each of which has their own HLS tool, DWARV [57] and Bambu [61], respectively. Together, we used LegUp, DWARV, Bambu, as well as a state-of-the-art commercial HLS tool, to compare the results across a common set of 17 C benchmarks. For all of these benchmarks, each entire program was compiled to hardware, and we used LegUp 4.0, released in 2015. Note that LegUp targeted an Altera FPGA (Stratix V), whereas the three other HLS tools targeted a Xilinx FPGA (Virtex-7). Both FPGAs are fabricated using the 28 nm TSMC technology.

Figure 2.7 shows the geometric mean performance results, in terms of clock cycles, Fmax, and wall-clock time (cycles/Fmax), for the four HLS tools, where the results are normalized to the commercial tool. Compared to the commercial tool, LegUp HLS produced circuits with 35% less clock cycles, and 7% higher clock period, leading to 32% better average wall-clock time. In addition, LegUp-generated

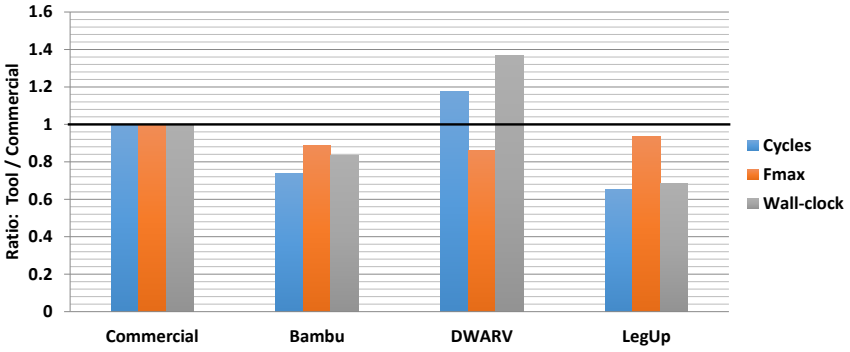


Figure 2.7: Performance comparison of circuits generated by four HLS tools, including LegUp 4.0.

circuits also outperformed those produced by Bambu and DWARV, on average. LegUp used techniques such as loop pipelining and multi-threading to generate these circuits, which significantly improved performance. Although area results are also reported in [56], it was difficult to bring forth conclusions due to the differences in the target FPGA architectures. Nevertheless, we showed that LegUp can produce circuits which are competitive or superior to those of several other currently available HLS tools.

2.5 Summary

In this chapter, we gave a high-level overview of a number of currently available HLS tools, with a discussion on the different parallel programming models which are supported in the HLS tools. We also explored a few cases studies which have compared the quality of HLS-generated hardware to human-designed hardware. We also provided a brief overview of LegUp HLS, and noted some of its most important features. Lastly, we highlighted a few comparative studies of LegUp with other HLS tools, which showed encouraging results for LegUp HLS.

Chapter 3

From Software Threads to Processor/Parallel-Accelerator Hybrid System

3.1 Introduction

With the end of the clock frequency scaling in the last decade, single-thread performance has stopped improving, necessitating the need for parallelization via multi-core processing. However, software compilers have a limited ability to infer parallelism, and even auto-parallelizing compilers, which aim to relieve programmers from the error-prone parallelization process, generally only work on loops that are structured in certain styles [93, 90]. As such, it is normally incumbent on the programmer to explicitly specify coarse-grained parallelism within the code. Common parallelization approaches include using parallel programming languages such as OpenCL [111] or CUDA [133], or the use of libraries like Pthreads [7] and OpenMP [85].

A question that naturally arises then is, how does one specify parallelism to an HLS tool? Fine-grained parallelism, such as instruction-level parallelism, or loop-level parallelism with loop pipelining, are handled quite well by current state-of-the-art HLS compilers. What is lacking, however, is a standard way in HLS to specify coarse-grained parallelism, such as at the thread-level or at the task-level. Coarse-grained parallelism in HLS is often realized by using an HLS tool to synthesize a single hardware core, and then manually instantiating multiple instances of the core in structural HDL – an approach which

requires knowledge of hardware design. Some HLS tools support the use of vendor-specific pragmas to create parallel hardware, which while useful, is a non-standard methodology [144].

We believe the correct approach to parallelism specification is to bring forth a solution which is not only vendor-agnostic, but also platform-agnostic – the same code which is used to execute in parallel in *software* can be used to execute in parallel in *hardware*. To this end, we propose using Pthreads and OpenMP, standard parallel programming methodologies that software engineers are already familiar with, for the specification of hardware parallelism to an HLS tool. We provide an HLS framework wherein the parallelism described in the software code is automatically synthesized into parallel hardware accelerators that perform the corresponding computations concurrently. In this chapter, our research is in the context of the *MIPS* processor-accelerator hybrid system, where threads are compiled to parallel accelerators, and remaining (sequential) portions of the program are on the MIPS processor. Note that the MIPS processor runs bare metal, without an OS. We describe support for using an *ARM* processor, with and without an OS, in Chapter 4.

Writing deterministic parallel software often requires the use of synchronization constructs that, for example, manage which threads may execute a given code segment at a given moment. Recognizing this, we also provide HLS support for two key thread synchronization constructs in the Pthreads/OpenMP library: mutexes and barriers. Our work represents a key step towards improving the performance of hardware that can be created by an engineer who solely possesses software skills.

3.2 Background

Several prior works consider the use of OpenMP for FPGAs. The work in [29] implements an extension to OpenMP so that computations can be off-loaded to an FPGA, although it was not in the context of high-level synthesis and required the FPGA hardware to be designed manually using HDL. The work in [31] describes a framework that generates Handel-C and VHDL from programs that use OpenMP, where the generated code is implemented on an FPGA. Although the work bears some similarity to our own, it has significant limitations, namely, it supports only the *integer* data type, the target hardware has no memory subsystem, and the hardware FSM allows only one statement in each state, making it impractical for use in a real system. Our framework does not have such limitations. The work in [27] describes a source-to-source translator that accepts a C program that uses OpenMP as input, and generates source files to be passed to an HLS tool. The authors do not provide the HLS capability themselves, but instead use a commercial tool: Impulse CoDeveloper [123].

Concerning Pthreads, [68] describes a framework which employs Pthreads to generate hardware

accelerators at *runtime* by running an FPGA CAD tool on an embedded ARM processor with an OS, also on the FPGA. However, the FPGA synthesis tool could not actually run on the embedded ARM processor, hence their results are based on their own C++ simulator, rather than from cycle-accurate simulation with ModelSim or on-board execution in silicon. Moreover, the work was done before ARM SoC FPGAs were introduced, and thus, even the ARM software execution was simulated rather than actual. In our work, we use cycle-accurate ModelSim simulation (and on-board execution in Chapter 4) to obtain accurate runtimes. HybridThreads (hthreads) provides a library to execute threads on a hybrid CPU/FPGA system [6]. While similar to Pthreads, hthreads is not a standard software library, thus is not portable to other platforms. The works in [39, 46] provide OS abstractions for communication between a CPU and hardware threads on an FPGA. Each provides their own thread APIs, neither of which are standard software APIs, and they also do not provide the capability to compile software threads to hardware.

It is also worthwhile to comment on the parallelization capabilities of the HLS tools offered by the two main commercial FPGA vendors. Xilinx's Vivado HLS [138] provides a rich set of features such as pipelining, memory partitioning/restructuring, arbitrary precision types, as well as supports other user-specified pragmas to control the generated hardware. Hardware knowledge is needed to fine-tune the hardware using pragmas and there is currently no support for standard software APIs to specify parallel execution. Altera's OpenCL SDK [100] and Xilinx's SDAccel [136] permit the compilation of OpenCL kernels to FPGA hardware. Parallelism is explicitly specified by the programmer in OpenCL, which is compiled to pipelined hardware units. Related to the Altera effort, [28] provides a source-to-source compiler to translate CUDA code into annotated C code to be input into another HLS tool, AutoPilot [30].

To our knowledge, there is no prior work that offers an open-source HLS tool with support for parallelism expressed using the Pthreads or OpenMP standards.

3.3 Parallel Programming with Pthreads/OpenMP

This section briefly describes Pthreads and OpenMP and illustrates how they can be used to express parallelism in software. We focus on the most widely-used aspects of the two parallelization approaches, which are the same aspects we have selected for automated synthesis to hardware.

Consider the following code snippet that uses Pthreads:

```
// fork threads
for (i=0; i<N; i++) {
```

```
  pthread_create(&threads[i], NULL, vector_add, &data[i]);
}

// join threads
for (i=0; i<N; i++) {
  pthread_join(threads[i], NULL);
}
```

The example code forks `N` threads with `pthread_create`, each of which executes the `vector_add` function, with a pointer to an element of the `data` array given as its argument. Multiple arguments can be supplied to a function by aggregating them into structures (use the C `struct` datatype). The `N` threads are joined with `pthread_join`, which waits for the thread specified by its thread variable (`threads[i]` in this case) to terminate. The return value from the threaded function can be retrieved via the second argument of `pthread_join` (`NULL` in this case).

In parallel programming, synchronization mechanisms are used to ensure correct execution of a multi-threaded program, with locks and barriers being the most commonly used mechanisms. With Pthreads, locks are specified with the `pthread_mutex_lock` and the `pthread_mutex_unlock` functions. A barrier is used to synchronize threads at a specific point in a program. The `pthread_barrier_init` is used to initialize the barrier with the number threads that must wait at a barrier. The `pthread_barrier_wait` function is used to synchronize threads at a barrier.

Turning now to OpenMP, consider the following code segment:

```
#pragma omp parallel for num_threads(2) private(i)
for (i = 0; i < SIZE; i++) {
  output[i] = A_array[i]*B_array[i];
}
```

The loop performs a dot product of two arrays, `A_array` and `B_array`. To parallelize this loop using OpenMP, one simply puts an OpenMP pragma, `omp parallel`, before the loop, as shown in the example. The `num_threads` clause specifies the number of threads to execute the loop in parallel. In the example code with two threads, the first thread computes the first half of the array; the second thread works on the second half. The `private` clause privatizes one or more variables to each thread. In the example, each thread has it own copy of the induction variable `i`. Note that the `parallel` pragma in OpenMP is blocking – all threads executing the parallel section need to finish before the program can continue to execute.

Table 3.1: Pthreads/OpenMP support in LegUp.

Pthreads Functions	**Description**
pthread_create(..)	Invoke thread
pthread_join(..)	Wait for thread to finish
pthread_exit(..)	Exit from thread, can be used to return data
pthread_mutex_lock(..)	Lock mutex
pthread_mutex_unlock(..)	Unlock mutex
pthread_barrier_init(..)	Initialize barrier
pthread_barrier_wait(..)	Synchronize on barrier object
OpenMP Pragmas	**Description**
omp parallel	Parallelize a section of code
omp parallel for	Parallelize a for loop
omp master	Parallel section executed by master thread only
omp critical	Specify a critical section
omp atomic	Specify an atomic section
reduction(operation: var)	Reduce a var with operation
OpenMP Functions	**Description**
omp_get_num_threads()	Get number of threads
omp_get_thread_num()	Get thread ID

OpenMP provides mutual exclusion capability with two pragmas: `omp atomic` and `omp critical`. The atomic pragma permits the specification of a single-statement critical section, whereas a multiple-statement critical section can be specified with the `critical` pragma. OpenMP also provides the `reduction` pragma, which can be applied to perform a reduction operation.

As illustrated, OpenMP provides a simple high-level approach for parallelization. With a single pragma, the user is able to parallelize a section of code without complicated code changes. Although the OpenMP code above showed a simple example with a single statement inside the loop body, OpenMP can also be used to parallelize more complex cases, such as a loop with a function call in the loop body, in which case the called function is executed in parallel.

On the other hand, Pthreads requires explicit forks and joins of threads, requiring more work from the programmer, but it also gives more fine-grained control. Also, forking threads in Pthreads is *non-blocking*, which can be useful in many scenarios.

Table 3.1 shows all Pthread/OpenMP APIs that we support in our work, in the sense that we are able to synthesize programs containing the listed constructs. It is worth noting that an input software program using any of the functions/pragmas shown in the table can be synthesized to hardware *as is*, without requiring any manual code changes by the user.

3.4 Parallel Threads to Parallel Hardware

Prior to this work, LegUp HLS was only able to exploit instruction-level parallelism, and loop-level parallelism via loop pipelining. The current work greatly expands the extent to which a user may specify parallelism to the HLS tool. By default, LegUp creates as many hardware accelerators for a Pthread function, or an OpenMP loop, as the number of threads.

In software, using Pthreads requires the `POSIX` library, whereas OpenMP requires the `libgomp` library, and in both cases, an OS is used to manage threads at runtime. Since neither libraries exist in hardware, and we do not use an OS on the MIPS processor, we need to implement our own custom logic to allow Pthread functions and OpenMP loops to run in parallel in hardware. Out of the many steps that are required for LegUp HLS to create a processor/parallel-accelerator system, two main steps pertain to making the management of threads in hardware possible, the `ParallelAPI` and the `SW Partitioning` compiler passes. The duo are custom LLVM compiler optimization passes that we have written, which are run as part of the hybrid generation flow, as shown in the highlighted red boxes in Figure 3.1. The `ParallelAPI` pass first transforms the Pthread/OpenMP functions in the LLVM IR into LegUp-specific versions. After a series of other optimization passes, the resulting IR is read in by the `SW Partitioning` pass, to generate the wrapper functions that call the Pthread/OpenMP hardware accelerators. The combination of work done in the two compiler passes enable Pthreads and OpenMP kernels to become concurrent hardware accelerators that execute in tandem with the MIPS processor, without an OS, and without Pthread/OpenMP libraries. We elaborate on the flow below.

3.4.1 Generation of Thread-Handling Logic

As shown in Figure 3.1, the `ParallelAPI` pass is one of the compiler optimization passes run as part of the *IR Transformations* step, as described in Chapter 2. It takes as input a program represented in LLVM IR, and outputs a *transformed* program in LLVM IR. Although we only discuss the hybrid flow in this chapter, the same code in the `ParallelAPI` pass is also used to enable the creation of parallel hardware modules in the hardware-only flow (described in Chapter 5).

The `ParallelAPI` pass is responsible for two major tasks: 1) Remove the dependency on the Pthreads and OpenMP libraries, and 2) creating logic to manage threads without an OS. For 1), the pass replaces the calls to Pthread/OpenMP library functions, with direct calls to the threaded functions (functions executed on threads). For instance, in the Pthread example shown in Section 3.3, `pthread_create` is used to execute the `vector_add` function in parallel. In this case `pthread_create` is replaced with a direct call to `vector_add` (i.e. pthread_create(&threads[i], NULL, vector_add, &data[i]) becomes vec-

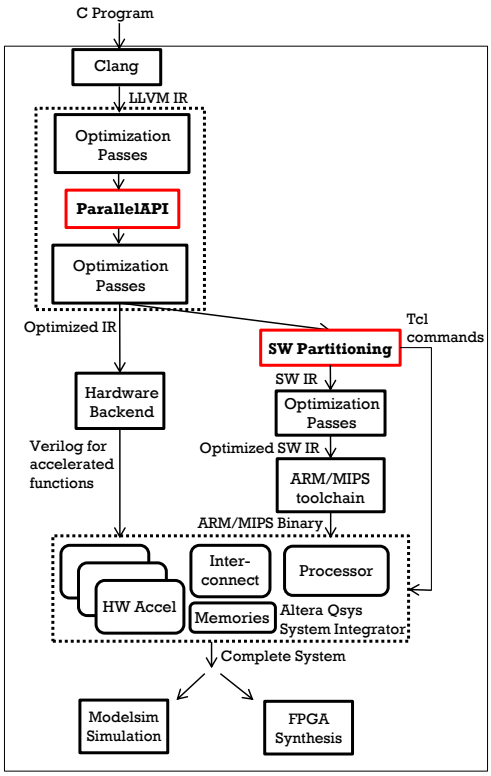


Figure 3.1: The `ParallelAPI` and the `SW Partitioning` compiler passes in the Processor-accelerator hybrid flow.

tor_add(&data[i])). `Pthread_join` is replaced with `legup_pthreadpoll`, which waits until its hardware instance is finished, and retrieves its return value if needed.

For OpenMP, the frontend compiler[1] transforms the OpenMP pragmas into OpenMP library function calls. When the `omp parallel` pragma is used on a loop, the compiler *outlines* the body of the loop to a separate function, and inserts a call to an OpenMP function, `GOMP_parallel_start`, invoking the outlined function. The `GOMP_parallel_start` function marks the start of a parallel section. Its function prototype is:

```
void GOMP_parallel_start (void (*fn)(void *), void *data, unsigned num_threads)
```

Here, `fn` is the outlined function to be executed in parallel, `data` is a pointer to a struct used to communicate data into and out of the outlined function (passed in as its argument), and `num_threads` specifies the number of threads to execute the function in parallel [88]. `GOMP_parallel_start` is immediately followed by a call to another OpenMP function, `GOMP_parallel_end`, which marks the end of the parallel section. On a processor, this makes the program wait until all threads have finished

[1]Note that for OpenMP, we use the `gcc` frontend with `dragonegg` [72], a `gcc` plugin that allows the compiler to output LLVM IR, as Clang did not support OpenMP at the time that this work was done. Since then, Clang has implemented OpenMP support [84].

execution, creating the blocking behaviour of the `omp parallel` pragma. In LegUp, we transform `GOMP_parallel_start` and `GOMP_parallel_end` function calls into direct calls to the outlined function (with as many calls to the function as the number of threads), and insert polling logic immediately after the last call, so the processor will wait until all hardware accelerators have finished execution (described in the next section).

The Pthread/OpenMP mutex and barrier functions are also handled in a similar manner, where they are replaced with calls to our own mutex and barrier functions. These functions communicate with our *hardware* mutex/barrier cores via memory-mapped loads and stores. The hardware mutex core itself is quite simple: it contains a register that holds the unique thread ID of the hardware accelerator that holds the lock, and has a flag to indicate its state (locked/unlocked). The `ParallelAPI` pass replaces the Pthread/OpenMP lock/unlock functions shown in Table 3.1 with `legup_lock` and `legup_unlock`. Our lock/unlock functions perform memory-mapped reads and writes to acquire and release a lock. Both functions have two arguments: the processing element ID, and the mutex index. The processing element ID is a *unique* ID for the processing element accessing the mutex, which can be an accelerator or the processor. We use the memory-mapped address of the processing element as its ID (described in the next section). The mutex index is used to differentiate between multiple mutex cores, which is generated when multiple mutex variables are used in the input software program. When a processing element calls the lock function, it first tries to write its ID to the mutex core corresponding to the mutex index. If the mutex is free, the write is successful and the ID is stored. If the mutex is already locked, the mutex core retains the previously stored ID. After the write, the processing element reads from the mutex to check if the stored ID matches its own ID. If there is a match, this indicates that the processing element has acquired the lock and is free to enter the critical section. If the processing element fails to get the lock, it repeats the locking procedure until it gets the lock (a behaviour akin to *spin locks*). In our unlock function, the processing element again writes to the mutex core with its ID. If the mutex is locked with the matching ID, this unlocks the mutex. Pthread barrier functions, `pthread_barrier_init` and `pthread_barrier_wait`, are likewise automatically replaced with `legup_barrier_init` and `legup_barrier_wait`, which are used to communicate with a hardware barrier core. The hardware barrier core contains a register, which is used to hold the number of threads that must wait at the barrier. This register is initialized when `legup_barrier_init` is called (`pthread_barrier_init` specifies the number of threads as its argument). The barrier core also contains a counter, which is incremented each time a processing element reaches the barrier. When `legup_barrier_wait` is called, the processing element first writes to the hardware barrier core to increment its counter, then it keeps polling on the barrier core, which returns a `1` until enough threads have reached the barrier. When the counter equals the number of threads, the barrier

core returns a `0`, at which point the processing elements can continue to execute. The barrier core also resets its counter to zero, so that it can be used again for the same barrier object.

The second task of the `ParallelAPI` pass is to generate logic to manage threads in hardware without an OS. For OpenMP, the `GOMP_parallel_start` function call exists for every use of the `omp_parallel` pragma, and each use of the pragma creates a *new* outlined function (using the `omp_parallel` pragma on two different loops creates two different outlined functions). Hence, there is a *one-to-one* mapping between a call to the `GOMP_parallel_start` function, and the outlined function to be executed in parallel. Furthermore, `GOMP_parallel_start` explicitly specifies the number of parallel threads in its argument. Consequently, for each `omp_parallel` pragma, we know exactly how many accelerators to create, and where the accelerators are called from (there is only one call site). This also aids in the handling of `omp_get_num_threads()` and `omp_get_thread_num()`, which are OpenMP library functions that return the total number of threads to execute a parallel section, and the thread ID for a particular thread. The number of threads are explicitly given as an argument to `GOMP_parallel_start` (which we use to create as many calls to the outlined function), hence, `omp_get_num_threads()` can be directly replaced with that value, and for each call to the outlined function, we statically assign a thread ID, from `0` to `num_threads - 1`, and pass it in as an additional argument into the outlined function. Hence, we create a new thread ID argument for the outlined function, then replace all calls to `omp_get_thread_num()` with the new argument. Once all OpenMP library functions have been replaced, we do not need to create additional logic to handle threads, other than the wrapper generation that is handled by the `SW Partitioning` pass (described in the next section).

For Pthreads, the required machinery is more complicated, as there is no one-to-one mapping between a call to `pthread_create` and the function to be executed on the thread. There can be many calls to `pthread_create`, each of which executes the *same* or *different* functions, and even the calls to same the function can reside in different parts of the program. More importantly, at compile time, `pthread_join` does not "know" which function it intends to join. This is determined at runtime based on the thread variable passed in as its argument. Hence, we need a methodology where we can keep track, at runtime, of which thread is accelerating which function, as well as which thread is running on which *instance* of the hardware accelerator for that function (as in LegUp, by default, we create as many instances of accelerators as the number of threads). To perform the needed bookkeeping, we propose using memories. We create a global variable in the LLVM IR for each *different* Pthread function (i.e. a single variable is created for `vector_add` in the example code, but when two different Pthread functions are called, two variables are created), which essentially acts as the *thread ID* counter for that function. Initialized to zero, the thread ID is incremented each time after calling its Pthread function. The value of the thread

ID, along with its thread variable (i.e. threads[i] in the example code), is passed in as arguments to the wrapper function which calls the Pthread accelerator. The thread ID permits tracking of which thread is using a particular instance of a Pthread accelerator, and we store this information to the thread variable from within the wrapper function. In essence, we make use of the existing thread variable in our system, for the same purpose it was originally intended for use in software. This is described in more detail in the next section. `Legup_pthreadpoll` is also passed the thread variable as its argument, which is used to determine the specific accelerator to poll.

3.4.2 Wrapper Function Generation for Parallel Accelerators

After all Pthread/OpenMP functions have been replaced, and the necessary logic to keep track of threads has been created in `ParallelAPI`, the next part of the work is done in the `SW Partitioning` pass. This pass generates wrapper functions for the MIPS processor to communicate with parallel hardware accelerators. After generating the wrapper functions, the pass removes from the original definitions of the accelerated functions from software, and replaces the calls to the original functions with calls to the wrapper functions. In addition, the pass assigns a *unique* base memory-mapped address to each hardware accelerator (whether it is a sequential, a Pthread, or an OpenMP function). From this base address, each accelerator is assigned an address *range*, where a number of registers, associated with the function, are memory-mapped to. The `argument` memory-mapped registers are used to transfer function arguments, the `status` memory-mapped register is used to start the accelerator, and also to check if the accelerator is done, and the `data` memory-mapped register is used to retrieve the return value.

For OpenMP, the pass generates a wrapper function for each function that was outlined due to the OpenMP pragma. An OpenMP wrapper function is used to call *all* hardware accelerator instances which belong to the same OpenMP function, as well as to wait for all of them to finish execution – analogous to the behaviours of `GOMP_parallel_start` and `GOMP_parallel_end` in software. The pseudocode for the generated wrapper function is shown below. Note that this wrapper is generated in LLVM IR and not in C code[2].

```
1:  function legup_omp_wrapper ( omp_argument )
2:    for each accelerator_instance from i=0 to num_threads-1
3:      base_memory_mapped_address of accelerator_instance at (i) =
          base_memory_mapped_address of accelerator_instance at (i=0) + i * constant
4:      status_register = base_memory_mapped_address of accelerator_instance at (i)
```

[2]The initial work in [14] generated C wrapper functions, but since then, we have moved to generating the wrappers directly in LLVM IR, which is more robust and allows more optimizations.

```
5:      arg_register = status_register + constant_offset;
6:      store omp_argument to arg_register
7:      store 1 to status_register
8:    end for
9:
10:   for each accelerator_instance from i=0 to num_threads-1
11:     base_memory_mapped_address of accelerator_instance at (i) =
          base_memory_mapped_address of accelerator_instance at (i=0) + i * constant
12:     status_register = base_memory_mapped_address of accelerator_instance at (i)
13:     poll on status_register until it return 1
14:   end for
15: end function
```

The OpenMP wrapper function has two loops, one to call its accelerators (calling loop, shown on lines 2–8), and one to poll to check if the accelerators are done (polling loop, shown on lines 10–14). The loop bounds of the calling/polling loops are from the `num_threads` argument given to `GOMP_parallel_start`, and the induction variable `i` of the loops acts as the thread ID for the OpenMP function. The calling loop iterates over each of its accelerators by offsetting the base memory-mapped address of the *first* accelerator for this OpenMP function, with the thread ID (multiplied by a *constant*, where the value of the constant depends on the memory-mapped range of an accelerator instance) (line 3). For each iteration of the loop, the processor communicates with a different instance of an accelerator for an outlined OpenMP function. There is a single wrapper function for all accelerators of an OpenMP function, rather than a wrapper for each individual OpenMP accelerator. For each accelerator, the calling loop uses the `argument` register to transfer its argument, then gives the *start* signal by writing a `1` to the `status` register (lines 4–7). Note that in some cases, the thread ID itself may be needed on the accelerator side, in which case, the induction variable `i` is passed in using another argument register. Immediately after the calling loop, the *polling* loop commences, which again iterates over each accelerator, each time polling on the `status` register to check if the accelerator is done its work (lines 10–14). An OpenMP function does not have a return value, instead, any outputs can be written to the pointer argument (`omp_argument` in the pseudocode). When the polling loop exits and the wrapper function returns, it indicates that all parallel accelerators for the outlined OpenMP function have finished execution.

For Pthreads, a wrapper function is generated for each different Pthread function (calling wrapper), and a single wrapper function is created for the call to `legup_pthreadpoll` (polling wrapper), used for

joining all Pthread accelerators. The calling wrapper invokes a *single* accelerator instance for a particular Pthread function, whereas a call to the polling wrapper can poll on a single accelerator instance for *any* Pthread function. This is also analogous to the behaviour of their software equivalents – a particular call to `pthread_create` is tied to a specific function (given as its argument), where as `pthread_join` can be used to join any threads, regardless of which function is executed by the thread. We show below the pseudocode for a Pthread calling wrapper for `vector_add`, as well as a polling wrapper.

```
1:  function legup_pthread_call_wrapper_vector_add (
      pthread_argument, thread_variable_ptr, thread_ID )
2:    base_memory_mapped_address =
        base_memory_mapped_address of vector_add + thread_ID * constant
3:    status_register = base_memory_mapped_address
4:    arg_register = status_register + constant_offset1
5:    store pthread_argument to arg_register
6:    store 1 to status_register
7:    store base_memory_mapped_address to thread_variable_ptr
8:  end function
9:
10: function legup_pthread_poll_wrapper ( thread_variable_ptr )
11:   load base_memory_mapped_address from thread_variable_ptr
12:   status_register = base_memory_mapped_address
13:   data_register = status_register + constant_offset2
14:   poll on status_register until it returns 1
15:   load accelerator return_val from data_register
16:   return return_val
17: end function
```

As previously mentioned, the *calling* wrapper is passed in as its arguments a thread ID and a thread variable, in addition to any arguments used by the Pthread function (line 1). Similar to the OpenMP case, the memory-mapped address of an accelerator instance for a particular thread is calculated by offsetting the base memory-mapped address of the Pthread function by the thread ID (line 2). Using this address with its offset, the calling wrapper transfers the function arguments and gives the start signal (lines 3–6). Then, it stores the calculated memory-mapped address of this accelerator instance into the *thread variable* (line 7). This tracks the thread that corresponds to the hardware accelerator.

Subsequently, the *polling* wrapper, which is also passed in a thread variable as its argument, loads from the variable, which returns the memory-mapped address of the accelerator instance for the thread (lines 10–11). Reading from this address allows the processor to communicate with the particular accelerator it needs to join. The polling wrapper then polls on the address until the accelerator is done, and retrieves its return value, if necessary (lines 14–15).

Once the `SW Partitioning` pass has finished, we will have performed all necessary software transformations in LLVM IR to allow the processor to fork and join Pthreads and OpenMP hardware accelerators. The transformed IR, shown as `SW IR` in Figure 3.1, is taken through a few more compiler optimization passes, and finally compiled with the MIPS toolchain, generating the MIPS binary to be executed on the processor.

3.4.3 Parallel Accelerator Instantiations

Now that the software binary is ready, we need to create the parallel accelerators and generate the overall system. During its execution, the `SW Partitioning` pass gathers information on the total number of accelerator instances needed for each Pthread/OpenMP function, and assigns a unique memory-mapped address to each instance. The number of accelerators need to be determined at *compile time*, hence if Pthread functions are called in a loop, the loop bound must be a compile time constant. The user can also specify the number of accelerators that are instantiated for a function, which is described in the next section. If mutexes or barriers are used, memory-mapped addresses are also assigned for the mutex and barrier cores. This information is output as a script containing `Tcl` commands to be used by Altera's Qsys system builder (as shown on the right side of Figure 3.1). The `Tcl` commands also specify the connections required between the components, such as the links between the MIPS processor and hardware accelerators, links between processor/accelerators and mutex/barrier cores, as well as the links from accelerators to memory.

As shown on the left side of Figure 3.1, the Hardware Backend generates hardware for each Pthread and OpenMP function. For each accelerator, it also generates a top-level wrapper module, which contains Avalon Interfaces to permit communication between the processor and the accelerator, as well as between the accelerator and shared memories (on-chip cache backed by off-chip memory). Once the hardware accelerator is created, the entire system is ready for generation.

Using the `Tcl` script, as well as the hardware accelerators, Qsys generates the complete processor-accelerator system. It instantiates the processor, hardware accelerators, mutex/barrier cores, an on-chip cache, and an off-chip memory controller, and creates the Avalon Interconnect to connect all components

together.

The entire flow, starting from parallel software that uses Pthreads/OpenMP, to generating a complete processor/parallel-accelerator system is be done automatically by our framework, without requiring *any* manual interactions from the user. In terms of runtimes, LegUp runs pretty quickly (i.e., typically less than a minute, mostly on the order of seconds to tens of seconds depending on the size/complexity of the input program), and Qsys generation takes around a few minutes, hence the entire SoC can be generated in a short amount of time.

3.4.4 Sharing an Accelerator Across Threads

So far, we have described an automated synthesis flow where the number of generated accelerators, or hardware *cores*, are exactly equal to the number of threads in software. However, for software executing on a regular processor, a user can freely fork more threads than the number of available cores, with the OS handling the scheduling and context switching of threads, allowing multiple threads to *share* a core. This time multiplexing of cores may also be important for hardware, where for an area-constrained design, the user may not always want to create as many accelerators as the number of threads. Also, Pthreads in software may be forked and joined many times from different parts of the code to execute the *same* function, in which case it would be inefficient to create as many hardware accelerators as the number of threads each time. Recognizing this, we also provide an option where a user can *constrain* the number of hardware accelerators created for a Pthread function. This can be specified with the following `Tcl` parameter:

```
set_accelerator_function "function_name" --numAccels max_number_of_instances
```

With this parameter, LegUp will only create as many accelerators for `function_name` as given by `max_number_of_instances`.

The required functionality is bundled into the `ParallelAPI` and the `SW Partitioning` compiler passes. As previously described, the `ParallelAPI` pass generates thread ID counters for each Pthread function. The value of the thread ID counter determines which accelerator instance the processor communicates with. For example, when the thread ID is zero, the processor communicates with the *first* accelerator for the particular Pthread function, and when the thread ID is incremented to one, the processor communicates with the *second* accelerator for the same Pthread function. Hence, limiting the maximum value of the thread ID limits the number of accelerator instances the processor can use. Therefore, when the `Tcl` parameter is used, the `ParallelAPI` pass inserts LLVM instructions which resets the *thread ID* counter to zero once it reaches `max_number_of_instances`. This means that when there

are more calls to a Pthread function than the number of available hardware accelerators, the additional calls will attempt to *re-use* the previously called hardware accelerators. If an accelerator being called is already running, the processor waits until the accelerator completes its execution. The *waiting* logic is generated by the `SW Partitioning` pass. As described in Section 3.4.2, the `SW Partitioning` pass generates a calling wrapper for each Pthread function and a polling wrapper for all Pthread functions. When sharing is enabled for a Pthread function, we modify its calling wrapper, so that before the arguments are transferred to the accelerator, the processor polls on the accelerator to check if it is done – just like the polling loop created inside the polling wrapper. If an accelerator to be called is already running, the processor waits until the accelerator is available for use, and only then, it proceeds to call the accelerator. Lastly, the `Tcl` commands generated in the `SW Partitioning` pass for Qsys are modified to only create only as many accelerators as specified by `max_number_of_instances`.

Note that there are a few caveats with sharing a Pthread accelerator across threads. First, a Pthread accelerator that is shared cannot have a *return value*. This is because subsequent uses of the accelerator can overwrite the previous return values, if the accelerator is invoked again before being *joined* by the polling wrapper (which retrieves the return value). This, however, can be circumvented via software changes alone, by storing the return value into memory instead. Secondly, for a particular execution, a shared accelerator is unaware of the thread the execution is for (out of the threads that share the accelerator), which would be problematic if the execution is thread dependant. This can be addressed by passing in the thread ID as an argument into the accelerator. The thread ID is already passed in when a mutex is used. Despite these limitations, we believe the above capability to share a hardware accelerator across threads is useful in many scenarios.

3.4.5 System Architecture

The overall architecture of the generated system is shown in Figure 3.2, comprising the MIPS processor and hardware accelerators sharing an on-chip data cache backed by off-chip memory. An accelerator may also have local memories for data not shared with the processor or other accelerators. The local memories are implemented in on-chip block RAMs instantiated within a hardware accelerator. Shared data, on the other hand, is stored in off-chip memory, which can be fetched into the on-chip cache. The components of the system communicate via the Avalon Interconnect, which is a point-to-point network, allowing multiple independent transfers to occur simultaneously. When multiple components are connected to a single component, such as the on-chip data cache, a round-robin arbiter is created by Qsys to arbitrate among concurrent accesses. For memory-intensive applications, the default dual-port

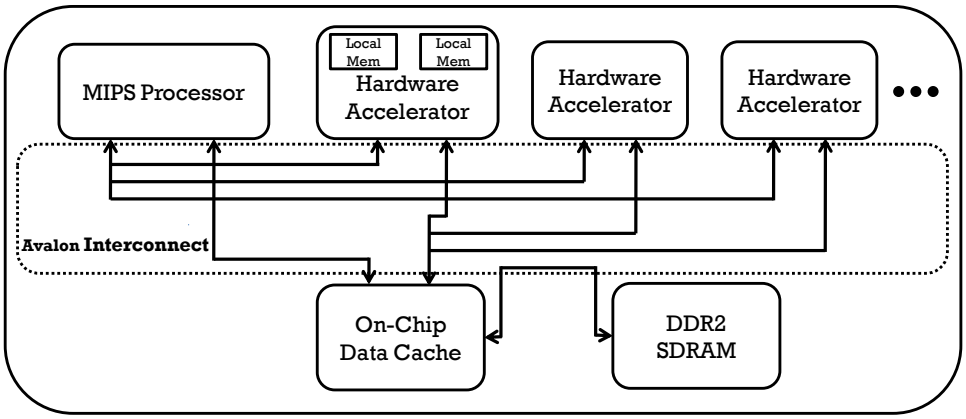


Figure 3.2: MIPS processor-accelerator hybrid system architecture.

cache can be replaced with a multi-ported cache (controlled by a `Tcl` parameter), which allows multiple accelerators to access the cache concurrently [17]. Note that, by default, LegUp generates a stall logic for each module, which allows an accelerator to be safely stalled on a cache miss. In addition, even when a module of an accelerator is stalled, other modules of the same accelerator can continue to execute (described in the next section).

3.4.6 Parallel Accelerator Architecture

In this work, we also allow *nested parallelism* – threads forking threads. Consider the case of there being multiple functions executed in parallel with Pthreads – a first level of parallelism. These functions could have one or more loops, some of which could be parallelized with OpenMP – a second level of parallelism. Currently, we only permit up to two levels of parallelism for automated hardware synthesis, with Pthreads being the first level and OpenMP being the second. OpenMP can also be used as the first level of parallelism, though we do not consider this case in the experimental study in this chapter. We refer to the second-level accelerators as *internal accelerators*. The internal accelerators are created inside their corresponding first-level accelerators. Internal accelerators can access the local variables of their first-level accelerators, which are created in local RAMs, as well as global/stack variables stored in the shared memory space.

Figure 3.3 shows the architecture of a hardware accelerator with internal accelerators. Internal accelerators, corresponding to OpenMP threads, are instantiated multiple times, each of which executes a portion of work in parallel. The internal accelerators are invoked simultaneously by an FSM, which also controls the execution of the first-level accelerator. As parallelization with OpenMP is blocking, after invoking the internal accelerators, the FSM waits until all of them have finished execution before continuing on to the next state (adhering to the semantics of the `GOMP_parallel_start/end`). Note that

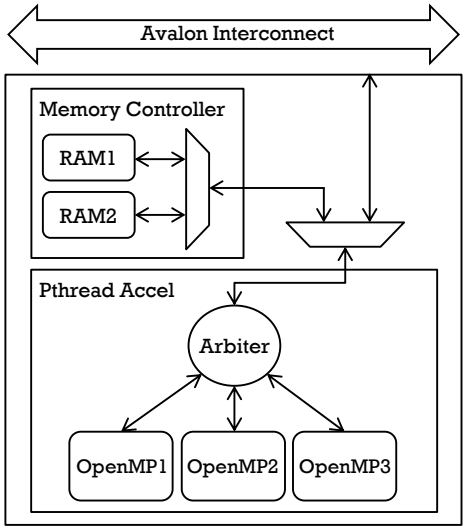


Figure 3.3: Nested accelerator architecture.

wrapper functions are not generated for internal accelerators, because they are invoked by the FSM (not the MIPS).

Since internal accelerators can access memory, and because they execute in parallel, a round-robin arbiter is created inside the first-level accelerator to arbitrate among internal accelerators as they access memory. When an internal accelerator makes a memory request, if it is not granted access by the arbiter, it stalls. Similarly, if an internal accelerator accesses shared memory and experiences a cache miss, it has to wait until the data is fetched from off-chip memory; however, other internal accelerators which are not accessing memory are free to execute. Hence, the internal accelerators do not execute in lockstep but work independently of one another.

3.5 Experimental Study

Using the hybrid flow, we study the performance and area of several different hardware configurations, each of which corresponds to a different parallelization scenario. The baseline configuration is the sequential MIPS processor/accelerator hybrid system, with no accelerators that operate in parallel – the processor and accelerators operate sequentially and the processor is stalled while an accelerator does its work. This sequential MIPS hybrid system generation was implemented as part of my M.A.Sc thesis work [18].

The parallel configurations fall into 3 classes: 1) a single-level of parallelization using Pthreads, 2) Pthreads combined with loop pipelining – a form of nested parallelism, and 3) Pthreads combined with OpenMP – nested parallelism as described in the previous section. For class #1, the Pthreads may

execute the same function or different functions, depending on the benchmark. When executing the same function, each accelerator performs a portion of the total work. Configurations in classes #2 and #3 can be used when the first-level threads contain loops – such loops can be parallelized by loop pipelining or OpenMP pragmas. Note that at the time this experiment was conducted, the loop pipelining capabilities of the LegUp HLS tool were limited and could only be applied to loops with bodies that contain no function calls or branches[3]. With the OpenMP support, however, loops with branches and function calls can be parallelized.

For class #3 (Pthreads combined with OpenMP), we experiment with various numbers of Pthreads and OpenMP internal accelerators, for a total of eight different configurations. The largest configuration is with 30 Pthreads with four OpenMP internal accelerators, which means that there are a total of 120 accelerators. Note that it does not necessarily mean that all 120 accelerators are *identical*, as the four OpenMP accelerators only parallelize the *loop* inside a Pthread function, and there can be other operations done outside the loop. We label architecture configurations as follows: `S` denotes the sequential baseline case, `4L1` denotes the 4 first-level Pthread accelerators architecture, `4L1-P` denotes the 4 first-level Pthread accelerators with loop pipelining, and `nL1-mL2` denotes the architecture with `n` first-level Pthread accelerators with `m` second-level OpenMP accelerators.

3.5.1 Benchmarks

We use a total of seven different benchmarks, each of which includes built-in inputs and golden outputs, with the computed result checked against the golden output at the end of the program to verify correctness. The inputs, golden outputs, and the computed results are held in global variables and stored in the shared memory space (off-chip DDR2 SDRAM). The benchmarks are:

- Black-Scholes: performs options pricing via a Monte Carlo approach. Computations are done in fixed-point.
- MCML: simulates light propagation from a point source in an infinite medium with isotropic scattering. The benchmark has been adopted from the Oregon Medical Laser Centre [60] with the computations done in fixed-point.
- Mandelbrot: an iterative mathematical benchmark which generates a fractal image.
- Line of Sight: uses the Bresenham's line algorithm [8] to determine whether each pixel in a 2-dimensional grid is visible from the source.

[3]Since then, we have implemented *if-conversion*, which can automatically remove branches by merging basic blocks, allowing loop pipelining to be applied to more complex loops.

- Division: performs integer division of two arrays.
- Hash: uses four different integer hashing algorithms to hash a set of numbers, and compares the number of collisions caused by the four different hashes.
- dfsin: adopted from the CHStone benchmark suite [37], it implements a double-precision floating-point sine function using 64-bit integers.

We synthesized each benchmark into each of the parallel architecture configurations and simulated the synthesized circuit using ModelSim to extract the total number of execution cycles. Following the cycle-accurate simulation, each benchmark was synthesized to the Altera Stratix IV (EP4SGX530KH40C2) with Quartus II (ver. 11.1SP2) to obtain area and critical path delay ($Fmax$) numbers. Execution time (wall-clock time) for each benchmark is computed as the product of execution cycles and post-routed clock period.

3.5.2 Results

The performance results for all benchmarks and all architectures are presented in Tables 3.2 and 3.3, and the area results are shown in Table 3.4. For those benchmarks where loop pipelining could not be used, or those which could not be parallelized with more accelerators (due to resource limits on the Stratix IV or due to the nature of the benchmark), the results are shown as "N/A".

Tables 3.2 and 3.3 show the number of execution cycles it takes to execute each benchmark, the $Fmax$, as well as the wall-clock time (in μs) based on the $Fmax$ and clock cycles results. The speedups and ratios of each parallel architecture relative to the sequential case are also shown. As expected, performance generally improves as the degree of parallelism is increased, up to a certain point. For most benchmarks, the total number of execution cycles decreases as more parallel accelerators are used. For Black-Scholes, MCML, and Mandelbrot, which are computationally intensive rather than memory intensive, the number of clock cycles scales well with the number of accelerators, especially up to the 4L1-4L2 architecture (16 accelerators). For 4L1-4L2, Black-Scholes, MCML, and Mandelbrot show 15.2×, 15×, and 13.6× speedup, respectively. For other benchmarks, which are more memory intensive, such as Line of Sight, Division, and Hash, the performance improvements are more modest. For the Line of Sight and Division benchmarks, a 4-ported cache [17] was used, which allows higher memory bandwidth, though it consumes more resources than the default cache, hence is not used for other benchmarks. For the Hash benchmark, locks are used to prevent race conditions between the internal accelerators. With more internal accelerators, the contention to access the mutex core increases, and thus the execution cycles actually increase as the number of internal accelerators is increased from three to four. Similarly,

for other benchmarks, as the total number of accelerators is excessively increased (up to 120 accelerators in Line of Sight), the reduction in clock cycles is either small, or the number of clock cycles increases. With too many accelerators, the work assigned to each accelerator becomes smaller, yet the memory contention increases, degrading performance.

Table 3.2: Performance results of all architectures for Black-Scholes, MCML, and Mandelbrot benchmarks.

	Black-Scholes			MCML			Mandelbrot		
	Time (Speedup)	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**
S	2058 (1)	437947 (1)	130.41 (1)	22317 (1)	1876187 (1)	84.07 (1)	18724 (1)	2523455 (1)	134.77 (1)
4L1	615 (3.9)	108654 (4.0)	127.39 (0.98)	6308 (3.5)	482855 (3.9)	76.55 (0.91)	4930 (3.8)	627059 (4.0)	127.18 (0.94)
4L1-P	N/A	N/A	N/A	N/A	N/A	N/A	4248 (4.4)	434533 (5.8)	102.28 (0.76)
4L1-2L2	488 (6.6)	56344 (7.8)	110.68 (0.85)	3613 (6.2)	246094 (7.6)	68.12 (0.81)	2849 (6.6)	355224 (7.1)	124.67 (0.93)
4L1-3L2	397 (6.9)	42410 (10.3)	87.3 (0.67)	3635 (6.1)	246094 (7.6)	67.71 (0.81)	1904 (9.8)	241805 (10.4)	127 (0.94)
4L1-4L2	364 (10.7)	28736 (15.2)	91.83 (0.7)	1961 (11.4)	125463 (15.0)	63.97 (0.76)	1455 (12.9)	185863 (13.6)	127.73 (0.95)
8L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	1131 (16.6)	126828 (19.9)	112.17 (0.83)
12L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	1089 (17.2)	113593 (22.2)	104.31 (0.77)

Table 3.3: Performance results of all architectures for Line of Sight, Division, Hash, and Dfsin benchmarks.

	Line of Sight			Division			Hash			Dfsin		
	Time (Speedup)	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**
S	4146 (1)	556933 (1)	134.34 (1)	2947 (1)	407837 (1)	138.37 (1)	2654 (1)	325859 (1)	122.76 (1)	2058 (1)	265513 (1)	129.02 (1)
4L1	1672 (2.5)	197625 (2.8)	118.22 (0.88)	867 (3.4)	112466 (3.6)	129.7 (0.94)	696 (3.8)	86378 (3.8)	124.16 (1.01)	615 (3.4)	67507 (3.9)	109.78 (0.85)
4L1-P	N/A	N/A	N/A	235 (12.5)	29457 (13.9)	125.13 (0.9)	635 (4.2)	83748 (3.9)	131.8 (1.07)	N/A	N/A	N/A
4L1-2L2	1045 (4.0)	127471 (4.4)	121.94 (0.91)	560 (5.3)	67237 (6.1)	120.11 (0.87)	540 (4.9)	64588 (5.1)	119.67 (0.97)	488 (4.2)	47533 (5.6)	97.5 (0.76)
4L1-3L2	882 (4.7)	102313 (5.4)	115.98 (0.86)	473 (6.2)	51349 (7.9)	108.59 (0.78)	449 (5.9)	50917 (6.4)	113.3 (0.92)	397 (5.2)	34520 (7.7)	86.89 (0.67)
4L1-4L2	851 (4.9)	98708 (5.6)	115.97 (0.86)	400 (7.4)	46854 (8.7)	117.12 (0.85)	594 (4.5)	66219 (4.9)	111.56 (0.91)	364 (5.7)	27463 (9.7)	75.55 (0.59)
8L1-4L2	535 (7.6)	59361 (9.4)	111.05 (0.83)	276 (10.7)	27627 (14.8)	100.08 (0.72)	N/A	N/A	N/A	N/A	N/A	N/A
12L1-4L2	407 (10.2)	43540 (12.8)	106.87 (0.80)	239 (12.3)	23133 (17.6)	96.64 (0.7)	N/A	N/A	N/A	N/A	N/A	N/A
16L1-4L2	482 (8.6)	49664 (11.2)	102.94 (0.77)	458 (6.4)	42663 (9.6)	93.09 (0.67)	N/A	N/A	N/A	N/A	N/A	N/A
20L1-4L2	482 (8.6)	44036 (12.7)	91.37 (0.68)	697 (4.2)	57042 (7.2)	81.87 (0.59)	N/A	N/A	N/A	N/A	N/A	N/A
30L1-4L2	499 (8.3)	42558 (13.1)	85.34 (0.64)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 3.4: Area results of all architectures for all benchmarks.

	Black-Scholes			MCML			Mandelbrot			Line of Sight			Division			Hash			Dfsin		
	Logic Util	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**
S	43474	140	88	38810	136	120	22111	134	28	23550	134	12	24527	134	8	31015	262	36	51763	136	52
4L1	111156	142	328	87874	150	456	26869	134	88	40385	134	24	43134	134	8	31210	262	36	138380	142	184
4L1-P	N/A	N/A	N/A	N/A	N/A	N/A	27208	138	56	N/A	N/A	N/A	43680	138	8	32414	278	36	N/A	N/A	N/A
4L1-2L2	198126	174	648	152660	166	904	34187	158	168	54926	158	40	79208	158	8	49550	326	64	276756	158	360
4L1-3L2	276243	174	968	264559	166	1020	37476	158	248	62988	158	56	98317	158	40	58328	326	92	387958	375	568
4L1-4L2	361327	443	1020	355873	419	1020	40890	158	328	70834	158	72	119359	158	72	69031	326	120	408169	451	776
8L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	61251	182	648	108737	182	136	204230	182	136	N/A	N/A	N/A	N/A	N/A	N/A
12L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	81224	206	968	146380	206	200	275185	635	200	N/A	N/A	N/A	N/A	N/A	N/A
16L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	181166	230	264	340888	1280	264	N/A	N/A	N/A	N/A	N/A	N/A
20L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	214658	254	328	400613	1280	328	N/A	N/A	N/A	N/A	N/A	N/A
30L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	310013	324	488	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Logic utilization is in terms of the number of half-ALMs, and DSPs are in terms of the number of used DSP blocks. There are a total of 424,960 half-ALMs, 1,280 M9Ks, and 1,024 DSP blocks on the device.

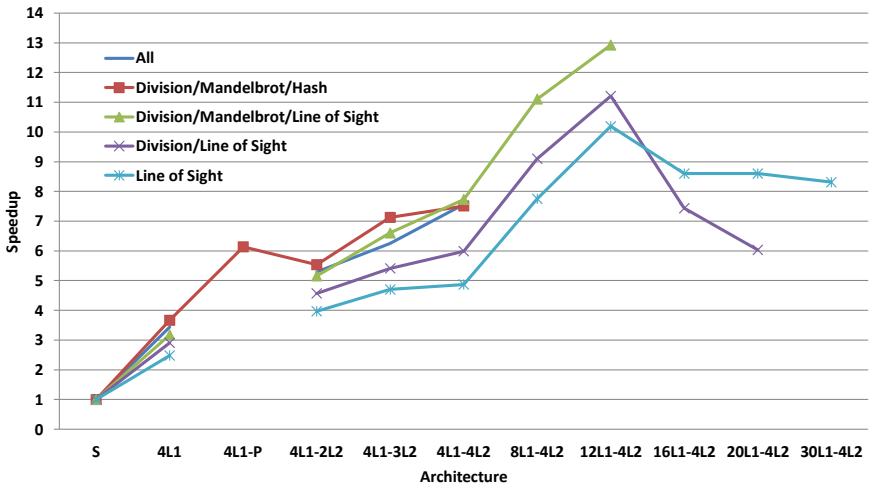


Figure 3.4: Geomean speedup ratios.

The $Fmax$ of the systems is also affected as the degree of parallelism is varied. Overall, $Fmax$ is negatively impacted with more accelerators, mainly due to the arbitration and the stall logic, needed to manage memory contention. Some benchmarks, such as dfsin, Black-Scholes, and MCML, exhibit more rapid reductions in $Fmax$ than others. We believe that, as the utilization of the Stratix IV becomes close to full, the Quartus II synthesis tool has more difficulty optimizing the implementation. For example, for the 4L1-4L2 architecture, dfsin showed 96% logic utilization, and Black-Scholes and MCML showed 85% logic utilization and used 100% of the Stratix IV DSP blocks. However, for the other benchmarks, the $Fmax$ reduction for 4L1-4L2 is ∼10% when compared to the baseline.

Figure 3.4 shows the geometric mean speedup (in wall-clock time) of the different architectures normalized to the baseline case. Since not all parallelization configurations could be used for all benchmarks, multiple lines are plotted, with each line showing the geometric mean speedup for a subset of circuits in which the particular configuration could be used[4]. The legend shows which benchmarks are included for each line on the graph. The geometric mean across *all* benchmarks (first line of the legend) shows that the best speedup of 7.6× is observed with the 4L1-4L2 architecture. The 4L1-P configuration is not included in this case, since loop pipelining could not be applied in all benchmarks.

For the benchmarks where loop pipelining could be used (Division/Mandelbrot/Hash), 4L1-P shows 6.17× speedup over baseline, and 4L1-4L2 still shows the best result with 7.51× speedup. There are cases where loop pipelining can perform far better, however. For instance, for the Division benchmark, Table 3.3 shows that 4L1-P outperforms all other architectures, even the 20L1-4L2 architecture which has 80 accelerators. This is because a 32-bit division takes 32 cycles in LegUp, using Altera's divider core pipelined to achieve the highest-possible $Fmax$. Since the divider itself is pipelined, it can accept a new

[4]If we had used a single line, each data point would represent the average for potentially different sets of circuits.

input every clock cycle, which is very well suited to loop pipelining. With 4 Pthread accelerators, each of which has only one hardware instance of the loop body, this 4L1-P architecture shows 12.5× speedup over the baseline architecture for the Division benchmark. The biggest speedup in Figure 3.4 is 12.9× with the 12L1-4L2 architecture for three benchmarks. Mandelbrot shows the largest single benchmark speedup with 17.2× with the 12L1-4L2 architecture, and 16.6× with the 8L1-4L2 architecture. Overall, as the number of accelerators is increased excessively, the geometric mean speedups decrease due to reductions in $Fmax$ and diminishing returns in clock cycle reduction.

Table 3.4 shows the area results in terms of Stratix IV logic utilization, M9K blocks, and DSP blocks. The logic utilization metric reported by Quartus II is an estimate of how full the device is, calculated from the number of half-ALMs (adaptive logic modules) used in the design. M9Ks are Altera's on-chip RAMs which can hold up to 9 Kbits of data including parity bits [3]. M144Ks, which are much larger RAMs that can hold up to 144 Kbits, are only used by one benchmark, Division, and hence are not shown on the table for space reasons. Note that usage of M144K blocks is taken into consideration when calculating the total area of the systems (see below). The area results presented in Table 3.4 are for the entire system, which includes the MIPS processor, the on-chip cache, the DDR2 controller, the interconnection network, as well as the hardware accelerators. In general, as expected, the area increases as parallelism is increased, both in terms of Pthreads and OpenMP accelerators. Mathematically intensive benchmarks, such as Black-Scholes, MCML, Mandelbrot, and dfsin, show significant increases in the number DSP blocks. In fact, even though the logic utilization for Mandelbrot is only ∼19%, it could not be parallelized more than 12L1-4L2, since DSP usage was at 95%. Note that, parallelization with Pthreads does not necessarily increase circuit area if different functions are executed in parallel, compared to when the same functions are executed sequentially. The accelerators consume the same amount area in both the sequential and parallel cases. Thus, for the Hash benchmark, the circuit area is roughly the same for the S and 4L1 architecture configurations.

Area-delay product is another important metric when evaluating the efficiency of different hardware architectures. Calculating the total circuit area of an FPGA can be particularly challenging since modern FPGAs consist of different types of blocks, such as logic blocks, memory blocks, DSP blocks, and routing, each of which consumes different amount of chip area. To account for this fact, we use the data from [149], which gives the chip tile area for each type of block[5]. Using this data and the results in Table 3.4, we calculate the total circuit area for each architecture configuration for each benchmark. With this area, and using the wall-clock time results from Tables 3.2 and 3.3, we computed the geometric mean

[5]Note that although [149] provides detailed area data for the types of tiles in Stratix III, Stratix IV contains the same types of tiles, so we believe the data can be used for this relative area comparison.

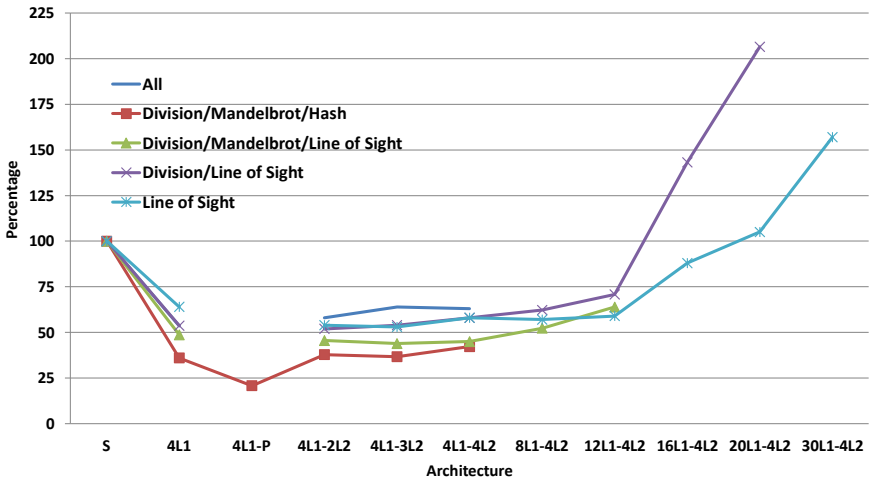


Figure 3.5: Geomean area-delay ratios.

area-delay product, shown in Figure 3.5 as a percentage compared to the baseline architecture. Similar to Figure 3.4, multiple lines are shown, each of which includes results for different architectures/benchmarks. Looking at the geomean result for *all* benchmarks (first line of the legend), the 4L1 architecture shows the best result with 53.5% area-delay product (1.87× improvement) of the baseline case, and the 4L1-2L2 and the 4L1-4L2 architectures follow with 58% (1.72× improvement) and 63% (1.59× improvement), respectively. For the three benchmarks where loop pipelining could be used, 4L1-P shows the best result with 20.8% (4.81× improvement) and 4L1 follows with 36.7% (2.72× improvement). For the Division benchmark, 4L1-P showed 12% area-delay product (8.33× improvement) when compared to the sequential case. Similar to the speedup results, as the degree of parallelism is excessively increased, the area-delay product is also significantly increased, with the 20L1-4L2 architecture showing 206% vs. the baseline for the Division/Line of Sight plot. Overall, for cases where loop pipelining can be used, Pthreads combined with loop pipelining can be the most beneficial in terms of the area-delay product.

In summary, the results above demonstrate the capabilities of our HLS tool to synthesize circuits automatically into a variety of architectures with dramatically different degrees of parallelization. Promising results are observed in terms of execution time and area-delay product. It is worthwhile to reiterate that with our approach, changing the parallelization configuration is straightforward, with Pthreads requiring only a few lines of code changes, and OpenMP requiring a *single* number (`num_threads` clause) to be changed. This enables wide design space exploration with ease, which is certainly not feasible with manually designed hardware. With our framework, exploring with different parallelization schemes in hardware is no more difficult than the analogous exploration in software.

3.6 Summary

In this chapter, we presented a framework which can automatically compile software threads to parallel hardware, targeting the MIPS processor-accelerator hybrid system. Two standard software parallelization techniques, Pthreads and OpenMP, are used to generate hardware accelerators which execute concurrently in a shared memory system. OpenMP allows a section of code, such as a loop, to be automatically compiled to parallel accelerators, whereas Pthreads allow the same and/or different functions to be synthesized to concurrently operating hardware accelerators. Our framework allows nested parallelism, where Pthreads can invoke OpenMP threads to allow 2 levels of parallel accelerators. Loop pipelining can also be used in conjunction with Pthreads. We also permit *sharing* a Pthread accelerator between multiple threads, synonymous to sharing a CPU core across threads. This is a practice that is often done in the software domain and thus could be *expected* by software engineers. A key advantage of this work is that software engineers without hardware knowledge can use standard software APIs to obtain significant speedup through parallel hardware systems.

The experimental study showed that, in geometric mean results over 7 benchmarks, using 4 Pthread accelerators, each of which contains 4 OpenMP accelerators, provides the best performance results, with 7.6× speedup and 63% area-delay product when compared to the single-threaded systems. The highest speedup was 17.2× in wall-clock time with the 12L1-4L2 architecture, and the best area-delay product was 12% (over 8× improvement) with the 4L1-P architecture.

This chapter includes work which was published in the 2013 IEEE International Conference on Field-Programmable Technology (FPT) [14], as well as improvements which have been made after the publication.

Chapter 4

ARM Hard Processor System and Direct Memory Access (DMA) Support

4.1 Introduction

FPGAs have traditionally been difficult to use, and despite their apparent performance and power benefits, their poor usability has deterred many from using the platform. One way to improve usability is by providing a processor system on the FPGA, which can be controlled via software. This gives access to FPGAs to those with only software skills, and for this reason, there have been numerous implementations of *soft* processors, from both industry and academia [135, 105, 151, 41, 152]. Although there are many different kinds of soft processors, with varying performance, area, and supported features, soft processors are typically big and slow, making them difficult to be used in creating a high-performance FPGA system that can, for example, outperform an `x86` CPU. Recognizing this, FPGA vendors have started to produce SoC FPGAs that have a *hardened* dual-core, or even a quad-core ARM processor, that can run at well over 1 GHz [106, 140]. These SoC chips integrate the hard ARM processor on the same die as the FPGA, allowing one to create a tightly-coupled high-performance system that takes advantage of the broad applicability of a high-speed multi-core ARM processor, as well as the performance and power-efficiency provided by the FPGA fabric.

To this end, we describe in this chapter our HLS support for using the ARM HPS (Hard Processor

System) on the Altera Arria V SoC FPGA to create a ARM hybrid SoC, where Pthread functions are accelerated in hardware, and the remaining software segments are executed in software on the ARM processor. Bringing up the ARM processor, especially in bare metal mode, is an onerous and troublesome task for both software and hardware engineers. With this work, we provide an SoC system that has been completely set up, so that the ARM processor can simply be programmed and used through makefile targets. In addition to the bare metal system, the user can also choose to run the ARM processor with a Linux OS, which is also provided with our framework.

A key architectural improvement discussed in this chapter is concerning the memory architecture comprising a processor-accelerator hybrid system. In Chapter 3, the hybrid system was limited to a fixed memory architecture of an on-chip cache backed by off-chip DDR memory. Although this architecture provides cache coherent data to hardware accelerators, its memory bandwidth can be quite limited, as only a single element of data can be accessed at a time through the cache. On a cache hit, the data is returned with a low latency, but a cache miss causes the accelerator to be stalled for many cycles, until the data is fetched from off-chip memory and into the cache, then returned to the accelerator. Repeated cache misses can lead to significant performance degradations. Hence, this type of memory architecture is typically unsuitable for pipelined hardware, where new input data can be required *every* clock cycle. To mitigate this potential problem, we provide direct memory access (DMA) support, which allows a large amount of data to be fetched in *bursts* from off-chip DDR3 memory to on-chip buffers (and vice versa). This allows pipelined accelerators to continuously run with much less frequent memory stalls. The DMA support is provided for both the ARM and MIPS hybrid systems. As an experimental study, we investigate the performance, area, and energy-efficiency of several ARM and MIPS hybrid systems. We also evaluate the speed and energy-efficiency one of the benchmarks, Black-Scholes, in three different scenarios: 1) when it is executed on an ARM hybrid system, 2) when it is executed purely in software on the ARM processor, and 3) when it is executed purely in software on two different `x86` processors, the Intel i7-4770K CPU and the Intel Xeon E5-1650 CPU. Note that the Intel i7-4770K is fabricated on a 22 nm process, and the Xeon E5-1650 is fabricated on a 32 nm process, where as the Arria V SoC FPGA is fabricated on a 28 nm process.

4.2 Background

Prior works which relate to compiling threads onto an FPGA, were discussed in Section 3.2. In this section, we only focus on those which relate to providing HLS support for an ARM SoC FPGA. Perhaps the tool which bears the most similarity to our own, is Xilinx's SDSoC (Software-Defined System On

Chip) [137]. Just recently released in 2015, the SDSoC is used for implementing heterogeneous embedded systems using the Zynq-7000 All Programmable SoC platform [141], which features a dual-core ARM Cortex-A9 MPCore (the same kind is used on the Altera Arria V SoC FPGA). Designed for software engineers, the SDSoC provides automated software acceleration and automated system connectivity generation for the Zynq FPGA. To use the SDSoC, one first writes an application in C/C++, then specifies a subset of the functions within the application to be compiled to hardware. The SDSoC system compiler then compiles the application into hardware and software, to realize the complete embedded system on a Zynq device [145].

In many ways, the SDSoC is similar to LegUp HLS, in that both tools allow the user to select a portion of the code to be compiled to hardware, with the tool automatically handling software/hardware partitioning and system generation. It is worth noting that the automated hybrid system generation flow in LegUp was first established in 2011 [10]. Our support for targeting an ARM processor on an SoC FPGA (Altera Cyclone V SoC FPGA) was first established by Blair Fort and Bain Syrowik in 2014 [32]. In any case, we believe that a *key* differentiator in LegUp HLS is the support for compiling *standard* parallel software to parallel hardware accelerators. Conversely, the SDSoC uses a vendor-specific pragma to create multiple hardware instances. Consider the following code snippet from the SDSoC user guide [144].

```
#pragma SDS async(1)
mmult(A, B, C); // instance 1

#pragma SDS async(2)
mmult(D, E, F); // instance 2

#pragma SDS wait(1)
#pragma SDS wait(2)
```

The `SDS async(id)` pragma directs the tool to create a hardware instance that is referenced by the `id`. It also makes the function call return, without waiting for the hardware function to finish execution, similar to the behaviour of `pthread_create`. The `SDS wait(id)` pragma makes the processor wait until the hardware function specified via the `id` has finished execution, similar to the behaviour of `pthread_join`. In this example, SDSoC creates two hardware instances of of `mmult`, which are called to execute in parallel, and synchronized afterwards with the `SDS wait` pragma.

Even though the use of these pragmas achieves creating parallel hardware, it is nevertheless a non-

standard solution. It is *vendor-specific*, and *platform-specific*. Compiling the code above with a standard software compiler, such as `gcc`, simply creates two sequential calls to the `mmult` function, as the pragmas will be ignored. This introduces behaviour that is *only* present in hardware, which makes debugging more difficult. A typical HLS design flow is to first verify everything in software, then compile the software to hardware. Making hardware behaviour different from software can introduce hardware bugs that are *invisible* from software. These bugs are then only *debuggable* in hardware through RTL simulations or via on-chip debugging, which are much more difficult and time-consuming than software debugging. Another issue with using vendor-specific pragmas is that the pragmas are foreign to anyone who is not familiar with the tool. The behaviours of the pragmas can be confusing, especially if they are not verifiable in software. Our work does not suffer from such issues, as we use standard software parallel programming techniques that are widely known and used by many software engineers, to create parallel hardware, where both software and hardware execute in parallel. Behaviours that only arise from parallel execution, such as synchronizing between threads, can be tested in software first, before compiling to hardware.

Altera's OpenCL SDK can also be used target an ARM SoC FPGA, such as the Cyclone V SoC [95]. The programming model, however, is different for OpenCL, where there is a clear division between the *host* code, to be executed on the ARM processor, and the *kernel* code, to be compiled to hardware. The user explicitly needs to make the separation, with the host code written in C/C++, and the kernel code written in OpenCL, hence there is no automatic software/hardware partitioning involved. The two parts also need to be compiled separately by the user, where the host code is compiled with a `gcc` cross-compiler targeting the ARM, and the kernel is compiled with `aocx`, Altera's HLS compiler. In OpenCL, many things, such as allocating buffers, or transferring data between host and kernel, need to be handled explicitly by the user through the use of OpenCL host APIs (some of which have many different possible configurations). OpenCL is typically used for massively parallel applications, traditionally on GPUs and now on some FPGAs. We believe that our methodology of using C with Pthreads and OpenMP is a more intuitive approach for users targeting the embedded space.

4.3 Pthreads to ARM Processor-Accelerator Hybrid System

The generation flow for an ARM processor-accelerator hybrid system remains largely the same as what was described for generating a MIPS hybrid system in Section 3.4. The choice between using the *soft* MIPS processor, or the *hard* ARM processor, can be made via a `Tcl` configuration parameter. When LegUp is invoked, the `ParallelAPI` pass transforms the Pthread library functions and creates

the thread managing logic, and the `SW Partitioning` pass partitions the program and generates the necessary wrapper functions. The partitioned software is compiled to a binary using the ARM compiler toolchain, then the Qsys System Integrator automatically generates the complete SoC with the ARM HPS.

4.3.1 ARM Hybrid System Architecture

The system architecture of an ARM hybrid system, which consists of the ARM HPS (Hard Processor System), hardware accelerators, and off-chip memory, is shown in Figure 4.1. The ARM HPS on the Arria V SoC Development Board consists of a dual-core ARM Cortex-A9 MPCore, running at 1.05 GHz [107]. In addition to the ARM CPUs, the HPS includes 32KB L1 on-chip instruction and data caches, a 512KB L2 on-chip cache, and an SDRAM controller which connects to an off-die (on-board) 1GB DDR3 memory. A number of different interfaces are provided to communicate between the HPS and the FPGA, including the HPS-to-FPGA interface (denoted as H2F in the figure), and the FPGA-to-HPS interface (denoted as F2H in the figure). The H2F is an AXI (Advance eXtensible Interface [109]) interface that allows the ARM processor to communicate with the hardware accelerators in a hybrid system. To access the shared memory space, hardware accelerators access the F2H interface (also AXI), which is connected to the Accelerator Coherency Port (ACP), which connects to on-chip caches in the HPS to provide cache coherent data. The HPS also provides the FPGA-to-HPS SDRAM interface (discussed in Section 4.4), allowing an FPGA component to directly access the HPS DDR3 memory without going through the cache.

Note that the Arria V SoC Development Board has two different types of DDR3 memories: 1) the HPS-side DDR3 memory, used by the ARM processor and can also be accessed from the FPGA via the FPGA-to-HPS SDRAM interface, and 2) the FPGA-side DDR3 memory, which is used by components implemented on the FPGA fabric only. The FPGA-side DDR3 memory is used if one choose to use the MIPS hybrid system on the Arria V SoC.

The operation of the ARM hybrid system remains similar to that of MIPS, where the processor invokes hardware accelerators via memory-mapped operations over the Avalon Interconnect, and the accelerators access the shared memory space over the interconnect. However, since the on-chip cache resides within the HPS and has to be accessed through layers of bridges/switches, in addition to crossing clock domains (HPS runs much faster than the FPGA fabric), the memory latency is significantly higher than accessing the on-chip cache in the MIPS system, where the cache is implemented on the FPGA fabric. An accelerator can also have local memories within the accelerator, which can be accessed without

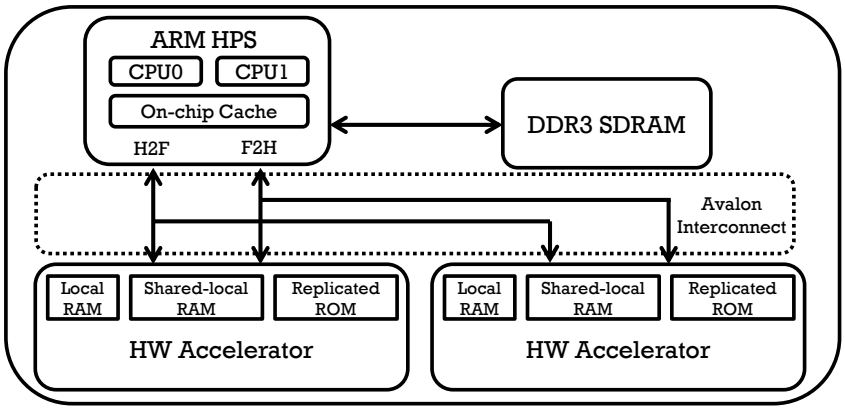


Figure 4.1: ARM processor-accelerator architecture.

going over the interconnect.

Improved Accelerator Memory Architecture

Since the work in Chapter 3, we have significantly improved the accelerator memory architecture by adding support for a *points-to analysis*, a static code analysis technique that establishes which variable/array a pointer can point to. With the points-to analysis we can *localize* more memories within accelerators, and access more of these memories in parallel. In the prior work, we used a simple memory partitioning algorithm, where only *local* arrays to the accelerated function (i.e., *auto* variables which would be stored on the stack in software), were implemented as local memories within the accelerator. Any other memories such as global variables/arrays, or local arrays of caller functions, were stored in the shared memory space. Local memories within the accelerator also needed to be instantatiated inside a single *global* memory controller, which provided steering logic to guide memory accesses to the correct RAM at *runtime*. However, this limited memory accesses to *two* per clock cycle, since the specific RAM to be accessed could only be determined at runtime (i.e., only a single RAM could be accessed at a time), with each RAM being dual-ported [11]. The global memory controller is described in more detail in Section 6.5.2. In our work in [15], which is described in Chapter 6, we implemented support in LegUp for Andersen's points-to analysis [5], which is one of the most well-known and accurate pointer analysis techniques. The points-to analysis algorithm was implemented in LLVM by Jia Chen at the University of Texas at Austin [89].

Using the points-to analysis, we can determine at *compile* time, which memory locations a pointer can reference, and intelligently designate arrays for implementation in *global*, *local*, or *shared-local* memories. An array is designated into a local or a shared-local memory if it is never referenced by a pointer that points to multiple arrays. If such an array is only accessed by a single function, it is designated as

local memory. Otherwise, if it is referenced in multiple functions, it becomes a *shared-local* memory. A local/shared-local memory is stored within an accelerator if it is *only* accessed by functions within the accelerator. Otherwise, it is stored in the shared memory space (off-chip DDR3 memory). Each local and shared-local memory stored within an accelerator is accessed through a dedicated set of memory ports, allowing concurrent memory accesses among such local and shared-local memories. We automatically create arbitration logic to handle memory contention for shared-local memories. An array is designated as *global* if it is referenced by a pointer which can also point to another array. Again, if the array is only referenced by pointers within the accelerator, we instantiate the memory inside the accelerator. In this case, however, pointer aliasing must be resolved at runtime, hence we instantiate the memory inside the global memory controller of the accelerator. It is worth noting that, for programs containing no dynamic memory allocation, we found that *most* pointer references can be resolved at compile time, hence most memories are not classified as global memories.

With the points-to analysis, we can also reason intelligently about variables/arrays which are not *declared* as a local to the accelerated function in software. For example, if a global array is only accessed by an accelerated function, it is implemented within the accelerator. Any arrays which are accessed by both the processor and the accelerator, or accessed by multiple accelerators, are stored in the shared memory space. However, constant memories, which are shared between threads, for example, can also be replicated and localized to each hardware accelerator (replicated ROM). Within an accelerator, all local and *independent* shared-local RAMs, as well as replicated ROMs, can be accessed in parallel. This improved memory architecture is also implemented for MIPS hybrid systems, as shown in Figure 4.2[1]. The memory partitioning and the generation of the memory architecture is completely automatic, without any specifications required from the user. The work on this improved memory architecture is described in more detail in Chapter 6.

4.3.2 Operating System Support

Traditionally, getting an operating system to run on an FPGA system has been an arduous task. *Soft* processors are generally slow, requiring a significant engineering effort for an OS to run on them. Even when an OS successfully runs on a soft processor, its speed is typically quite slow, making it potentially impractical for deployment in a real system. With the introduction of hard ARM processors on FPGAs, the situation has changed. Multi-core ARM CPUs which can run at over 1 GHz can reliably boot an OS, while providing sufficient performance for embedded tasks. A user can develop and compile code *on*

[1]Note that to improve system modularity, we have *pulled* out the instruction cache from inside the MIPS processor to instantiate it as its own Qsys component.

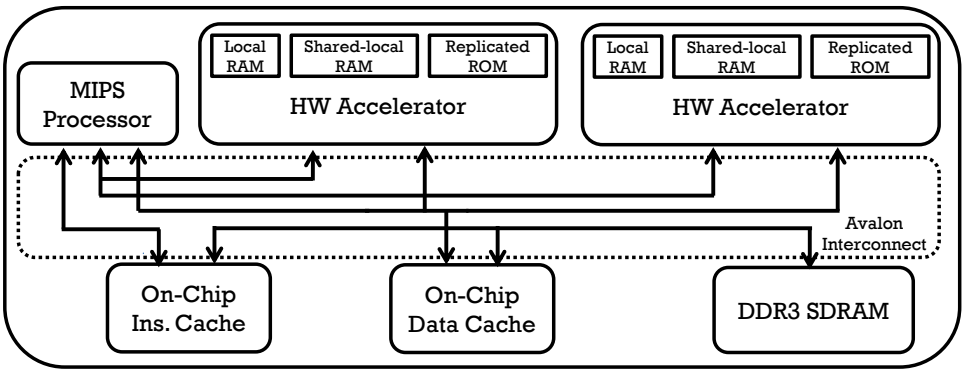


Figure 4.2: MIPS processor-accelerator architecture.

the processor itself, and even execute multi-threaded code, with the OS managing threads as necessary. Having an ARM SoC FPGA with an OS opens many doors for FPGAs, where software engineers can develop, compile, and execute code in an environment they are already familiar with, while having the option of accelerating a critical portion of their code on the FPGA with an HLS tool. The OS also comes pre-installed with all standard compilers and libraries.

We have set up the Linux OS flow for the Arria V SoC, where the hard ARM processor can boot the OS from an SD card. RocketBoards.org [91], a website which provides documentation and references for Altera SoC boards, gives instructions on how to boot the OS on the Arria V SoC, including generating the preloader for the ARM HPS, creating an SD card image of the OS, setting up board jumpers, and configuring a serial connection to allow communication between the ARM and the host PC. Although the website also provides a pre-built SD card image of the OS, we have compiled the OS from source, which allowed us to make additional configurations. We have verified this compiled OS on the Arria V SoC, and its SD card image is provided on our website [74]. This work was done by Ruo Long Lian.

Using the Linux OS, the ARM HPS can be used to execute software which works in tandem with hardware accelerators on the FPGA fabric. Note however that since the OS uses *virtual* addresses, we need to map the *physical* addresses of the hardware accelerators (memory-mapped addresses in the the Qsys system), to virtual addresses used in the OS. To do this, we use the `mmap` Linux system call [79], which creates a mapping in the virtual address space. This maps the physical addresses of hardware accelerators to virtual addresses seen by the software running on the OS, thus reading/writing to the mapped virtual addresses is tantamount to communcation with hardware accelerators. `Mmap` can also be used to map a contiguous non-cacheable region of physical memory, which is needed for performing DMA transfers between accelerators and off-chip memory [96]. An example code for using the `mmap` function to map a hardware accelerator is shown in Appendix A.

Note that although the generation of an ARM hybrid system is automatically done by LegUp, the

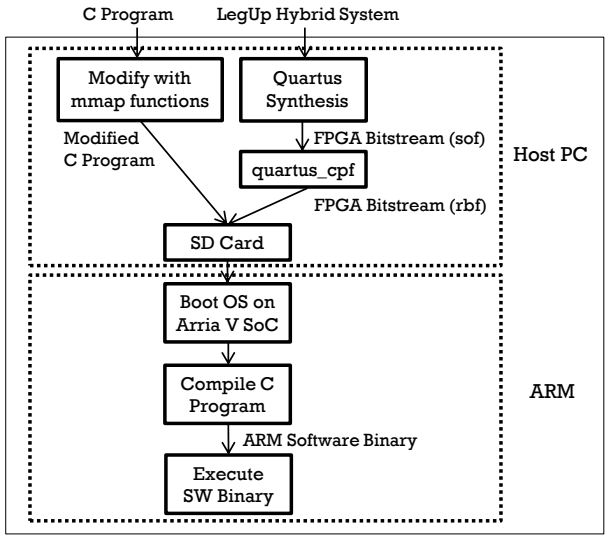


Figure 4.3: ARM hybrid flow with OS.

insertion of the `mmap` function calls is not yet automated. The user must edit the software to call the `mmap` functions to map the hardware accelerators, and read/write to the relevant pointers to communicate with the accelerators. Figure 4.3 shows the steps required to use an ARM hybrid system with an OS. First the ARM processor-accelerator system is generated automatically with LegUp, and compiled with Altera's synthesis tool, Quartus II, to generate the FPGA programming bitstream. This bitstream, in `sof` (SRAM Object File) format, needs to be converted to an `rbf` (Raw Binary File) format with `quartus_cpf`, Altera's conversion tool. Then, the user needs to modify the original C code to map hardware accelerators with `mmap` functions. Both the modified C code, and the `rbf` file are copied to the SD card with the OS. All of these steps are performed on the host PC. Now, the OS can be booted on the Arria V SoC from the SD card, and a serial connection program such as `picocom` [76] can be used to establish the connection between the ARM and host PC. Through this connection, the user can compile the C code with `gcc` on the ARM processor, then execute the binary, which invokes the hardware accelerators.

4.3.3 Bare Metal Support

Although having an operating system provides many benefits, as the OS handles various complicated tasks behind the scenes for a system to *just* work, it can also add overheads. If the role of the processor is to simply set up components, configure and start up hardware accelerators, without requiring any libraries or complex tasks such as multi-threading, it can also be beneficial to run the processor in bare metal (no OS). To this end, we also provide bare metal support for the ARM HPS on the Arria V SoC

FPGA.

The biggest challenge to using a bare metal system is to bring up the system in the first place. Everything from writing ARM assembly code to set up the memory management unit (MMU), L1/L2 caches, FPGA-to-SDRAM bridge, to writing a linker script that brings all of these together to work with the software program, must be handled by the user. Essentially, in bare metal, much of what would be handled automatically by an OS, must be set up manually by the user. This can require considerable engineering effort, and is a non-trivial task for both software and hardware engineers. Recognizing this, we provide with LegUp HLS a bare metal system where the ARM HPS is completely set up, so that the user can simply use it via makefile targets to run applications on the ARM. A significant portion of this work was done by Bain Syrowik for the ARM HPS on the Cyclone V SoC, where I had to make additional modifications for the Arria V SoC.

One of the major modifications that was required was for the preloader. A typical way of running software on an ARM HPS is to use the ARM Development Studio 5 Altera Edition (DS-5 AE) [101], provided as part of Altera's SoC Embedded Design Suite (EDS) [102]. The DS-5 provides an Eclipse-based IDE, which allows the user to design software, debug and compile it, and download the software binary onto the HPS off-chip memory to execute it on the ARM processor. However, the DS-5 cannot be used as part of LegUp's automatic hybrid generation flow. In addition, a *paid* license is required to use the DS-5, further limiting its use even to those who simply just want to use the ARM HPS. To remove the dependency on the DS-5, we had to implement our own functionality to boot up the ARM, which required editing the preloader code. In one of many `C` files generated by the Embedded Design Suite, we had to insert inline ARM assembly to the preloader code, to first switch the processor from Thumb mode (which runs 16-bit Thumb instructions, used to reduce memory footprint), to ARM mode (which runs 32-bit ARM instructions), then stop the boot sequence to run our start up code, which sets up the ARM HPS, then executes the user application[2]. The step-by-step instructions are given in Appendix B. We believe that this will be of tremendous help to anyone who wishes to simply download and execute a software binary on the ARM HPS in the bare metal flow, without using the additional features provided by the DS-5.

Figure 4.4 shows the flow for using a bare metal ARM hybrid system. Contrary to the ARM OS flow, which currently requires the user to make minor changes to the C code to map hardware accelerators in virtual memory, the bare metal flow is completely automated. The complete SoC can be compiled, downloaded and executed on an FPGA with *three* makefile targets. The generated SoC can be compiled

[2]This method of editing the preloader code is used for the Altera Monitor Program for the DE1-SoC board with the Cyclone V SoC FPGA (which also has the ARM HPS). We would like to thank Kevin Nam at the Altera University Program for helping us with this change.

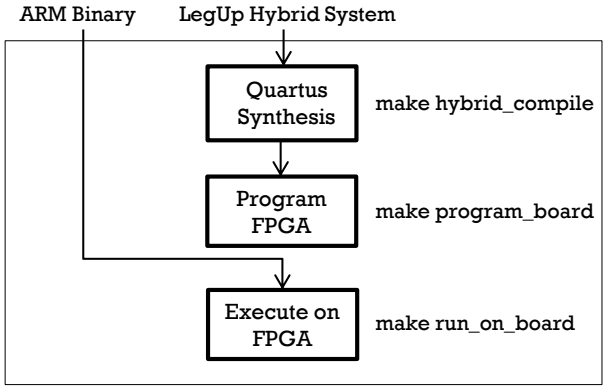


Figure 4.4: ARM hybrid flow with bare metal.

to a programming bitstream with `make hybrid_compile`, and downloaded onto the FPGA with `make program_board`. Finally, the software binary, which has been automatically modified to call the hardware accelerators, can be downloaded onto the HPS DDR3 memory and executed on the ARM processor (which invokes the hardware accelerators) with `make run_on_board`. The ARM cross-compiler, which creates the ARM binary, and `quartus_hps`, which programs the HPS, are all set up to be used for the Arria V SoC and executed *under the hood* via the Makefile targets. An SD card is not needed for bare metal, as the software binary is stored in the HPS-side DDR3 memory. With this automated bare metal flow, one can go from C code to an ARM hybrid system running on an SoC FPGA with only a handful of commands.

4.4 Direct Memory Access (DMA) Support

In this section, we describe providing DMA support for our processor-accelerator hybrid systems, which can significantly increase memory bandwidth and improve performance. First, a software library is needed to control DMA operations from the processor. We have created an intuitive, easy-to-use software library that can be used to start/reset a DMA operation, as well as make the processor wait until a DMA transfer is done. Their function prototypes are shown in the code below.

```
// The main DMA function which configures a DMA core,
// and starts a DMA transaction.
void startDMA(
  volatile unsigned long *DMA_ADDRESS,
  volatile unsigned long *srcAddr,
```

```
  volatile unsigned long *destAddr,
  WORD_SIZE wordSize,
  const unsigned bytes_to_transfer,
  ADDRESS_INCREMENT_SCHEME addrIncrementing
);

// Wait until the DMA core is done with its transfer.
void pollDMA(volatile unsigned long *DMA_ADDRESS);

// Reset a DMA core.
void resetDMA(volatile unsigned long *DMA_ADDRESS);

// A wrapper function used to make a DMA transfer to a FIFO.
void startDMAtoFIFO(
  volatile unsigned long *DMA_ADDRESS,
  volatile unsigned long *srcAddr,
  FIFO *fifoAddr,
  const unsigned bytes_to_transfer
);

// A wrapper function used to make a DMA transfer from a FIFO.
void startDMAfromFIFO(
  volatile unsigned long *DMA_ADDRESS,
  FIFO *fifoAddr,
  volatile unsigned long *destAddr,
  const unsigned bytes_to_transfer
);
```

`StartDMA` is used to first configure a DMA core that is mapped at `DMA_ADDRESS`, to read from `srcAddr` and write to `destAddr`, `bytes_to_transfer` number of bytes, with the size of each word being `wordSize`. Once these are configured, the DMA operation starts automatically. The `WORD_SIZE`, which represents the size of a single DMA transfer, is an `enum` type, which can either be a `BYTE` (1 byte), a `HALF_WORD` (2

bytes), a `WORD` (4 bytes), a `DOUBLE_WORD` (8 bytes), or a `QUAD_WORD` (16 bytes). The `WORD_SIZE` parameter is used to configure the DMA core. `ADDRESS_INCREMENT_SCHEME` is also an `enum`, which specifies how the source and destination addresses should or should not be incremented. An address should not incremented, for instance, if we are writing to a FIFO, where the DMA keeps writing to one end of the FIFO, which is mapped to a fixed address, with an accelerator taking data from the other end. `SRC_DEST_ADDR_CONSTANT` specifies that both addresses stay constant, `SRC_ADDR_INCREMENT` indicates that the source address increments while the destination address stays constant, `DEST_ADDR_INCREMENT` denotes that only the destination address increments, while `SRC_DEST_ADDR_INCREMENT` means that both addresses increment. The `pollDMA` function is used to wait until the DMA core mapped at `DMA_ADDRESS` has finished its DMA transfer, and `resetDMA` is used to reset the DMA core. `startDMAtoFIFO` and `startDMAfromFIFO` are wrapper functions used to transfer data to/from a FIFO, both of which call the `startDMA` function inside. The DMA functions read/write to memory-mapped registers of a DMA core to control its operation. Our DMA library can be used by including `legup/dma.h` in the source code.

For the DMA core itself, we currently use Altera's DMA Controller Core [99]. This DMA core offers basic but sufficient functionality to transfer a large chunk of data in burst mode. With this core, Altera also provides a DMA device driver that integrates into its HAL (Hardware Abstraction Layer) system library used for NIOS II systems. However, we found this device driver to be overly complicated and difficult to use, and impractical for non-NIOS II systems, thus we decided to write our own. We think that the DMA core itself can also be easily replaced in the future with our own implementation.

We show an example of using our DMA functions to transfer data from an input array in off-chip memory to an input buffer of an accelerator, then transfer data from an output buffer of the accelerator to an output array in off-chip memory.

```
1:  // get memory-mapped pointers for DMA core and accelerator input/output buffers
2:  volatile unsigned long *DMA_addr =
      (volatile unsigned long *) DMA_MEMORY_MAPPED_ADDR;
3:  volatile unsigned long *input_buffer_addr =
      (volatile unsigned long *) INPUT_BUFFER_MEMORY_MAPPED_ADDR;
4:  volatile unsigned long *output_buffer_addr =
      (volatile unsigned long *) OUTPUT_BUFFER_MEMORY_MAPPED_ADDR;
5:  // cast pointers for input/output arrays
6:  volatile unsigned long *input_addr =
      (volatile unsigned long*) input_array;
```

```
7:  volatile unsigned long *output_addr =
      (volatile unsigned long*) output_array;
8:  ...
9:  // reset DMA core
10: resetDMA(DMA_addr);
11: // transfer data from input array in off-chip memory to
       input buffer of accelerator
12: startDMA(DMA_addr, input_addr, input_buffer_addr, WORD,
                NUM_BYTES, SRC_DEST_ADDR_INCREMENT);
13: // wait until the DMA transfer is complete
14: pollDMA(DMA_addr);
15: // invoke accelerator execution
16: // transfer data from output buffer of accelerator to
       output array in off-chip memory
17: startDMA(DMA_addr, output_buffer_addr, output_addr, WORD,
                NUM_BYTES, SRC_DEST_ADDR_INCREMENT);
18: // wait until the DMA transfer is complete
19: pollDMA(DMA_addr);
```

First we create memory-mapped pointers for the DMA core and the input/output buffers of an accelerator, and cast the input/output arrays in off-chip memory (where the data is to be taken from and to be stored to) to the type used by our DMA function (lines 1–7). After resetting the DMA core (line 10), the DMA transfer from the input array in off-chip memory to the input buffer of the accelerator is initiated by calling the `startDMA` function (line 12). The processor waits until the DMA transfer is complete by calling the `pollDMA` function (line 14). Then, the data stored on the output buffer of the accelerator can also be copied back to the output array in off-chip memory in the same manner (lines 17 – 19). Note that when an OS is used, the `mmap` function must be used to allocate space for the DMA core, as well as to allocate contiguous non-cached off-chip memory regions for input/output arrays.

Figure 4.5 shows the hybrid ARM architecture with DMA support. For clarity, we only show the connections related to DMA operations, although the connections between the processor and hardware accelerators also exist, as shown previously. Within a hardware accelerator, there are on-chip RAMs, which are used as input and output buffers. The accelerators consume data from their input buffers to process, and store outputs to their output buffers. A DMA core is used to fill an input buffer with data

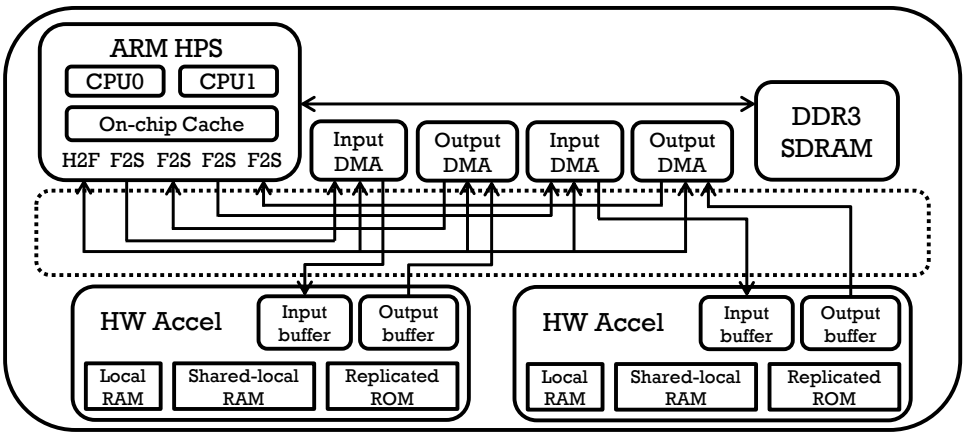


Figure 4.5: ARM hybrid architecture with DMA support.

from off-chip memory, and another DMA core is used to transfer data from an output buffer back to off-chip memory. Such an architecture allows concurrent input and output transfers, which is beneficial when overlapping data transfers with accelerator computations (i.e. the accelerator is continuously executing while data is being moved in and out of input/output buffers). This requires *double buffering* which is described below. If data transfers and computations are performed *sequentially*, it is also possible to have a single DMA core which manages both the input and output buffers. The `F2S` interfaces shown in Figure 4.5 denote the FPGA-to-HPS SDRAM interfaces, which connect directly to the DDR3 SDRAM without going through the L3 interconnect (the interconnect within the ARM HPS), the ACP port, or the caches, providing significantly higher memory bandwidth compared to accessing the cache. The ARM HPS supports having up to six `F2S` interfaces, depending on their bitwidths. Since this direct SDRAM interface does not go through the cache, it is not cache coherent. Therefore in the ARM hybrid systems which use DMA, we turn off the data cache in the ARM HPS via the ARM start up assembly code.

The sequence of operations for a hardware accelerator with DMA transfers works as follows: When data transfer and accelerator computation are performed *sequentially*, all input data is transferred to an input buffer first, then the accelerators starts to execute, storing all output data to an output buffer, then the outputs are subsequently copied from the output buffer to off-chip memory. When the data set is too large to be stored in input/output buffers at once, this operation can be divided into multiple iterations, where each iteration works on a section of data at a time. In scenarios where not all input/output data can be held in on-chip buffers at once, *overlapping* data transfers with computations becomes especially useful, and this requires *double buffering* to be effective. Double buffering requires there to be two instances of a buffer. For *input* double buffering, while a DMA core fills up input data to one of the

buffers, the accelerator consumes data from the other buffer. When the first buffer is completely filled with data, the DMA switches to the second buffer, with the accelerator now taking data from the first buffer. Similarly for *output* double buffering, the accelerator writes to one of the buffers, with the DMA taking data from the other, and the accelerator and the DMA core switch buffers when they are each done with their own. When both input *and* output are double buffered, and the system is in steady state (both input and output data transfers, as well as accelerator computations are all concurrently being done), DMA transfers operate as follows: If the accelerator is currently working on iteration `i`, the input DMA is transferring data for iteration `i+1` (to the input buffer not currently being used by the accelerator), and the output DMA is taking data from iteration `i-1` (from the output buffer not being written to by the accelerator).

Figure 4.6 shows the architecture of a hardware accelerator where its *output* data is double buffered. We provide the *Double Buffering Module*, which can simply be instantiated to use the double buffering functionality. A hardware accelerator wrapper module, which normally instantiates the hardware accelerator module (the compute core) and connects it to Avalon Interfaces, also instantiates the Double Buffering Module. This self-contained Double Buffering Module internally provides the logic to switch between buffers, thus its communicating modules (a DMA core and the accelerator module) do not need to be aware of double buffering, meaning that no changes are required for those modules to use double buffering. The DMA software functions called from the processor do not need to be modified either, since externally, it behaves as a single buffer. Within the Double Buffering Module, there are two sets of dual-ported RAMs, where the ports on one end connect to a DMA core, with the other end connecting to the accelerator module. There also also two sets of counters, one counter keeping track of the number of writes, and the other counter keeping track of the number of reads. When the number of writes reaches the depth of the RAM, it switches the RAM being written to, and the write counter resets. When the number of reads reaches the depth, the DMA reads are also steered to the other RAM, with the read counter resetting to zero. The Double Buffering Module also provides stall logic, which is important if the data rate for reads is not as fast as writes, and vice versa. The idea is that both the DMA core and the accelerator cannot be reading and writing from/to the same buffer at the same time, which can cause valid data to be overwritten. For example, for output double buffering, the accelerator module first completely fills the first buffer, then starts to write to the second buffer, at which time the DMA core starts to transfer data from the first buffer. If the DMA core cannot keep up with the accelerator module, so that the accelerator module completely fills the second buffer and switches to write to the first buffer again, the accelerator is stalled until the DMA core finishes transferring data from the first buffer. Along the same lines, if the accelerator cannot write data as fast as it is being read by the DMA

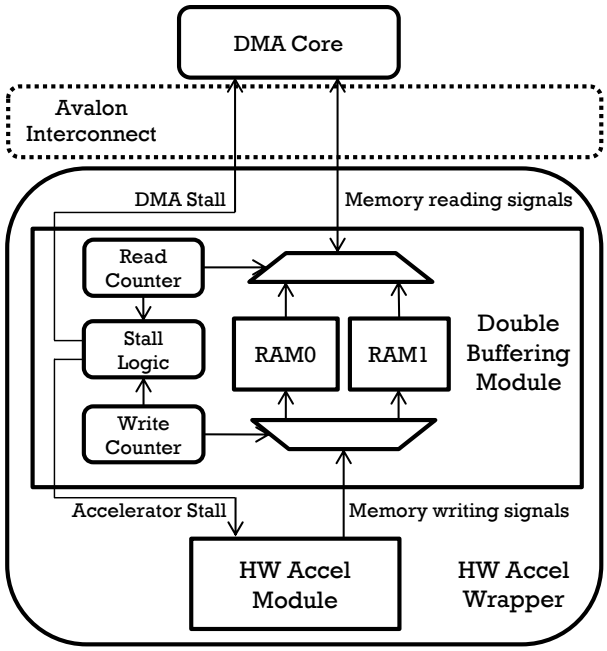


Figure 4.6: Hardware accelerator with output data double buffered.

core, and the DMA core attempts to start reading data from the same buffer which is being written to by the accelerator module, the DMA core is stalled until the accelerator module finishes writing to the buffer.

Note that we can also implement the DMA support with double buffering for the MIPS hybrid architecture, as shown in Figure 4.7. Again, the figure only shows those connections which relate to DMA operations. For this architecture, we have also removed the on-chip data cache in order to keep data coherent between the processor and accelerators (since accelerators write read/write directly from/to off-chip memory via DMA transfers).

Overall, our DMA support with the software library and the Double Buffering Module provides a method for transferring data in and out of hardware accelerators with high memory bandwidth – a necessity in achieving high performance for pipelined architectures. Currently, the instantiations of the DMA cores and the Double Buffering Module are not automatically handled by LegUp and need be done manually by the user, but we have plans to automate this in the future.

4.5 Experimental Study

In this section, we study the performance, area, and energy-efficiency of the ARM system, in comparison to the MIPS system, across six different benchmarks. For each type of system, we investigate four different architectures, which are:

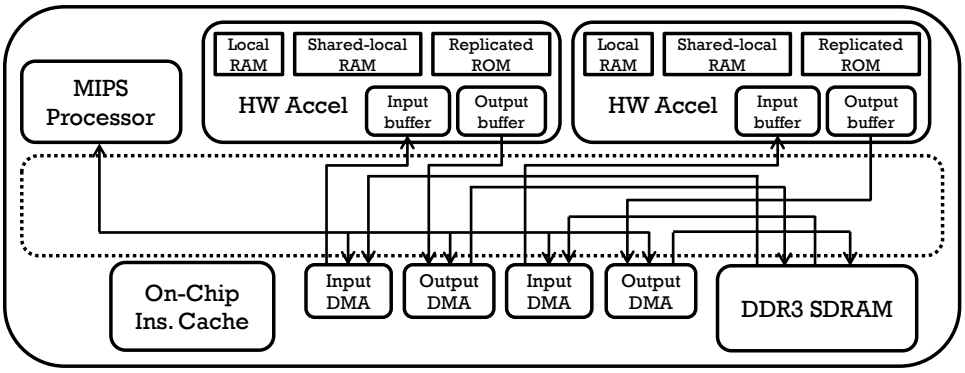


Figure 4.7: MIPS hybrid architecture with DMA support.

- The processor-only architecture, which executes the *entire* program in software, without any hardware accelerators, and without Pthreads (denoted as Arch. 0 in Section 4.5.2).
- The *single-threaded* processor-accelerator hybrid architecture, where the processor and a *single* hardware accelerator execute sequentially (denoted as Arch. 1).
- The *multi-threaded* processor-accelerator hybrid architecture, where the program is parallelized with Pthreads and compiled to *three* concurrent hardware accelerators (denoted as Arch. 2).
- Lastly, the *multi-threaded and pipelined* processor-accelerator hybrid architecture, where each concurrent hardware accelerator is also pipelined within (denoted as Arch. 3p).

Note that due to the structure of the benchmarks, not all of benchmarks were pipelinable, hence for the results shown in Section 4.5.2, an architecture with the `p` postfix includes results for only those benchmarks which could be pipelined (four out of six benchmarks). For the ARM/MIPS hybrid system, Arch. 3p (multi-threaded and pipelined) uses DMA tranfers to move data in and out of hardware accelerators (double buffering is not used in this case). All of the above systems (both processor-only and hybrid) run bare metal, without an OS.

We also examine the performance and energy consumption of one of the benchmarks, Black-Scholes, when running on a ARM hybrid system, versus when running purely in software on the ARM processor, as well as two different `x86` processors, the Intel Xeon E5-1650 and the Intel i7-4770K, where all of these systems (ARM, ARM hybrid, `x86`) are running Linux on their processors. The Intel Xeon E5-1650 (released in 2012) is a 32 nm six-core (12 cores with Hyper-Threading [125]) CPU, running at 3.2 GHz (3.8 GHz with TurboBoost) with 2 MB L2 and 12 MB L3 caches. It is running Ubuntu 14.04 and has 32 GB of DDR3 memory running at 1333 MHz. The Intel i7-4770K (released in 2013) is a 22 nm four-core (eight cores with Hyper-Threading) CPU, running at 3.5 GHz (3.9 GHz with TurboBoost) with 1 MB

L2 and 8 MB L3 caches. It is also running Ubuntu 14.04 and has 32 GB of DDR3 memory running at 1600 MHz. With this, we investigate the performance and energy consumption of the Black-Scholes benchmark when running on one, two, and three threads, on the ARM processor-only, ARM hybrid, and `x86` systems. For the ARM hybrid systems, we pipeline the threads and use DMA transfers *with* double buffering. For the `x86` systems, we also investigate using as many threads as the number of *logical* cores (number of cores seen with HyperThreading).

4.5.1 Benchmarks and Measurement Methodologies

We use a total of six benchmarks, each of which has a number of different variants. We create three versions of each benchmark, a sequential version, a parallelized version with Pthreads, and a parallelized version with Pthreads which uses DMA transfers. For running the Black-Scholes benchmark on the ARM hybrid systems with Linux, we use the `mmap` functions described previously. The six benchmarks are described below.

- Black-Scholes: Estimates the price of European-style options. It uses Monte Carlo simulation to compute the price trajectory for an option using random numbers. Computations are done in fixed-point.
- Dfdiv: Adopted from the CHStone benchmark suite [37], it computes double-precision floating-point division using 64-bit integers.
- Dfsin: Adopted from the CHStone benchmark suite [37], it computes the Sine function for double-precision floating-point numbers using 64-bit integers.
- Gaussian filter: As the first step of Canny edge detection, it is used to filter out noise by convolving the Gaussian filter with the image.
- Hash: Runs four different hashing algorithms.
- Mandelbrot: An iterative mathematical benchmark which generates a fractal image.

Each benchmark includes built-in inputs and golden outputs, with the computed result checked against the golden output at the end of the program to verify correctness. All input/output data reside in off-chip DDR3 memory, and for the benchmarks with DMA operations, each transferred output data from accelerators to off-chip memory is *individually* verified at the end of the program on the processor. Out of the six benchmarks, `Dfdiv` and `Dfsin` were not pipelinable. We synthesize each hardware system using Quartus 15.0 targeting the Altera Arria V SoC FPGA (5ASTFD5K3F40I3) to obtain area and

critical path delay (*Fmax*) numbers. For the MIPS systems, we use ModelSim to extract the total number of execution cycles and compute the total execution time (wall-clock time) as the product of the execution cycles and the post-routed clock period. To obtain power data, we use the post-synthesis timing simulation to generate a `VCD` (Value Change Dump) file with ModelSim, which is used by the Quartus PowerPlay Power Analyzer to get power numbers[3]. To get the execution time on ARM systems running bare metal (processor-only and hybrid), we use the Performance Monitoring Unit on the ARM processor [22] to get the processor cycle count, and divide this by the processor's clock speed, 1.05 GHz, to calculate the wall-clock time. We start the counter before starting the hardware functions, and stop after they have returned, which also includes DMA configuration and transfer times for those systems with DMA, but does not include the time to *verify* individual elements of transferred output data on the processor. For measuring power on the ARM systems, we use the Power Monitor on the Arria V SoC Development Kit Board [98], which measures real-time power consumption of both the HPS and the FPGA fabric via their power rails, providing accurate real-time power measurements.

As previously mentioned, we also compare the Black-Scholes benchmark when running a single thread, as well as on multiple threads, on the ARM processor-only, ARM hybrid, and `x86` systems, with all processors running Linux OS. We have made our best efforts to make this comparison as fair as possible. Before measuring, we reboot the systems to start from a clean slate and kill any user spawned processes. To compile the benchmark running purely in software, we use `gcc` with `-O3` optimization and the `-pthread` flag (on both ARM and `x86`). To obtain execution time, we use the `gettimeofday` function, a Linux function that can be used to get the current wall-clock time. We call it before `pthread_create` and after `pthread_join` (which includes DMA transfer times for the hybrid case) to get the total execution time. With the OS, if the total runtime is very short, the thread start-up and stop times can be a significant portion of the runtime, which puts the processor-only systems at a disadvantage. To avoid this, we increase the total number of Black-Scholes simulations done to the maximum that could fit in the 1GB HPS DDR3 memory, which was around 600 MB (each simulation output is stored in memory to be verified on the processor at the end of the program). Also with the OS, runtime is not always deterministic, so we execute the benchmark 100 times on all platforms and take the geometric mean. On the `x86` processors, we set the `governor` to *performance*, which allows the CPUs to run at maximum speed with TurboBoost. To measure the effect of TurboBoost on CPU frequency, we use `turbostat`, a Linux command-line utility which reports real-time stats such as frequency and temperature [81]. In addition, to get more consistent results over runs, we also use `taskset` [75], a Linux command which

[3]This method of measuring power consumption via a post-synthesis timing simulation is frequently used, however, it is reportedly to have a margin of error of about +/-10%.

allows *binding* a process to specific cores to avoid the process from jumping between different cores. We use the first three cores on both the Xeon and the i7 CPUs. To measure power on the `x86` processors, we use Intel's Performance Counter Monitor [147], which also has a utility to measure power consumption.

4.5.2 Results

Table 4.1 shows the geometric mean results for the MIPS processor-only system (Arch. 0/0p), where the entire program is executed in software on the processor, as well as for the single-threaded (Arch. 1/1p), multi-threaded (Arch. 2/2p), and multi-threaded and pipelined (Arch. 3p) hybrid architectures. The complete circuit-by-circuit results are shown in Appendix C. We show four different types of metrics: 1) Performance, which includes total execution time (in ms), total clock cycles, and Fmax (in MHz), 2) area, which includes the number of ALMs (Adaptive Logic Modules), registers, DSPs, and M10Ks (Altera's memory blocks which can hold 10 Kbits), 3) power, which includes static and dynamic power (in mW), as well as the sum of the two, and lastly, 4) efficiency, which include total energy consumption, calculated as the product of total execution time and total power, and area-delay product. To calculate the area-delay product, we first calculate the total chip area by using the data from [149, 64] (as was done in Chapter 3), and multiply this with total execution time to obtain the area-delay product for each architecture[4]. The last five lines of the table show relative ratios, comparing the results of each architecture to the processor-only (Arch. 0/0p) and the single-threaded hybrid architectures (Arch. 1/1p).

When comparing to Arch. 0/0p, all architectures exhibit significant improvements in performance, energy-efficiency, and area-delay product. With three concurrent cores (Arch. 2), we see a speedup of 53× in wall-clock time, while being 45.5× more energy-efficient. Fmax stays relatively constant, as the critical path is mostly limited by the on-chip cache and the interconnect in the processor system. As expected, the most notable improvements come from Arch. 3p, which uses pipelining with DMA transfers (without double-buffering). Compared to Arch. 0p, Arch. 3p shows 271.9× speedup and 212.4× energy-efficiency, with 104.2× better area-delay product. With three parallel pipelined accelerators, DSP usage increases notably by 17.6×, since the processor-only system only uses 6 DSP blocks. Other area elements also increase by 2.14× $\sim$ 3.36×, and with this, Fmax drops by 13.4% and total power consumption increases by 28%. In comparison to Arch. 1p, Arch. 3p shows improvements of 16.4×, 13.3×, 9.1× in wall-clock time, energy-efficiency, and area-delay product, respectively. Area also increases by 1.5× $\sim$ 3.2× for the

[4][149] provides detailed area data for Stratix III, which uses M9K memory blocks. We calculate the M10K area on Arria V from the given M9K area by using the relative memory capacity ratio. For ALMs, we assume that the area remains the same between Stratix III and Arria V. For DSP blocks, the Arria V uses Stratix V DSP blocks, hence we use the DSP area data from [64], which show that a DSP block uses an equivalent amount area to 30 ALMs.

different types of blocks, with total power consumption going up by 24%. Compared to Arch. 2p, Arch. 3p shows increases in ALMs and registers, as well as many more M10Ks (2.99×), due to pipelining, input/output buffers of accelerators, as well as DMA cores (which also use memory blocks). Overall, the three *pipelined* hardware accelerators with much higher memory bandwidth via DMA transfers (Arch. 3p) show significant improvements in performance, at the expense of relatively moderate area increase.

Table 4.2 shows the geomean results for ARM processor-only and hybrid systems running bare metal. The power consumption, obtained via the Power Monitor, is shown in terms of HPS power (which includes power for the HPS core, HPS I/O, HPS DDR3, and HPS internal/peripheral devices [97]), FPGA power (which includes power for FPGA core/clock, FPGA I/O, and FPGA internal/peripheral devices), and total power (the sum of the two). We do not show area-delay product results since the HPS area is unknown. All clock cycle numbers shown on this table are in terms of processor cycles, and the execution times are obtained by dividing the number of clock cycles by the processor speed, 1.05 GHz. The Fmax results shown for Arch. $1 \sim 3p$ are the frequencies of the PLLs that are driving hardware accelerators' clocks. The HPS processor-only system (Arch. 0), consumes a small amount of FPGA area due to its peripherals such as JTAG UART, which is used by the processor to communicate with host PC (using `printf`), as well as the interconnect. Note that the FPGA fabric also consumes about 340 mW in this case, even when all processing is done on the HPS. With the hard ARM processor running at over 1 GHz, the processor-only system can achieve fairly good performance. With this, in addition to the limited memory bandwidth of accessing the on-HPS cache through the ACP port from hardware accelerators, Arch. 1 shows a *slow down* of 2.19× compared to Arch. 0. With 4.4% more total power consumption (31.3% more FPGA fabric power), energy consumption is also 2.28× of Arch. 0. This trend continues to Arch. 2, which exhibits 11.2% more runtime than Arch. 0, with 10.5% more total power (63.2% more FPGA fabric power) and 22.8% more energy consumption. This shows that non-pipelined hardware with limited memory bandwidth cannot outperform the hard ARM processor. On the other hand, Arch. 3p, which has three *pipelined* hardware accelerators with direct access to DDR3 via DMA, shows significant improvements over the ARM processor system. The architecture shows 31.5× speedup in wall-clock time with 26.6× better energy-efficiency. The hybrid systems with LegUp-generated hardware accelerators can significantly outperform a 1 GHz ARM processor in both performance and energy-efficiency. When compared to Arch. 1p, Arch. 3p is 60.8× faster and 54× more energy efficient.

Table 4.1: Geometric mean results for MIPS processor-accelerator hybrid systems.

	Performance			Area				Power			Efficiency	
Architecture	**Time (ms)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**	**Static Power**	**Dyn. Power**	**Total Power (mW)**	**Energy (mJ)**	**Area-delay Product**
Arch. 0	67.22	8,618,499.64	128.21	9,629.00	12,444.00	6.00	76.00	1,339.35	325.94	2,218.01	149.10	1,811.04
Arch. 0p	69.70	8,935,681.09	128.21	9,629.00	12,444.00	6.00	76.00	1,339.35	325.94	2,218.01	154.59	1,877.74
Arch. 1	3.36	420,025.54	124.93	14,683.81	23,363.61	55.20	80.13	1,340.18	466.13	2,314.93	7.78	139.86
Arch. 1p	4.22	527,468.28	125.12	13,711.87	21,310.13	49.39	79.22	1,339.96	443.61	2,290.03	9.65	164.47
Arch. 2	1.27	159,685.16	125.83	21,610.78	38,805.89	129.30	86.34	1,342.50	711.96	2,579.92	3.27	78.90
Arch. 2p	1.60	196,517.80	122.86	17,357.50	30,753.96	105.44	85.52	1,341.40	604.20	2,457.33	3.93	81.86
Arch. 3p	0.26	28,444.26	110.98	20,621.39	39,706.97	105.44	255.40	1,345.27	987.15	2,839.69	0.73	18.03
Arch. 1 / Arch. 0 Ratio	0.050 (19.99×)	0.049 (20.52×)	0.974	1.525	1.878	9.199	1.054	1.001	1.430	1.044	0.052 (19.16×)	0.077 (12.95×)
Arch. 2 / Arch. 0 Ratio	0.019 (52.97×)	0.019 (53.97×)	0.981	2.244	3.118	21.550	1.136	1.002	2.184	1.163	0.022 (45.54×)	0.044 (22.96×)
Arch. 3p / Arch. 0p Ratio	0.004 (271.93×)	0.003 (314.15×)	0.866	2.142	3.191	17.574	3.361	1.004	3.029	1.280	0.005 (212.40×)	0.010 (104.15×)
Arch. 2 / Arch. 1 Ratio	0.377 (2.65×)	0.380 (2.63×)	1.007	1.472	1.661	2.343	1.077	1.002	1.527	1.114	0.421 (2.38×)	0.564 (1.77×)
Arch. 3p / Arch. 1p Ratio	0.061 (16.44×)	0.054 (18.54×)	0.887	1.504	1.863	2.135	3.224	1.004	2.225	1.240	0.075 (13.26×)	0.110 (9.12×)

Table 4.2: Geometric mean results for ARM processor-accelerator hybrid systems.

	Performance			Area				Power			Efficiency
Architecture	**Time (ms)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**	**HPS Power**	**FPGA Power**	**Total Power (mW)**	**Energy (mJ)**
Arch. 0	1.50	1,578,439.36	1,050.00	1,650.00	2,551.00	0.00	4.00	1,754.73	340.73	2,095.59	3.15
Arch. 0p	2.20	2,305,313.96	1,050.00	1,650.00	2,551.00	0.00	4.00	1,756.03	339.87	2,096.10	4.60
Arch. 1	3.29	3,453,671.54	140.96	6,187.96	12,420.47	48.86	7.25	1,739.42	447.47	2,187.77	7.20
Arch. 1p	4.24	4,450,387.83	146.78	5,173.58	10,337.21	43.12	6.90	1,754.34	452.35	2,207.30	9.36
Arch. 2	1.67	1,754,443.41	125.66	13,904.07	30,795.08	122.05	13.14	1,755.17	556.22	2,314.79	3.87
Arch. 2p	1.78	1,869,232.14	127.29	10,417.79	23,748.90	98.28	12.54	1,749.68	511.65	2,262.07	4.03
Arch. 3p	0.07	73,167.36	144.57	15,501.36	33,486.58	98.28	121.43	1,767.26	716.93	2,484.79	0.17
Arch. 1 / Arch. 0 Ratio	2.188 (0.46×)	2.188 (0.46×)	0.140	3.750	4.869	48.858	1.812	0.991	1.313	1.044	2.284 (0.44×)
Arch. 2 / Arch. 0 Ratio	1.112 (0.90×)	1.112 (0.90×)	0.125	8.427	12.072	43.116	3.284	1.000	1.632	1.105	1.228 (0.81×)
Arch. 3p / Arch. 0p Ratio	0.032 (31.51×)	0.032 (31.51×)	0.144	9.395	13.127	122.050	30.357	1.006	2.109	1.185	0.038 (26.58×)
Arch. 2 / Arch. 1 Ratio	0.508 (1.97×)	0.508 (1.97×)	0.891	2.247	2.479	2.498	1.812	1.009	1.243	1.058	0.537 (1.86×)
Arch. 3p / Arch. 1p Ratio	0.016 (60.82×)	0.016 (60.82×)	0.985	2.996	3.239	2.280	17.596	1.007	1.585	1.126	0.019 (54.03×)

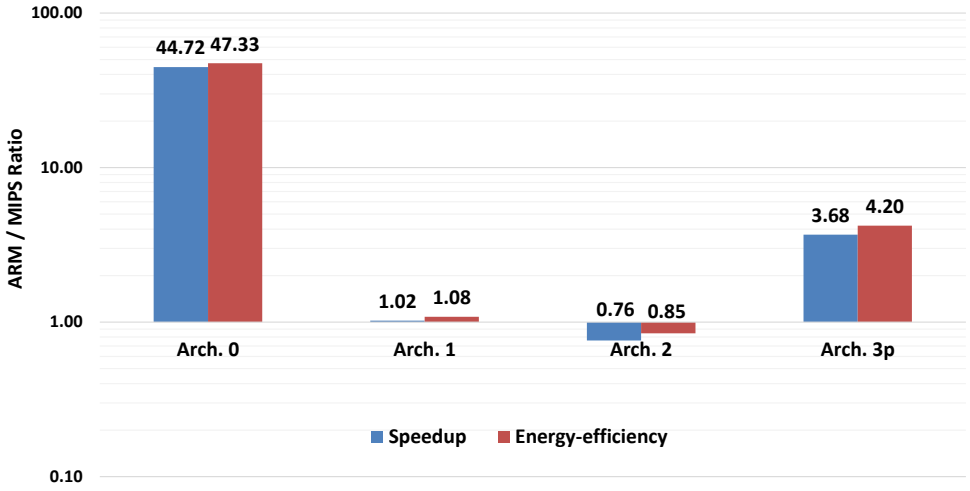


Figure 4.8: Relative Speedup and energy-efficiency ratios comparing ARM to MIPS.

As all scenarios were implemented on the *same* FPGA, for the *same* set of benchmarks, it can be interesting to contrast the results of the ARM and the MIPS systems. Figure 4.8 shows the relative speedup and energy-efficiency ratios (in logarithmic scale) for each architecture, with the MIPS system used as the baseline for each architecture. It can be seen that for Arch. 0, where the *all* computations are performed in software, the ARM processor shows significant speedup and energy-efficiency compared to the MIPS (44.72× and 47.33× respectively), showcasing the power of the hardened processor. When *most* computations are moved to a single accelerator in Arch. 1, both systems exhibit close results. When computations are *parallelized* with Pthread accelerators in Arch. 2, the ARM hybrid system shows a *slow down*, as the memory latency, hence the number of cycles the accelerators are stalled due to memory contention from concurrent accelerators attempting to access the cache, is larger for the ARM hybrid architecture. However, the memory bandwidth is improved in Arch. 3p, since the HPS offers multiple FPGA-to-HPS SDRAM interfaces, allowing the parallel accelerators to access the off-chip memory concurrently. With this, the ARM hybrid systems for Arch. 3p show 3.68× speedup and 4.20× energy-efficiency improvement over the MIPS hybrid systems[5].

Table 4.3 shows the performance, power, and energy consumption results of the Black-Scholes benchmark when executing purely in software on the ARM (denoted as ARM SW), the Intel Xeon E5-1650 (Xeon SW), and the Intel i7-4770K (i7 SW), as well as when running on the ARM hybrid architecture (ARM Hybrid). On each platform, we increase the number of threads from one (1T), to two (2T), to three (3T). All architectures are running Linux on their processors and all hybrid systems are using DMA transfers with double buffering. The table also shows the frequencies of the `x86` processors with

[5]If memory bandwidth was the same in both cases, we think that the performance of the ARM and MIPS for Arch. 3p would be close to each other (similar to Arch. 1), as *most* computations are off-loaded to accelerators in this case.

TurboBoost, measured with `turbostat` while the benchmark is being executed. All frequency values fall within the range of their turbo bins provided by Intel, which indicate the frequency increases that can be achieved with TurboBoost for each number of active cores [129]. In general, the table shows that for each architecture, as the number of threads is increased, the performance improves, with increasing power consumption. The results remain similar however, for ARM SW 2T to ARM SW 3T, as the ARM processor has a *dual*-core CPU.

Table 4.3: Results for Black-Scholes on ARM processor-only, ARM hybrid, and `x86` architectures.

Architecture	**Time (s)**	**Power (W)**	**Energy (J)**
ARM SW 1T	50.717	2.128	107.921
ARM SW 2T	25.418	2.327	59.159
ARM SW 3T	25.870	2.337	60.453
ARM Hybrid 1T	1.184	2.301	2.725
ARM Hybrid 2T	0.642	2.529	1.622
ARM Hybrid 3T	0.484	2.683	1.297
Xeon SW 1T @ 3.75 GHz	3.312	55.214	182.878
Xeon SW 2T @ 3.75 GHz	1.668	69.699	116.273
Xeon SW 3T @ 3.65 GHz	1.144	79.234	90.640
i7 SW 1T @ 3.9 GHz	2.735	24.096	65.909
i7 SW 2T @ 3.9 GHz	1.372	26.504	36.355
i7 SW 3T @ 3.8 GHz	0.944	35.459	33.476

To make the comparisons *between* the architectures easier, we plot their relative ratios on Figures 4.9 and 4.10. The x-axis shows the different architectures, with increasing number of threads from left to right, and the y-axis shows their ratios in logarithmic scale. Each plotted line shows the ratio values of a single architecture when compared across all other architectures, and for readability, we only plot five different architectures (out of 12) as shown in the legends. Figure 4.9 shows the speedup results for the Black-Scholes benchmark, where the hybrid architecture speedups are highlighted in bold. First, when comparing ARM Hybrid 3T to i7 SW 3T (running at 3.8 GHz), Xeon SW 3T (running at 3.65 GHz), and ARM SW 3T (running at 1.05 GHz), we see speedups of 1.95×, 2.37×, 53.51× respectively. It is also worth noting that ARM Hybrid 3T shows 104.9× speedup compared to ARM SW 1T. This speedup is much larger than what was shown in Table 4.2 for Arch. 3p/1p mainly due to double-buffering. Even with two concurrent hardware accelerators (ARM Hybrid 2T), we can still outperform the triple threaded case of software, by 1.47× (i7 SW 3T), 1.78× (Xeon SW 3T), and 40.32× (ARM SW 3T). The single hardware accelerator system (ARM Hybrid 1T) also runs faster than the single-threaded software executing on the i7 (running at 3.9 GHz), the Xeon (running at 3.75 GHz), and the ARM (running at 1.05 GHz), by 2.31× (calculated from Table 4.3), 2.8×, and 42.84× respectively. The LegUp-generated hardware systems can surpass the performance of Intel CPUs which are running *much* faster in clock

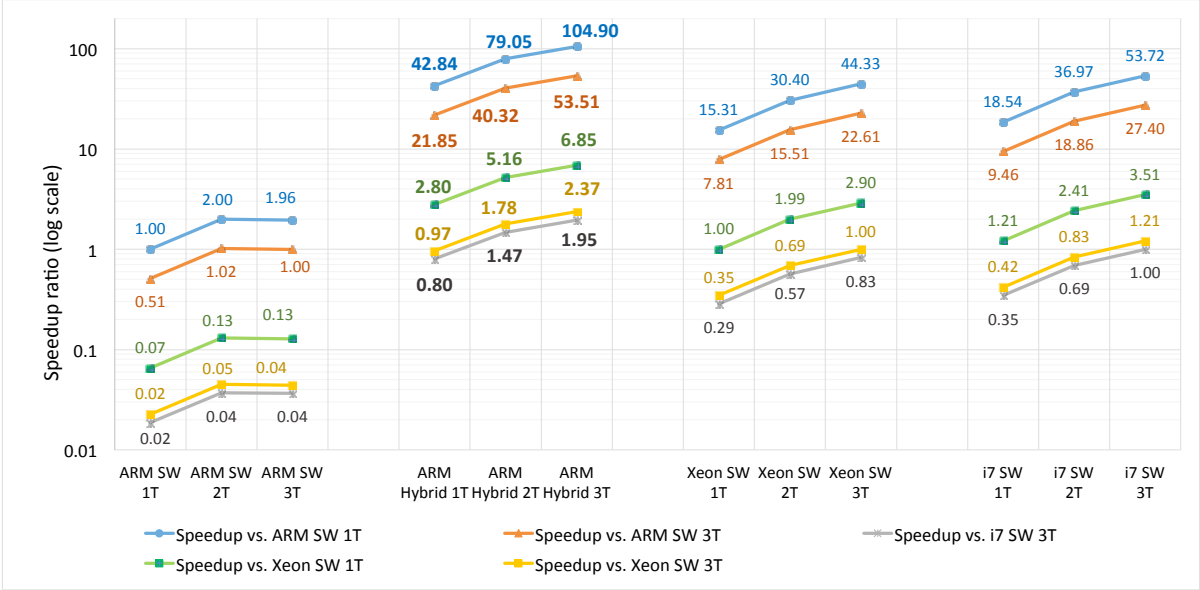


Figure 4.9: Speedup ratios for Black-Scholes on ARM processor-only, ARM hybrid, and `x86` architectures.

speeds (120 MHz vs. ∼4 GHz).

Energy-efficiency is also another strength of FPGAs, and this demonstrated in Figure 4.10, where all hybrid systems exhibit at least an order of magnitude improvement in energy-efficiency compared to all SW systems. Compared to i7 SW 3T, Xeon SW 3T, and ARM SW 3T, ARM Hybrid 3T shows 25.81×, 69.87×, 46.6× better energy-efficiency. The Xeon processor consumes considerable amount of power, so the ARM Hybrid 3T is 140.98× more energy efficient than Xeon SW 1T. Even with a single accelerator (ARM Hybrid 1T), we observe improvements of 12.29× and 33.27× compared to i7 SW 3T and Xeon SW 3T, and 24.19× (calculated from Table 4.3) and 67.12× when compared to i7 SW 1T and Xeon SW 1T. Again, this demonstrates that we can achieve significant improvements in energy-efficiency with LegUp-generated ARM hybrid systems.

One may question, however, that there are unutilized cores on the Intel CPUs which can simply be used by forking more threads. To investigate this, we modified the Black-Scholes benchmark to use as many threads as the number of logical cores (number of cores with HyperThreading) for each Intel CPU. Table 4.4 shows the result for the Xeon CPU when using 12 threads, and for the i7 CPU with 8 threads, as well as their relative ratios compared to ARM Hybrid 3T. By utilizing more cores, both CPU frequencies with TurboBoost have dropped as expected, in accordance to their turbo bin values. In terms of total execution time, ARM Hybrid 3T still outperforms the Intel CPUs, by 4% compared to Xeon SW 12T, and by 20% compared to i7 SW 8T. The power consumption of both CPUs also increase with more utilized cores, thus we still see significant energy-efficiency improvements of 42.97× compared to Xeon SW 12T,

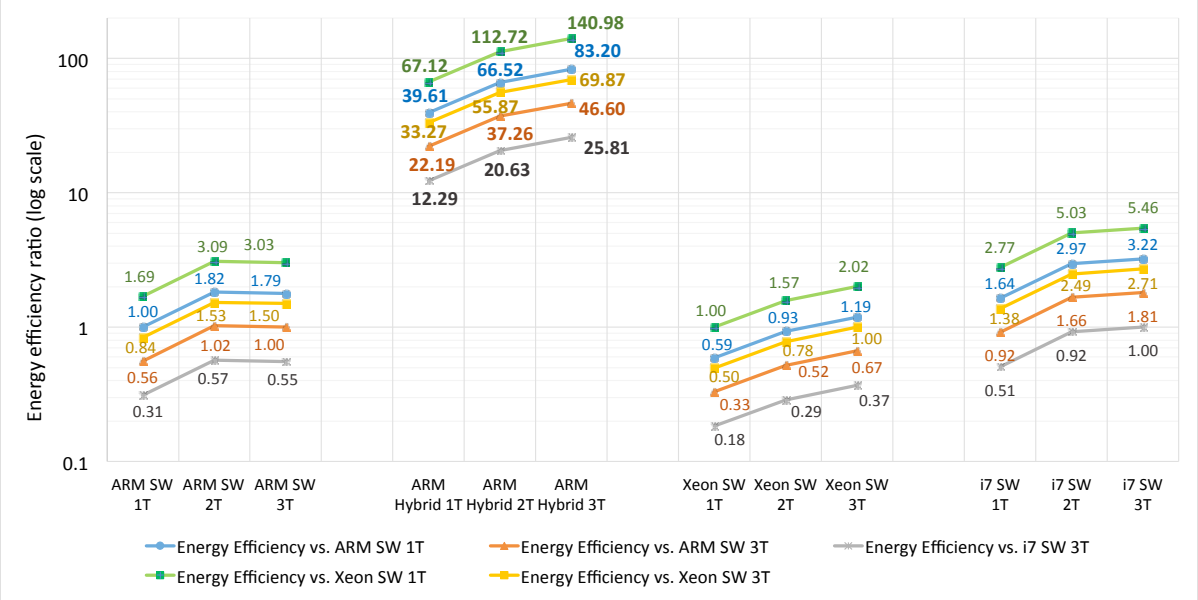


Figure 4.10: Energy-efficiency ratios for Black-Scholes on ARM processor-only, ARM hybrid, and `x86` architectures.

and 20.49× compared to i7 SW 8T. Hence, the LegUp ARM hybrid system still provides substantial benefits even when all Intel CPU cores are utilized. Moreover, additional hardware accelerators can be instantiated in the ARM hybrid case as well, since the ARM Hybrid 3T only consumes about 10% of logic and 7% of DSP blocks on the Arria V SoC FPGA. We expect that having more accelerators will increase performance until it is limited by memory bandwidth (DMA to/from off-chip DDR3).

Table 4.4: Results for Black-Scholes on `x86` processors when using as many threads as the number of cores.

Architecture	Time (s)	Power (W)	Energy (J)
Xeon SW 12T @ 3.5 GHz	0.504	110.560	55.740
vs. ARM Hybrid 3T	0.484 (1.04×)	2.68 (41.21×)	1.30 (42.97×)
i7 SW 8T @ 3.65 GHz	0.578	45.990	26.580
vs. ARM Hybrid 3T	0.484 (1.20×)	2.68 (17.14×)	1.30 (20.49×)

4.6 Summary

In this chapter, we described our support for using the ARM Hard Processor System on the Arria V SoC to accelerate multi-threaded software applications. This significantly enhances the processor-accelerator hybrid flow of LegUp, which was previously limited to using only the soft MIPS processor. With respect to the ARM processor, we provide support for running bare metal (no OS), as well as running a Linux OS. We also discussed our DMA software library and the Double Buffering Module, which can transfer

large chucks of data in bursts to and from off-chip memory, essential for achieving high performance in pipelined circuits.

Compared to software executing on the MIPS processor, our multi-threaded and pipelined hybrid systems obtained over two orders of magnitude improvements in performance and energy-efficiency. Compared to the ARM processor, however, our multi-threaded hybrid systems showed a slow-down, due to the limited memory bandwidths of accessing the on-HPS cache memory. However, pipelining the accelerators and enabling DMA data transfers improved the performance and energy-efficiency of the hybrid systems to 31.5× and 26.6× of software executing on the ARM. Even compared to `x86` processors, our ARM hybrid system running on a 28 nm FPGA, outperformed multi-thereaded software running on a 22 nm Intel i7 CPU running at 3.8 GHz by ∼2×, and a 32 nm Intel Xeon CPU running at 3.65 GHz by 2.4×, while consuming only 3.9% and 1.4% of their energy, respectively. Even when all eight cores of the i7 CPU and all twelve cores of the Xeon CPU were utilized, the ARM hybrid system outperformed the `x86` processors, all the while consuming 4.9% and 2.3% as much energy as the CPUs. The HLS framework can generate entire SoCs that can show significant benefits in term of speed and energy-efficiency compared to the MIPS, the ARM, and the `x86` processors.

This work is to be submitted to IEEE Transactions on Very Large Scale Integration Systems (TVLSI).

Chapter 5

Synthesis of Software Threads to Parallel Hardware-only System

5.1 Introduction

In Chapters 3 and 4, we described our support for generating a processor-accelerator hybrid system where parallel hardware accelerators can work in tandem with an embedded processor. This allows one to benefit from the flexibility of a processor, while taking advantage of the performance and energy-efficiency benefits offered by FPGA hardware. However, we note that for some applications, it may be beneficial to compile the program to *purely* hardware, instead of using the processor-accelerator hybrid system. ARM SoC FPGAs are still relatively nascent, with only a handful of SoC FPGAs on the market, and soft processors can add significant area/power overheads. It is therefore desirable if the Pthreads/OpenMP HLS support described previously can also be used without a processor. In this chapter, we describe our support for the *hardware-only* flow, where a multi-threaded software program can be compiled entirely to hardware, with parallel threaded modules executing concurrently within a hardware system. By removing the processor requirement, we believe that our HLS support for Pthreads/OpenMP can be applied to a wider range of applications.

The new flow is beneficial in a number of ways: 1) When the entire program is intended for implementation in hardware, 2) when having a processor is not practical due to limited area resources, or an SoC board is not available, or 3) when the circuit is to be used as part of a larger framework where it will be *plugged in* as a module. For 1), some applications domains, such as high frequency trading, require very low latency computations. To meet low-latency requirements, all computations from inputs

to outputs may need to be done in hardware. For 3), the hardware-only flow also allows the user to specify a portion of the program, such as those computations executed in threads, to be compiled to hardware, with the rest of the program acting as a testbench. This can be useful if the hardware is to be used as a part of a larger framework. Note that this differs from the hybrid flow, which generates a complete SoC, including the processor, on-chip caches, and interconnect.

Overall, the hardware-only flow with HLS support for Pthreads and OpenMP provides an efficient and a convenient method to express hardware parallelism using software threads. We are unaware of any other active HLS tools which support the synthesis of Pthreads and OpenMP into a purely hardware architecture, where parallel threaded modules execute within a larger circuit.

5.2 Parallel Hardware-only System Generation

When synthesizing a parallel hardware-only system, two major steps are performed which differ from those for generating a non-parallel sequential circuit. One of these steps, as shown in Figure 5.1, is `ParallelAPI`, the same compiler pass used in the hybrid flow to handle Pthread/OpenMP library functions and insert thread managing logic. Keeping a consistent flow with code re-use greatly improves the maintainability and the modifiability of our framework, as we have many compiler passes and supported features in LegUp HLS. The second major step that is different is performed in the Hardware Backend, and is responsible for creating the parallel hardware modules. In the hardware-only flow, we do not use Qsys, which allows one to flexibly generate an interconnection fabric, but it also adds *substantial* area overhead. We believe such overhead to be unjustified when many of the useful features of the Avalon Interconnect, such as memory-mapped communication, are not used in a hardware-only system. We have built our own *system generator*, which can generate a *custom* interconnect allowing different components in a hardware-only system to communicate and work together. The system generator is responsible for instantiating multiple parallel hardware instances for a threaded module in a hardware-only system (by default, it instantiates as many instances as the number of threads) and also creates necessary arbitration logic to allow the parallel modules to share memories. The system generator is described in more detail in Chapter 6.

Returning to the `ParallelAPI` pass, the majority of the work done in the pass remains the same as in the hybrid flow, where the pass transforms the Pthreads/OpenMP library function calls to the *native* function calls supported by LegUp. For OpenMP, the same code used for the hybrid flow is used in the hardware-only flow to replace calls to `GOMP_parallel_start` with calls to the *outlined* OpenMP functions (with as many calls to the outlined function as specified by the `num_threads` argument in

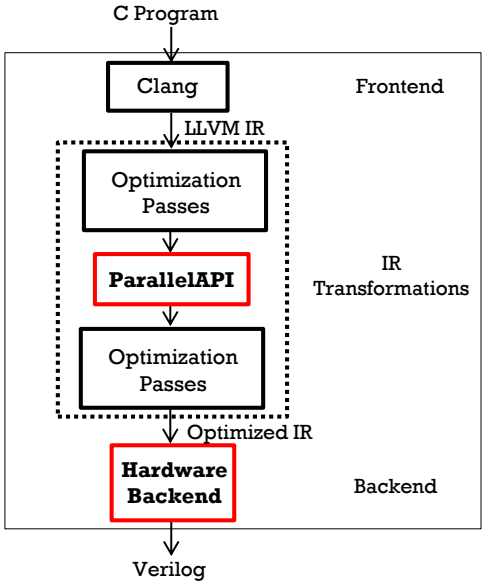


Figure 5.1: Parallel hardware-only flow in LegUp HLS.

`GOMP_parallel_start`). As before, each outlined OpenMP instance is statically assigned a thread ID, which is passed in as its argument. In the Hardware Backend, when we generate an FSM for the module which invokes the parallel OpenMP modules, all calls to the parallel modules for the *same* outlined OpenMP function are scheduled in the *same* state (occur in the same cycle), since we know exactly which and how many OpenMP modules to invoke (due to the one-to-one mapping described in Section 3.4). After starting the OpenMP modules, the FSM remains in the same state until *all* of the OpenMP modules complete their work (analogous to the blocking behaviour of OpenMP).

Figure 5.2 shows the architecture of a hardware-only system generated for a program where the *main* function invokes two OpenMP functions, `omp_0` and `omp_1`, each of which runs on two threads. For an OpenMP module, the same start signal is connected to all of its instances, as all instances are started at the same time. The same argument signal (if one exists) is also connected to all instances of the same OpenMP function, as OpenMP builds a `struct` that is passed in as a pointer to the function, with each thread accessing a different field of the `struct` as its argument during execution. An outlined OpenMP function does not have a return value; outputs are communicated through memory. All *finish* signals for a particular OpenMP function are `AND'ed`, to make sure that all of its instances finish before the FSM in the *main* module can continue to execute.

Note that parallel hardware instances can also access memories, which can be local to a thread (local RAM), or shared between between multiple threads (shared-local RAM), where the points-to analysis is used to determine which memories are accessed by which hardware modules. The system generator automatically creates a round-robin arbiter for each shared-local RAM that is accessed by multiple

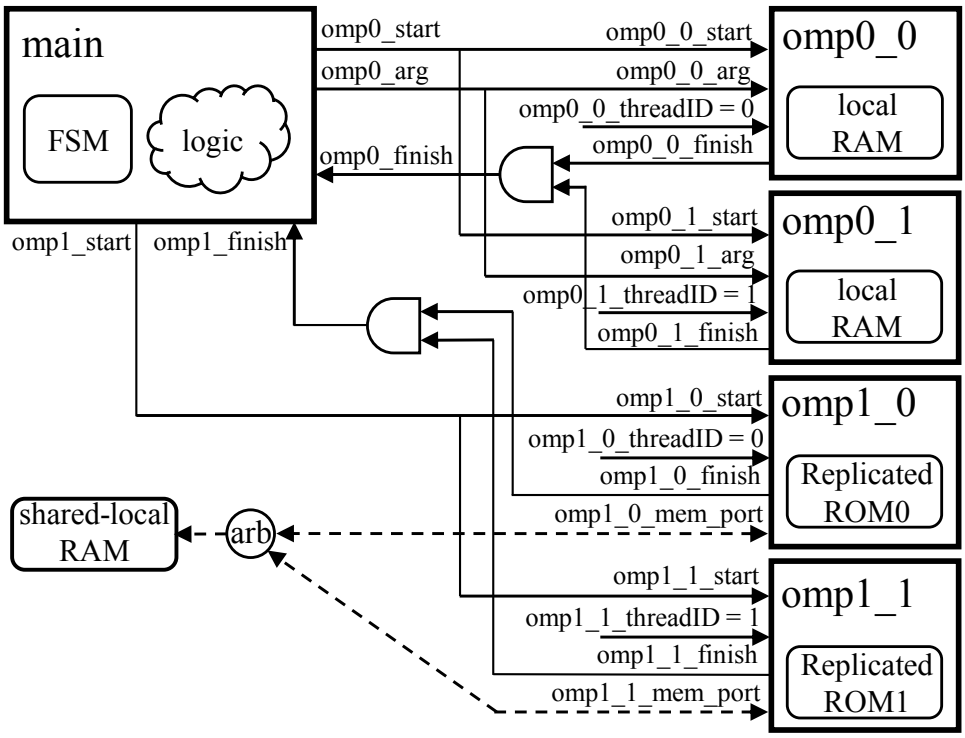


Figure 5.2: Hardware-only system architecture for OpenMP.

concurrent hardware modules (an `OR` gate is created when hardware modules execute sequentially). For each shared-local RAM, a dedicated memory port is created from the accessing module, so that independent memories can be accessed in parallel. As in the hybrid flow, we can also replicate read-only memories across threads (Replicated ROM), which helps to reduce memory contention and improve performance.

For Pthreads, there is considerable shared functionality with the hybrid flow in the `ParallelAPI` pass, as well as new implementation specific to the hardware-only flow. As in the hybrid flow, a call to `pthread_create` is replaced by a direct call to the threaded function, and `pthread_join` is replaced with `legup_pthreadpoll`. A global variable is created for each *different* Pthread to act as its *thread ID*. When there is more than one Pthread function, we statically assign a *function ID* to each different function. In the hybrid case, the memory-mapped address of the *first* accelerator for a Pthread function was used as its function ID, as its subsequent accelerators were accessed by offsetting the base memory-mapped address with the value of the thread ID. Since we do not have a memory-mapped interconnect in a hardware-only system, we needed to create a *memory-mapped-like* feature, which can be used in hardware to fork and join parallel modules. We again use the thread variable to achieve this. On a call to a Pthread function, we store into the thread variable, its function ID, together with its thread ID. We use the top 16 bits of the thread variable to store the function ID, with the bottom 16 bits storing

the thread ID (i.e. function ID << 16 OR threadID). This allows us to use the existing thread variable without having to create another storage element. Assigning 16 bits to store each portion limits the maximum number of different Pthread functions, and the number of threads per Pthread function to 65,535, but we think this is more than enough for all practical purposes[1]. The thread variable is again used in `legup_pthreadpoll` to determine which parallel instance to be poll.

It is worth nothing that one of the advantages of using an LLVM pass to generate the function ID and thread ID counters is that, in many cases, the compiler can determine the values of the IDs at compile time, so that they simply become direct stores of constant values to the thread variable. For instance, when two different functions with two threads are forked, LLVM can simply turn the storing of (function ID << 16 OR threadID) into storing four constants: `0` (first thread of first function), `1` (second thread of first function), `65536` (first thread of second function), and `65537` (second thread of second function). This removes the need to load and increment a thread ID counter, which becomes a memory load and an addition in hardware. However, even if the compiler is not able perform such optimizations, due to control flow that cannot be resolved at compile time, our generated logic will work at runtime.

`legup_pthreadpoll` (replaces `pthread_join`) uses the thread variable to check whether its corresponding parallel instance finishes execution, and retrieve its return value. Unlike in software wrapper generation, we do not create a function definition for `legup_pthreadpoll`, but we simply use the call to `legup_pthreadpoll` as a *placeholder* to determine *when* a parallel module needs to be joined. Before a call to `legup_pthreadpoll`, we create a load to retrieve the value of the thread variable. Then in the Hardware Backend, the system generator creates multiplexers which can select between all Pthread modules, and the value of the thread variable is used as the *select* signal of the multiplexers. The multiplexers are used to check the `finish` signal of a Pthread module, and retrieve its return value. Generating the multiplexers directly in the Hardware Backend produces a more efficient hardware implementation (compared to generating LLVM IR that gets compiled to multiplexers), since we know exactly what kind of hardware to create. This is similar to the hybrid case, where Qsys generates the memory-mapped interconnect directly in hardware.

Figure 5.3 shows the hardware architecture created for a hardware-only system with two Pthread functions, each of which uses two threads. As mentioned previously, each thread becomes an independent hardware *core* by default. The *solid* arrows in the figure depict signals for calling and joining Pthread modules; the *dotted* arrows indicate signals for memory. An FSM in the caller module (`main` in this case) steps through hardware states to control the circuit, and invokes the Pthread hardware modules.

[1] If needed, the pass can be modified to allocate different number of bits to the function or thread IDs, or even create a separate variable to store the function ID.

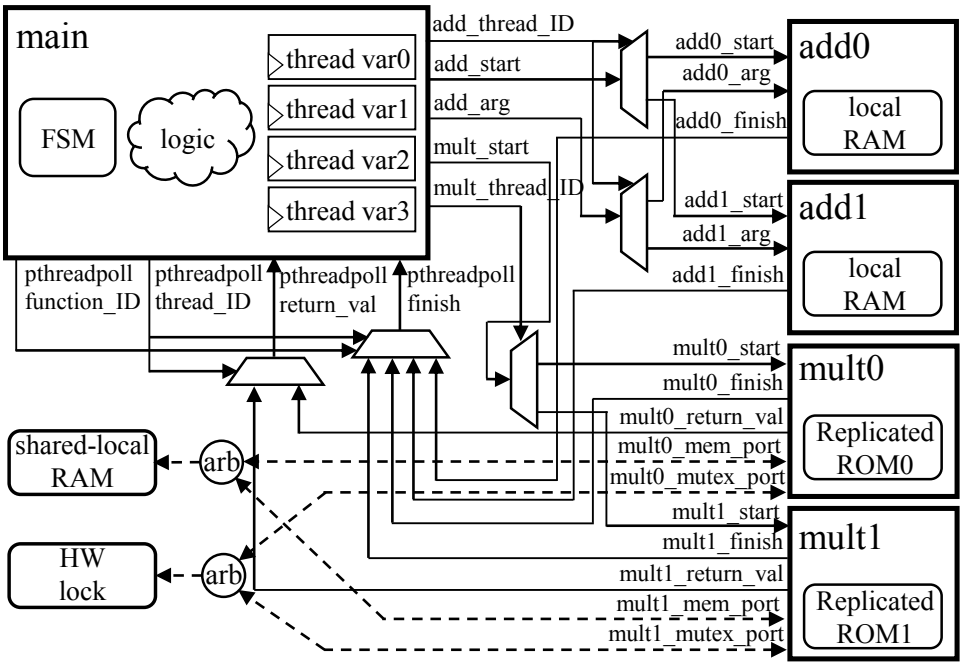


Figure 5.3: Hardware-only system architecture for Pthreads.

In a non-Pthreaded sequential circuit, when an FSM invokes a hardware module, it remains stalled in the same state until the hardware module is done, analogous to the behaviour of sequential software (as well as for OpenMP). However, the FSM is handled differently for Pthreads, as its behaviour is non-blocking, where the program continues to execute after forking threads. Similarly in LegUp, the FSM in the caller module continues to execute after invoking the parallel hardware instances. To start and send arguments to a Pthread module, demultiplexers are generated to steer data to the correct parallel instance, by using the thread ID as the *select* signals. This thread ID is the value of the global variable created for the particular Pthread function. As shown in the figure, the thread ID is sent out on an output port from the caller module to the demultiplexers used to select an instance of a Pthread module. After invoking a parallel instance, the caller module stores the thread ID *combined* with the function ID into the thread variable. The thread variables are stored in registers, residing in the caller module. When the FSM reaches the state corresponding to `legup_pthreadpoll`, it stalls until the hardware instance corresponding to the value of the thread variable is done, matching the software behaviour of `pthread_join`. The caller module loads the value of the thread variable and sends out the top 16 bits on the `pthreadpoll_function_ID` output port, with the bottom 16 bits driven on the `pthreadpoll_thread_ID` output port. Multiplexers are also created, which uses the function/thread ID values from their output ports to select the correct parallel instance. A multiplexer selects the `finish` signal using the function/thread IDs, which is checked by the caller FSM. Once the *finish* is asserted, the

caller FSM proceeds to retrieve the return value in the same manner, if one exists for the module. For memories, Pthread modules can also have local/shared-local RAMs and replicated ROMs, as was shown for the OpenMP architecture above. Parallel hardware instances (for both Pthread and OpenMP) can also access hardware mutexes and barriers, where a round-robin arbiter is created for each mutex/barrier module.

5.2.1 Sharing a Hardware Core Across Threads

As was described in Chapter 3 for the hybrid flow, we also support sharing a hardware core across Pthreads. The *same* `Tcl` parameter used in the hybrid flow, `set_accelerator_function "function_name" --numAccels max_number_of_instances`, can be used in the hardware-only flow to constrain the number of parallel hardware instances created for a Pthread function. With this parameter, the system generator only creates as many hardware instances as given by `max_number_of_instances`, and its thread ID counter resets every time it reaches the maximum value, which forces any additional threads to *re-use* existing hardware instances. Again, the function/thread IDs are stored into the thread variable to keep track of which thread maps to which hardware instance. The main difference in the hardware-only flow is that, since the memory-mapped interconnect does *not* exist, and wrapper functions are no longer generated, the logic necessary to wait for a hardware instance to become available needs to be *embedded* into the hardware itself. This is done by changing the FSM logic so that, before invoking a parallel module that is *shared*, the FSM checks its *finish* signal first (similar to how the calling wrapper in the hybrid case first polls on a shared accelerator to check if it is available for use). If the parallel instance is already being used, the FSM remains stalled, and once the instance is available for use, the caller invokes the instance, and the FSM continues to execute.

5.3 Experimental Study

In this section, we study the performance, area, and energy-efficiency of *three* different hardware-only architectures, where we use the same architecture notations used in Section 4.5. Arch. 1 denotes a *sequential* hardware-only system compiled from single-threaded software. Arch. 2 denotes a *parallel* hardware-only system with *three* concurrently operating modules, compiled from a multi-threaded program with Pthreads. Lastly, Arch. 3p denotes a *multi-threaded and pipelined* hardware-only system, where each concurrent hardware core is also pipelined within. We use the same set of benchmarks described in Section 4.5.1, where there were four out of six pipelinable benchmarks. We again use a `p` subset for each architecture that includes the results for only the pipelinable benchmarks.

5.3.1 Results

Tables 5.1 and 5.2 show the geometric mean results over all the benchmarks for the sequential (Arch. 1/1p), multi-threaded (Arch. 2/2p), and multi-threaded and pipelined (Arch. 3p) architectures of the hardware-only systems. The complete circuit-by-circuit results are presented in Appendix D. We report four metrics, in terms of performance, area, power, and efficiency, and the last three lines of both tables show the relative ratios, comparing Arch. 2 to Arch. 1, Arch. 3p to Arch 1p, and Arch. 3p to Arch. 2p.

Looking at the ratios between the architectures in Table 5.1, we see significant improvements in performance for Arch. 2 and especially for Arch. 3p, over Arch. 1. With three parallel cores, Arch. 2 shows 2.81× and 2.6× speedups in clock cycles and wall-clock time respectively, with area increasing by 2.70× for ALMs, 2.84× for registers, 2.50× for DSPs, and 1.77× for M10Ks. As was shown in previous chapters, memory bandwidth is a crucial factor for performance. Thus for Arch. 3p, we partitioned the input/output memories in C and also replicated constant memories across threads with LegUp (described in Section 6.5.3) to minimize memory stalls. With memory partitioning/replication and *three* pipelined cores operating concurrently, Arch. 3p shows vast speedups over Arch. 1p, where clock cycles is improved by 125.5×, and wall-clock time is improved by 122.2×. Fmax stays relatively constant compared to Arch. 1p, with the area increasing by 2.28× to 4.1× for the different types of FPGA blocks. The ratio between Arch. 3p and Arch. 2p exhibits the performance benefits of combining pipelining with memory partitioning/replication. Compared to having three non-pipelined cores, Arch. 3p shows 45.56× speedup in wall-clock time, although pipelining also increases register usage by 51.3%, and memory partitioning/replication increases M10K usage by 2.52×.

Table 5.2 shows the power and efficiency (energy consumption and area-delay product) results of the three architectures. The area-delay products were calculated in the same method described in Section 4.5.2. With three parallel cores (Arch. 2), dynamic power consumption increases by 2.3×, although total power consumption only increases by 4.2%, when compared to Arch. 1. In terms of energy-efficiency, we see an improvement of 2.5×, but the area-delay product basically stays the same. Comparing Arch. 3p to Arch. 1p, dynamic power consumption increases by 3.5× (total power by 7.5%), but the large improvements in performance leads to 113.7× better energy-efficiency and 2.4% area-delay product (41.7× improvement). Comparing Arch. 3p to Arch. 2p, we see a 71.2% increase in dynamic power, but also 43.5× better energy-efficiency with 2.8% area-delay product (36.1× improvement).

Table 5.1: Geometric mean performance and area results for hardware-only systems.

	Performance			Area			
Architecture	**Time (μs)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**
Arch. 1	2,064.10	342,846.46	166.10	3,931.72	7,987.11	48.86	5.83
Arch. 1p	2,669.84	481,973.16	180.53	2,880.20	5,909.52	43.12	7.03
Arch. 2	792.40	121,838.05	153.76	10,609.17	22,698.89	122.05	10.32
Arch. 2p	995.44	168,556.69	169.33	7,060.69	16,032.12	98.28	6.82
Arch. 3p	21.85	3,840.87	175.78	9,406.79	24,251.33	98.28	17.18
Arch. 2 / Arch. 1 Ratio	0.384 (2.60×)	0.355 (2.81×)	0.926	2.698	2.842	2.498	1.771
Arch. 3p / Arch. 1p Ratio	0.008 (122.19×)	0.008 (125.49×)	0.974	3.266	4.104	2.280	2.443
Arch. 3p / Arch. 2p Ratio	0.022 (45.56×)	0.023 (43.89×)	1.038	1.332	1.513	1.000	2.521

Table 5.2: Geometric mean power and efficiency results for hardware-only systems.

	Power			Efficiency	
Architecture	**Static Power**	**Dyn. Power**	**Total Power (mW)**	**Energy (μJ)**	**Area-delay Product**
Arch. 1	1,328.75	48.26	1,393.94	2,877.24	25,358.72
Arch. 1p	1,328.77	45.64	1,396.86	3,729.38	25,839.86
Arch. 2	1,329.24	111.13	1,452.28	1,150.78	25,454.88
Arch. 2p	1,329.03	93.37	1,432.48	1,425.95	22,398.69
Arch. 3p	1,329.65	159.86	1,501.21	32.80	620.43
Arch. 2 / Arch. 1 Ratio	1.000	2.303	1.042	0.400 (2.50×)	1.00 (1.00×)
Arch. 3p / Arch. 1p Ratio	1.001	3.503	1.075	0.009 (113.69×)	0.024 (41.65×)
Arch. 3p / Arch. 2p Ratio	1.000	1.712	1.048	0.023 (43.47×)	0.028 (36.10×)

5.4 Summary

In this chapter, we presented a hardware-only flow (processor-less) which allows one to compile a multi-threaded software program with Pthreads and OpenMP to a purely hardware platform. By removing the processor and the Qsys system builder requirements, we can create a completely generic parallel hardware system. All of the generation of software and hardware needed to handle threads are done behind the scenes, without requiring any interaction from the user. As in the hybrid flow, we also provide a method to *share* a hardware core across multiple threads. Results show that the multi-threaded and pipelined hardware with memory partitioning and replication can achieve more than two orders of magnitude better performance and energy-efficiency compared to sequential hardware.

This work is to be submitted to IEEE Transactions on Very Large Scale Integration Systems (TVLSI), along with the work presented in Chapter 4.

Chapter 6

Resource and Memory Management Techniques for HLS of Parallel Hardware

6.1 Introduction

In the previous chapters, we described our *system-level* implementations which allow one to compile a multi-threaded software program to either a hardware-only system, or to a processor-accelerator hybrid system. In this chapter, we discuss *architectural-level* optimizations which pertain to generating more efficient hardware. In particular, we describe *resource* and *memory* management techniques for improving the performance and area of parallel hardware generated by HLS. One direction investigated pertains to how modules in the HLS-generated parallel hardware should connect to one another: 1) with a *nested* topology, or 2) with a *flat* topology. In the nested topology, hardware modules are created in a hierarchical manner – modules are instantiated *inside* the modules that use them. Conversely, the flat topology instantiates all hardware modules at the *same* level of hierarchy. For the flat topology, we describe a system generator that automatically generates the required interconnect between all hardware modules, as well as flexibly shares *or* replicates functions, functional units, and memories. In the presence of parallel threads (parallel hardware), the system generator also automatically inserts arbitration and deadlock-prevention circuitry.

We also explore methods to reduce memory contention among hardware units that operate in parallel

by investigating three different memory architectures which use: 1) a *global* memory controller, 2) *local* memories, and 3) *shared-local* memories. Local and shared-local memories are dedicated RAM blocks for a single or a set of hardware modules, and help to increase memory bandwidth by allowing concurrent memory accesses. We use a points-to analysis to determine, at *compile* time, which array a pointer can reference, and use this information to designate memories as global, local, or shared-local. We also describe how the points-to analysis can be used to improve our support for mutexes and barriers. Lastly, we consider memory replication to localize memories in hardware modules, and convert small memories to registers to further improve performance and memory usage.

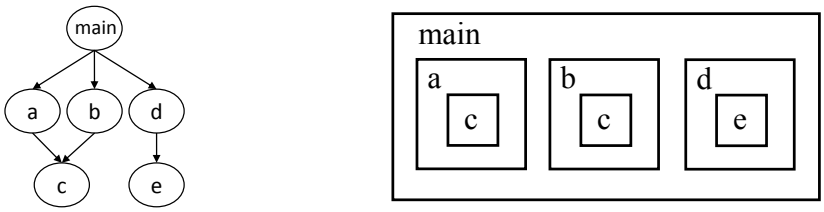
6.2 Background

A number of prior works have focused on the architecture of HLS-generated circuits. In [69] and [35], the authors investigated implementing efficient pipelined hardware for multi-threaded kernels. The works in [31, 29], and [68] implement parallel hardware architectures with Pthreads and OpenMP using HLS. Altera's OpenCL compiler [100] also creates deeply pipelined hardware from massively parallel OpenCL kernels. Vivado HLS [138] provides knobs for pipelining both entire functions and loops. Implementing efficient loop pipelining hardware was investigated in [24, 153, 9, 55]. Such prior works pertain mainly to the *micro*-architecture of the data-path produced by HLS. Conversely, our work considers the *macro*-architecture, specifically, how functions, memories, and functional units can be connected together within a larger surrounding circuit and how they can be shared *or* replicated between parallel modules. In fact, the techniques proposed in this chapter are compatible with prior work on synthesis of pipelined hardware modules.

In terms of resource sharing, [40] discusses sharing across *call hierarchies* using the flat topology, but not in the context of parallel-operating hardware. Sharing resources were investigated in [19, 34], but only within a module. In [44], resources are shared between loops for optimizing throughput. In contrast, our work investigates resource sharing between parallel threads. To our knowledge, no other work has analyzed the impact of circuit topology together with function, memory, functional unit sharing and replication on the area and speed of parallel HLS-generated hardware.

6.3 Circuit Topology

Unlike software compilers, which target fixed processor architectures, HLS offers the freedom to evaluate and choose the best architecture for a specific application. The circuit topology considered here is a



(a) Call graph (b) Circuit Architecture

Figure 6.1: A call graph and its circuit architecture using nested topology.

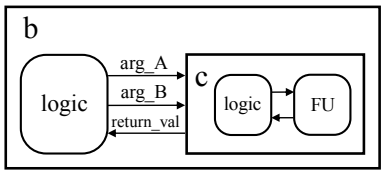


Figure 6.2: Internal architectures of module `b` and `c`.

dimension along which such a design's architecture may be optimized. In this section, we describe two circuit topologies, the nested topology and the flat topology. In the nested topology, each hardware module is self-contained, meaning that, aside from data in memories, it does not rely on other hardware modules outside of its own module hierarchy. Figure 6.1a shows a call graph of a program, and Figure 6.1b shows the corresponding nested circuit architecture. As depicted, the architecture is hierarchical, with `main` being the top-level module, and its hardware modules recursively instantiated inside. Note that due to the hierarchical structure, there are *two* copies of module `c`. This is the default architecture used by Vivado HLS, and was also used by LegUp before this work. The hierarchical approach precludes the sharing of modules by other modules, potentially leading to higher area consumption.

Figure 6.2 shows the internal architectures of modules `b` and `c`. Observe that module `c` has a functional unit (FU) instantiated within. In the nested topology, the arguments of a function become input ports of its hardware module, and the return value (if any) becomes an output port of the module.

The advantage of the nested topology lies in its simplicity: connectivity between modules is entirely local. Any modules used by another module are directly instantiated within the module itself and connected inside. The hardware module interface is aligned to that of software (i.e. arguments *passed in* become input ports, any data *returned* become output ports). Each hardware module is also self-contained, so if one needs to re-use a particular module in a new hardware system, simply instantiating the particular module is sufficient, as this will also bring with it all of its sub-modules.

However, the nested architecture is inefficient in a number of ways. In the input software, if a function is called by multiple different functions, the nested topology replicates hardware, as was shown in Figure 6.1b. Likewise, since functional units are instantiated within hardware modules, sharing is also

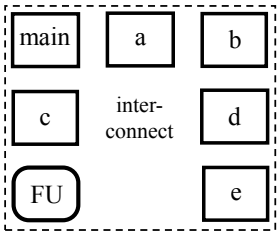


Figure 6.3: Circuit in Figure 6.1 with flat circuit topology.

precluded for them. Given that dividers or floating-point units are generally large, this can considerably increase overall circuit area, particularly if such functional units are used in many different modules. With the architecture shown in Figure 6.1b, two instances of the functional unit are also created, since there are two copies of module `c` in hardware. Not that this occurs even when the modules are running sequentially (meaning that only one of the functional units will be utilized at a time), leading to an unnecessary increase in circuit area.

Figure 6.3 shows the circuit architecture using the flat topology, for the same circuit shown in Figure 6.1. In the flat architecture, all modules reside at the same level of hierarchy, which enables sharing of functions and functional units. As shown, only one instance of module `c` is created, which is shared by modules `a` and `b`. The system generator, described in the next section, automatically creates the interconnect to directly connect or share functions, functional units, and memories in both sequential and parallel-execution modes.

6.4 System Generator

For the flat circuit topology, we built a system generator to automatically connect all communicating hardware components in the system. The system generator handles both sequential and parallel execution, generating a different interconnect optimized for each case. It is similar to other system generators, such as Qsys, except that it is completely integrated into the HLS framework. As such, it requires no additional input from the user; whereas, with Qsys, the user must specify which components connect to which other components, through which type of interface.

Our system generator automatically creates the interconnect by traversing the function call graph of the input program. All connections are point-to-point, allowing concurrent independent transfers. Each connection is composed of a pair of interfaces, a *master* interface and a *slave* interface. A master interface initiates a transfer, and a slave interface responds to the transfer. For instance, a function accesses a memory through its master interface (composed of address, enable/write enable, read/write

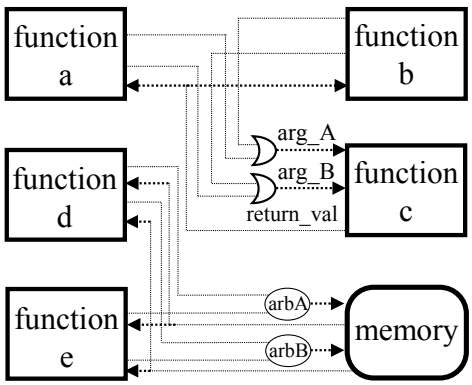


Figure 6.4: An example interconnect generated by system generator.

data ports), and the memory responds through its corresponding slave interface. A single module can have multiple interfaces, allowing concurrent transfers (i.e. a function can have multiple interfaces to access multiple memories simultaneously, as well as interfaces to call other functions, and access functional units). The widths of the interfaces depend on the types of data that the interfaces are used to carry. For example, for an argument interface of a function, its width is sized according to the bitwidth of the argument data type. Similarly, for a memory interface, read/write data signals are sized according to the width of data that the memory is holding. Control signals, such as enable and write enable, are 1-bit wide. When multiple master interfaces are connected to a single slave interface, with the master components executing sequentially, the system generator creates a simple `OR` gate to handle contention efficiently. The `OR` suffices in this case, as long as inactive masters output logic-`0` to their corresponding `OR` inputs. When multiple *parallel* masters are connected to a slave, a round-robin arbiter is automatically created. This differs from Altera's Qsys, which creates a round-robin arbiter regardless of whether the components are executing concurrently or sequentially, negatively impacting area and Fmax. An example interconnect generated by the system generator is shown in Figure 6.4. In this case, function `a` and `b` execute sequentially and share function `c`, and functions `d` and `e` run in parallel and share a memory. Each memory is dual-ported, so we create a separate arbiter for each port to maximize memory bandwidth.

The system generator is also responsible for selectively sharing or replicating common hardware modules, based on a user's performance vs. area requirements. If a function is parallelized with threads, then the system generator, by default, creates as many hardware instances of the function as the number of threads in the input program (unless constrained via the `Tcl` command discussed in Section 5.2.1). If the threaded function has descendant functions, it also replicates the descendant functions in hardware to maximize throughput. For example, Figure 6.5 shows the circuit architecture where `main` forks two

(a) With functional unit sharing.

(b) Without functional unit sharing.

Figure 6.5: Parallel hardware with/without functional unit sharing.

threads to execute function `d`, which has a descendant function `e`[1]. Sharing (Figure 6.5a) vs. replication (Figure 6.5b) of functional units or memories is controlled by a `Tcl` parameter. In the case of replication, the component is instantiated *inside* the module which uses it (Figure 6.5b), creating a dedicated component for that module. Replication of memories is further discussed in Section 6.5.

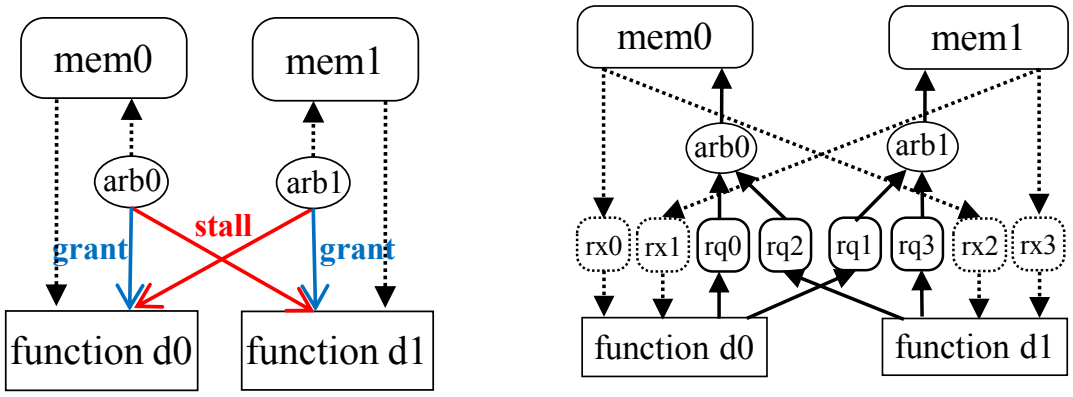
6.4.1 Automatic Deadlock Prevention

As shown in Figure 6.4, arbiters are generated to handle concurrent accesses by multiple masters to a shared resource. However, when multiple masters request access to multiple common slaves at the same time, a deadlock can occur. This is illustrated in Figure 6.6a, where functions `d0` and `d1` request access to *both* `mem0` and `mem1` in the same clock cycle (with each function have a dedicated memory port to each memory). In the example, `arb0` grants access to function `d0`, and `arb1` grants access to function `d1`. Both functions are not able to continue as they are both waiting to receive a grant for the "other" memory – a deadlock[2]. To prevent deadlocks, our system generator automatically inserts deadlock prevention modules where necessary.

There are two parts to the deadlock prevention module, the *request* module and the *data receiver* module, denoted as `rq` and `rx` in Figure 6.6b. This set of deadlock prevention modules is created for each dedicated memory interface for each function (`rq/rx` 0 and 1 are for function `d0`, and `rq/rx` 2 and 3 are for function `d1` in Figure 6.6b). The request module handles the request for a master interface to its connecting slave arbiter. It ensures that once a master interface has received a grant from its arbiter (which allows the master to access the slave in the same cycle, and the slave, in this case memory,

[1] If the descendant function is only called *once*, it may be beneficial to *inline* the function, to allow additional compiler optimizations. This can be controlled by the user through a `Tcl` parameter.

[2] A function's FSM remains stalled as long as its stall signal is asserted. Its stall signal is a logical `OR` of all stalls for its master interfaces from their arbiters.



(a) A deadlock situation.

(b) With deadlock prevention modules.

Figure 6.6: Circuit architecture with/without deadlock prevention modules.

responds by returning its data after its latency amount of cycles), it does not make the same request again (requests are *state* dependent, thus being stalled in the same state would keep the request signal high continuously).

We re-illustrate the previously described deadlock scenario, this time with the deadlock prevention modules. In the first clock cycle, both functions `d0` and `d1` request to access both `mem0` and `mem1`, and `arb0` grants access to function `d0`, with `arb1` granting access to function `d1`. When an access is granted, the corresponding slave is accessed in the same cycle, hence in this case, function `d0` accesses `mem0` and function `d1` accesses `mem1` in the first clock cycle. In this example, we assume that memories have a one cycle access latency. In the next clock cycle, data is returned from `mem0` to function `d0`, with the data also being stored in `rx0`. Similarly, data is returned from `mem1` to function `d1`, and is stored in `rx3`. In the same clock cycle (i.e., second), the request modules lower the requests signals that have been granted. Hence, `rq0` lowers the request from function `d0` to `mem0`, and `rq3` lowers the request from function `d1` to `mem1`. Now only the requests from function `d0` to `mem1` and from function `d1` to `mem0` remain for arbitration, which are independent to each other, and thus a deadlock does not occur. Hence the memories are again accessed, and in the subsequent cycle (i.e., third) the data is returned from `mem1` to function `d0`, and from `mem0` to function `d1`.

The circuit for the request module (i.e., `rq`) is simple, it contains a register which: 1) stores a `0` when the grant is given for a master interface, but the stall is still asserted for the function (due to stalls for its other master interfaces), and 2) otherwise stores a `1`. This register output is `AND`'ed with the request from the master interface, preventing the request signal to the arbiter from being kept high, once its grant is received. In the example above, `rq0` is stored a `0` once the grant from `arb0` is received, hence `AND`'ing this with the request from function `d0` lowers the request to `arb0`. The request modules

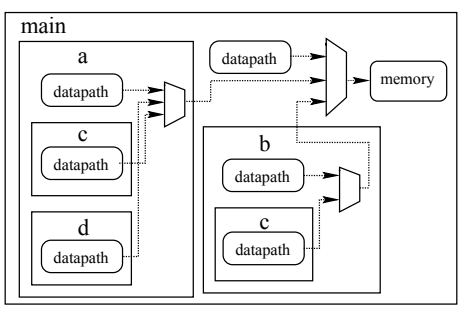
essentially ensure that after all of the requests have been granted once, the function is able to continue to execute, and it can work for any number of requests.

Note that if concurrent requests from the same function reach their corresponding memories at different clock cycles due to contention, the data from the memories also returns at different cycles. In the example above, the data from `mem0` to function `d0`, and the data from `mem1` to function `d1` returned in the second clock cycle (the memories were successfully accessed in the first clock cycle, and we assumed that memories have a one cycle access latency), but the data from `mem1` to function `d0` and the data from `mem0` to function `d1` returned in the third clock cycle. This results in incorrect execution, since the functions expected both data items at the same time (i.e., were requested in the same FSM state).

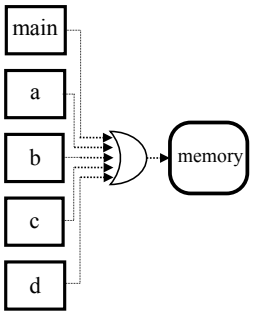
The purpose of the data receiver module is to ensure that the data returned to the master is received correctly, by buffering the data, as appropriate. If there were no stalls, the data receiver passes through the returned data directly, otherwise it returns the buffered data. Hence in the example, `rx0` and `rx3` return their buffered data, whereas `rx1` and `rx2` pass the data through directly. The circuit for the data receiver is also straightforward: It contains a shift register, with its size equal to the latency of the slave, and it shifts in `1` into the LSB when the grant is given from the arbiter. When the MSB of the shift register contains a `1`, it indicates that the slave is returning its data in that clock cycle. At this time, the data receiver stores the returning data in its internal registers, but also passes it through directly to the master (if the master was not stalled, the data is needed in that clock cycle). In subsequent clock cycles, it returns the stored data, until the data is overwritten by new data. The data receiver is parametrized to allow connecting to slaves with any latency and data width. For instance, it can be connected to a divider, which has a latency equal to its bitwidth, as well as a multiplier or a memory, which have shorter latencies. It can also work for variable latency operations (i.e. off-chip memory access), by enabling the shift register only when the `valid` signal[3] from the variable latency operation is received. Although there are many prior works on implementing mechanisms to avoid deadlocks (including [43, 42, 148]), our method of using the set of *request* and *data receiver* modules provide a simple and modular approach, which do not require user code changes in software or modifications to the internal architectures of hardware modules.

In summary, our system generator creates an efficient interconnect completely automatically, benefiting from the integration within the HLS framework and access to the program's call graph in the compiler. It handles arbitration for sequential and parallel execution modes, allows flexible sharing or replication of functions and functional units, and inserts dead-lock prevention modules when necessary.

[3]IP cores for variable latency operations, such as off-chip memory, have a `valid` signal to indicate that the data being returned is *valid* in that cycle.



(a) Sharing a memory in nested topology.



(b) Sharing a memory in flat topology.

Figure 6.7: Memory sharing in nested/flat topology.

We believe these are unique features of our work.

6.4.2 Advantages of Flat Topology with the System Generator

The flat circuit topology enables the efficient sharing of modules. In the nested topology, to share a memory between two modules, the accessing modules need connectivity to the module where the memory is instantiated. If an accessing module is deep in the function hierarchy, memory ports must be created and signals must be passed across all intermediate modules, as shown in Figure 6.7a (left side of figure). In Vivado HLS (and with LegUp prior to this work), multiplexers are created at each level of hierarchy (in the sequential case; i.e. when modules sharing the memory are not operating concurrently). The size and the depth of the multiplexers grows linearly with the number of functions that access the memory and the depth of the call hierarchy. Modules instantiated multiple times due to the nested topology also increase the multiplexer size unnecessarily, as shown in Figure 6.7a for function `c`. This leads to poor performance and area. Functions can be inlined to remove some multiplexers, but this may also increase circuit area. In the flat topology, all shared modules are instantiated at the same hierarchy level, and we *zero* out all memory signals when they are not being used, so that our system generator can simply connect them through an `OR` gate (in the sequential case), as shown in Figure 6.7b. In the parallel case, `OR` gates are replaced with arbiters, as is done with functional unit sharing described earlier. In Vivado HLS, we were not able to share functional units across different functions[4]. As for Altera's OpenCL Compiler, it simply *inlines* all descendants functions of a kernel, eliminating the option to share functions or functional units.

[4]The Vivado HLS user manual shows that the `config_bind` configuration with the `min_op` option can be used to share functional units globally in a program. However, when we tried to share a divider in two separate functions, Vivado HLS created a divider inside each of the functions.

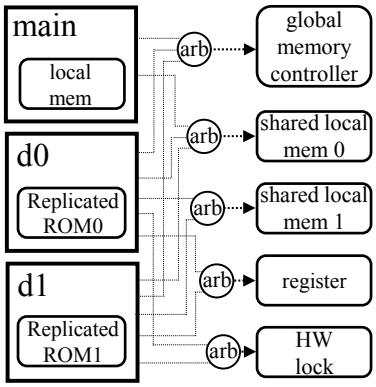


Figure 6.8: Circuit using the different types of memories.

6.5 Memory Architectures

Memory architecture can play a critical role in any hardware system, and memory bandwidth is often the limiting factor for performance. A key architectural feature of FPGAs is the availability of on-chip block RAMs which provide low-latency memory accesses. Block RAMs are distributed throughout the chip, and can be accessed in parallel. There are also an abundant number of registers, which can also be used to store data. We therefore examine the different ways we can make use of the block RAMs and registers to reduce memory contention, in the presence of parallel operating hardware. We first consider three different memory architectures which use: 1) a *global* memory controller, 2) *local* memories, and lastly 3) *shared-local* memories. We use points-to analysis to designate arrays for implementation in global, local, and shared-local memories. Shared-local memories are shared by multiple modules in a system, hence require logic to handle contention, which may increase circuit area and latency. To mitigate this, we investigate replicating constant (read-only) shared-local memories across parallel modules to eliminate the overhead of arbitration logic. We also show how points-to analysis can be used to efficiently implement locks and barriers for thread synchronization. Lastly, we consider converting small memories to registers to lower memory usage and latency. An example circuit containing all of these features is shown in Figure 6.8. In the figure, the functions `main`, `d0` and `d1` execute in parallel, and they share the global memory controller, shared-local memories 0 and 1, a register module, as well as a hardware lock module. For simplicity, only one port of memory is shown and the deadlock prevention modules are also not shown. Each component is accessed through a set of dedicated ports, allowing concurrent accesses.

6.5.1 Points-to Analysis

To intelligently designate arrays for implementation in *global*, *local*, *shared-local* memories, we use a points-to analysis, which determines which memory locations a pointer can reference. There have been

many points-to analysis algorithms developed by the compiler community. Andersen [5] described the most accurate of these approaches, which formulates the points-to analysis problem as a set of inclusion constraints for each program variable, which are then solved. Steensgaard [67] presented a less accurate points-to analysis, which used a set of type constraints modeling program memory locations that can be solved in linear-time. In this work, we use Andersen's points-to analysis [5] which was implemented in the LLVM compiler by [89]. For each memory access in a program, the points-to analysis returns a set, which contains all the arrays the address can possibly point to. If it returns a set of size 1, it indicates that the address can only point to a single array, which will be located in one logical hardware RAM by the HLS tool. Otherwise, the address points to multiple arrays, and it needs to be resolved at run-time. Points-to analysis algorithms have varying levels of accuracy and may be overly conservative, but for programs without dynamic memory, recursion, and function pointers, *most* pointers are can be resolved at compile time [66].

6.5.2 Global Memory Controller

The purpose of the global memory controller is to automatically resolve pointer ambiguity at run-time. The global memory controller is only created if there are pointer references that cannot be resolved at compile-time with the points-to analysis (i.e. pointers pointing to multiple arrays). Its architecture is shown in Figure 6.9. For clarity, some of the signals are combined together in the figure. Even though the figure depicts a single-ported memory, all memories are dual-ported by default. The memory controller steers memory accesses to the correct RAM, by using a tag, which is assigned to each array in the program by the HLS tool. A tag is set to be the top 9-bits (which can address up to 512 global memories; this bitwidth is easily configurable) of an incoming address, and it is used to determine which memory block to enable, with all other memory blocks disabled. The same tag is used to select the correct output data between all memory blocks. The lower bits of the address are used to get the offset into the RAM. Each block RAM has latency of one cycle, and the output of the multiplexer is also registered (to improve Fmax), making a memory access two cycles by default.

The advantage of this memory controller is that it can support generic pointers and resolves pointer references at run-time (by using the tags). This permits the support of a wider range of input programs, including those which may not be amenable to pointer analysis. However, there are a number of drawbacks to this memory architecture, in terms of its performance and area. First, any memories in the memory controller must be accessed sequentially. This is because the memory being accessed is determined at run-time, and hence needs to be accessed through a shared set of memory ports. This

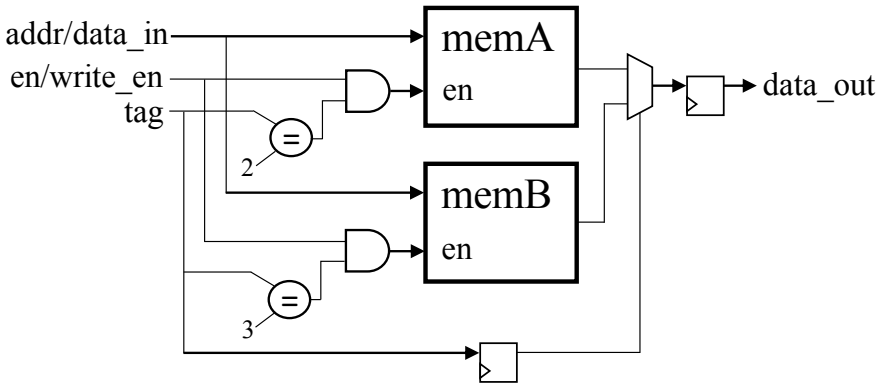


Figure 6.9: Global memory controller architecture.

limits memory accesses to two per cycle (owing to the underlying dual-ported memories). In addition, the output multiplexer of the memory controller grows linearly in size with the number of RAMs. Increasing the number of global memories can hurt both the Fmax of the circuit as well as its area. The performance and area deterioration becomes worse when using the global memory controller with the nested circuit topology, as was shown in Figure 6.7a, due to the large amount of multiplexing required to connect to the memory controller. Despite this, the memory controller ensures that the circuit can handle all types of pointer accesses, and may be needed for some programs.

6.5.3 Local and Shared-local Memories

Using the points-to analysis, we can designate arrays in the program to implement in *local* and *shared-local* memories. An array is designated into a local or a shared-local memory if the points-to analysis can determine that it is never referenced by a pointer that points to multiple arrays. If such an array is only accessed by a single function, it is designated as local memory. Otherwise, if it is referenced in multiple functions, it becomes a shared-local memory. Each local and shared-local memory is accessed through a dedicated set of memory ports, allowing concurrent memory accesses among the memories. A local memory is instantiated *inside* the module which accesses it, hence connected directly, and a shared-local memory is instantiated *outside* the module, with an arbitration unit created to handle memory contention between its users. Because local and shared local memories have limited numbers of accessors, the memory latency is set to *one* clock cycle. We have empirically found that this improves the overall performance (the latency can also be easily configurable by the user). In Vivado HLS, memory access latencies are set to *two* cycles for all memories. Within local and shared-local memories, we perform a number of optimizations to improve performance. As described below, we replicate read-only memories, and convert memories to registers. We can also implement synchronization constructs efficiently with

the points-to analysis.

Constant Array Replication

Constant arrays are implemented in read-only memories (ROMs), and as such, can safely be replicated in each accessing module. Although replication increases memory usage, for memory-intensive applications, where many threads contend for the same memories, it can be beneficial to create a dedicated memory for each thread. By localizing the memory to each thread, we can improve performance by reducing stalls due to contention, and also decrease area by removing the arbitration logic. Enabling this feature is controlled through a `Tcl` parameter in our work.

Memory to Register Conversion

By default, LegUp HLS implements all arrays in block RAMs. However, for small arrays with few elements, implementing them in registers may be beneficial to reduce memory usage. In LLVM, global variables are treated the same way as global arrays. Therefore, a naive implementation would use an entire RAM to store a single global variable. In LegUp, we detect when an array has a single element, or if it is a global variable, and we store it in a register module. LLVM has an existing compiler pass called *mem2reg*, which promotes memory to registers [78], however we found that the pass works only in a very limited number of cases, necessitating this optimization. If the converted register is used by a single module, we create the register inside the module, or if it is used by multiple modules, we create a register module outside, with the system generator connecting it to all of its accessors. Similar to how we had set the memory latency to `1` clock cycle for local and shared-local memories, we can actually set the memory load latency for these registers to `0` clock cycles, further reducing memory latency. In addition, since the load latency is `0`, the register outputs are directly connected to the accessing modules, and do not need to connect through the data receivers, reducing area.

As described, we can flexibly adjust the memory latencies of the different types of memories to optimize performance, with global memories having `2` cycles, local/shared-local memories having `1` cycle, and memories converted to registers having `0` cycle.

Handling Synchronization

With the points-to analysis we also improve our support for locks and barriers. Prior to this work, all locks and barriers needed to be accessed through a set of shared memory ports, which were also shared with memories. This not only limited concurrent accesses to different locks/barriers, but also limited memory bandwidth. With the points-to analysis, we can access multiple locks and barriers concurrently,

and independently of other memories. To do this, we treat the synchronization variables as memory variables, and classify them as a shared-local memory (since they are accessed by multiple modules). Points-to analysis returns a list of functions which use the synchronization variable. Then, we create dedicated ports from each function to each lock/barrier variable. When multiple locks are used in a program, they each have dedicated ports and can be accessed concurrently. Again, the system generator automatically creates arbiters and dead-lock prevention modules for each lock/barrier variable.

The actual operation of the locks and barriers remain mostly the same as what was described in Section 3.4.1. With each lock/barrier variable replaced with a hardware lock/barrier module, the communication with the modules is achieved through memory loads and stores.

6.6 Experimental Study

In this section, we study the impact of the different circuit and memory architectures on the performance and area of parallel hardware. We consider in total 8 different architectures:

1. Nested topology with a global memory controller.
2. Flat topology with a global memory controller.
3. Architecture 2 *plus* divider sharing across threads.
4. Architecture 3 *plus* multiplier sharing across threads.
5. Architecture 4 *plus* local, shared-local memories (all memories have latency of `2` cycles).
6. Architecture 5 *plus* memory to register conversion (latency of local/shared-local memories set to `1` cycle, register module has latency of `0` cycle).
7. Architecture 6 *plus* constant memory replication across threads.
8. Architecture 7 *minus* multiplier sharing across threads.

With each successive architecture, we enable/disable a feature, allowing us to analyze its impact in isolation. Architectures 1, 2, 3, and 4 have no local or shared-local memories. A global memory controller can be used in any of the eight architectures, but is only created for benchmarks which require it. Comparing architectures 1 and 2 reflects the utility of the flat architecture vs. the nested architecture. With architectures 3 and 4, we can examine the impact of sharing functional units. Architecture 5 shows the impact of having dedicated memories with local and shared-local memories, and Architecture 6 illustrates the effect of reducing the memory access latencies. With, Architecture 7, we can investigate the effect of memory replication on memory contention, and lastly, with Architecture 8, we analyze the area/performance impact of disabling multiplier sharing across threads. For the rest of this chapter, each architecture is referred to by its number (i.e. Arch. 1 = nested with global memory controller). We

use the hardware-only flow in this work, where the entire program is compiled to hardware.

6.6.1 Benchmarks

We use a total of 15 benchmarks, each of which is parallelized with Pthreads. Each benchmark is described below.

- Alphablend: Alphabends two images.
- Barrier: An accumulation benchmark which uses a barrier.
- Blackscholes: Options pricing via a Monte Carlo approach.
- Box Filter: A convolution filter commonly used in image processing. C implementation of the filter adopted from [2].
- DF: Adopted from the CHStone [37], it performs double-precision floating-point operations using 64-bit integers.
- Division: Integer division of two arrays.
- Dot Product: Dot product of two arrays.
- Hash: Four different hashing algorithms, with the number of collisions compared at the output.
- Histogram: Accumulates integers into 5 equally-sized bins.
- Line of Sight: uses the Bresenhams line algorithm [8] to determine whether each pixel is visible from the source.
- Mandelbrot: An iterative mathematical benchmark which generates a fractal image.
- Matrix Multiply: matrix multiplication of two arrays.
- MCML: Adopted from the Oregon Medical Laser Centre [60], it simulates light propagation from a point source in an infinite medium with isotropic scattering.
- Mutex: An accumulation benchmark which uses a lock.
- Vector Add: Performs vector addition of two arrays.

Other than the *Mutex* benchmark, *Barrier* and *MCML* benchmarks also use Pthread locks. Each benchmark was synthesized, placed and routed into the Altera Stratix V FPGA (5SGSMD8K1F40C2) with Quartus 15.0.

Time	Execution Cycles	Fmax	Logic Util.	DSPs	M20Ks
262.86	45562.53	173.34	3727.72	11.03	39.38

Table 6.1: Geomean baseline results (Arch. 1).

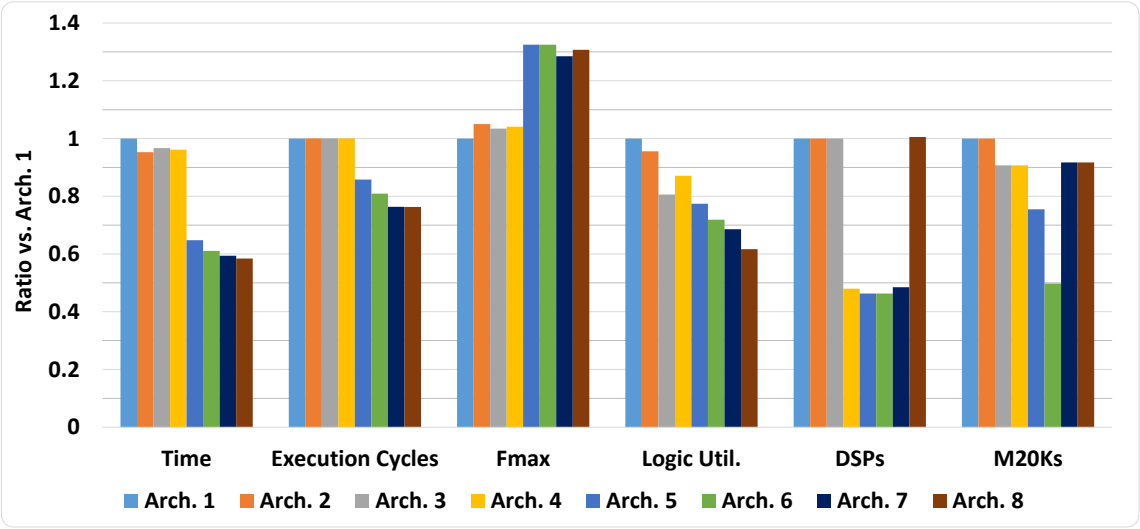


Figure 6.10: Geomean performance and area results for each architecture.

6.6.2 Results

Table 6.1 shows the geometric mean results across all benchmarks for Arch. 1, and Figure 6.10 shows the geometric mean results for each architecture relative to Arch. 1. The complete circuit-by-circuit results are presented in Appendix E. There are three performance metrics (total wall-clock time (# cycles $\times$ clock period), total number of clock cycles, and Fmax of the circuit) and three area metrics (logic utilization, DSP blocks, and M20K blocks). M20Ks are Altera's on-chip RAMs that can each hold up to 20 Kbits of data.

The general trend is that, as we progress from Arch. 1, towards the Arch. 8, results improve in terms of both performance and logic utilization. Comparing Arch. 1 and 2, both logic utilization and Fmax improve slightly, owing to the previously described efficiency of the flat topology vs. the nested topology. With the Fmax improvement, geomean wall-clock time improves by 4.7%. With divider sharing in Arch. 3, logic utilization and M20K usage drop. Altera's divider cores use M20Ks within, thus memories are also saved in sharing dividers. There is virtually no impact on execution cycles (0.1% increase). This is because in our system, threads are started in a staggered manner (i.e. one after another), and dividers are pipelined (to the depth equal to the operand's bitwidth). Thus, when sharing dividers across threads, stalls caused by divider contention among threads are minimal. When sharing multipliers in Arch. 4, DSP usage drops as expected, and execution cycles are again affected minimally. Logic utilization does increase, however, due to multiplexers required on the inputs of the multipliers. In Arch. 3, logic

utilization decreased when sharing dividers only, since the decrease from sharing dividers exceeds the increase from the added input multiplexers. Performance significantly improves in Arch. 5, owing to the local and shared-local memories. Compared to Arch. 4, execution cycles and total execution time improve by 14.3% and 32.6%, respectively. The local/shared-local memories also shrink the expensive multiplexers in the global memory controller (as described in Section 6.5.2), improving logic utilization and Fmax by 11.1% and 27.3%, respectively. M20K usage drops with local memories, since Quartus is able to perform more optimizations, such as reducing a RAM block to registers, when memories directly connected to the data-path. When RAMs are created inside the global memory controller, behind multiplexers, Quartus is not able to perform such optimizations. However, Quartus cannot automatically convert *all* small memories to registers, which we handle in Arch. 6.

With memory-to-register conversion, M20K usage decreases by 34.1% from Arch. 5, and logic utilization decreases by 7.1%. We also reduce the memory-load latencies to 1 clock cycle for local/shared-local memories and to 0 clock cycles for memories converted to registers. This improves both execution cycles and wall-clock time by an additional 5.7%, compared to Arch. 5. In Arch. 7, we localize ROMs to each thread through replication to reduce memory contention between threads. With this, execution cycles and total execution time improve by 5.7% and 3.6% respectively, relative to Arch. 6. M20K usage increases, however, by 1.97× due to replication. In Arch. 8, we disable multiplier sharing across threads, as the input multiplexers can increase circuit area and lower Fmax. The execution cycles improves minimally, and the logic utilization improves by 10%, compared to Arch. 7. As expected, DSP usage also increases significantly by 2×. Overall, Arch. 8 yields the best performance and logic utilization results out of all architectures, with an improvement of 41.6% (wall-clock time) and 38.3% (logic utililization), compared to Arch. 1.

We also calculate the area-delay product, again using the data from [64]. Figure 6.11 shows the ratios of the geometric mean area-delay product for each architecture when compared to Arch. 1. As seen in the figure, the area-delay product generally improves from Arch. 1 up to Arch. 6, at which point it become worse in Arch. 7. This is because the performance generally improves up to Arch. 6, with the area also significantly reduced by sharing functional units, reducing multiplexing logic, and converting RAMs to registers. From Arch. 7, the performance continues to improve slightly, however, area is increased significantly when replicating memories in Arch. 7. Area-delay product is slightly improved in Arch. 8 (by 1.6%) from Arch. 7, as the total tile area is lower in Arch. 8 (the reduction in area by eliminating the multiplexers required to share multipliers is more than the increase in area by *not* sharing multipliers), and the performance is also better for Arch. 8. Overall, Arch. 6 shows the best area-delay product, with an improvement of 60.9% over Arch. 1.

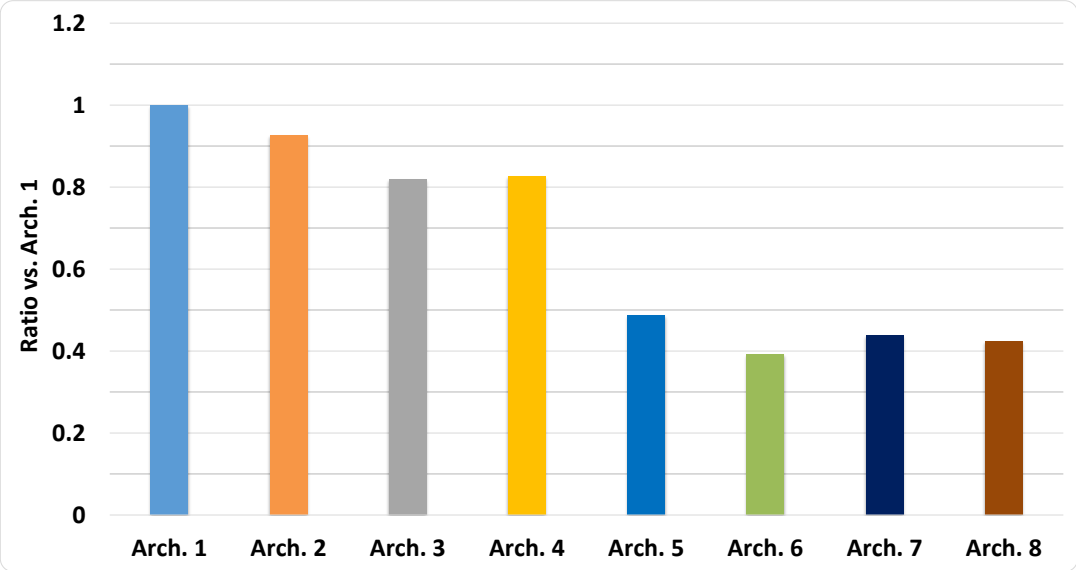


Figure 6.11: Geomean area-delay product for each architecture.

In summary, we observed that local/shared-local memories and memory-to-register conversion, together with reduced access latencies, significantly improve performance and area. Constant memory replication also helps to reduce memory contention further, but degrades area-delay product. Sharing functional units across threads had little impact on performance degradation, while producing considerable area savings. This is because Pthread modules are invoked at different clock cycles, making them slightly "out-of-step" with one another, reducing contention.

6.7 Summary

In this work, we analyzed the impact of two circuit topologies with different memory management techniques on the performance and area of parallel HLS-generated hardware. We considered the nested topology, where hardware modules are instantiated in a hierarchical manner, and the flat topology, other where modules are instantiated at the same level of hierarchy. We described a system generator, integrated within HLS, to automatically create efficient interconnect between hardware modules, with the ability to share/replicate functions, functional units, memories between functions/threads, as well as insert deadlock-prevention modules for parallel operating hardware. Three different memory architectures were also investigated: the global memory controller, and local/shared-local memories. The global memory controller handles memory accesses that are not amenable to points-to analysis. Local and shared-local memories improve memory bandwidth by providing concurrent direct memory accesses, and decrease area by reducing the multiplexing logic. Additional memory management techniques, mem-

ory replication and memory-to-register conversion, were explored to reduce memory contention between threads and also to reduce memory usage and latency.

This work has been published in the 2015 IEEE International Conference on Field-Programmable Technology (FPT) [15].

Chapter 7

Inferring Streaming Hardware with Pthreads

7.1 Introduction

In this chapter, we propose using software techniques to infer parallel *streaming* hardware in HLS. Specifically, we use Pthreads to automatically generate multiple pipelined hardware modules, which can continuously execute concurrently in a streaming fashion. In streaming hardware, there can be multiple data items that are *in flight*, which are concurrently executed at different pipeline stages of the hardware, similar to a pipelined processor. This contrasts with the work in previous chapters, where each invocation of a thread executed a threaded module once until its completion (i.e. only a single data item was in flight at a time).

In recent years, advances in compiler and HLS research have continued to improve the quality of HLS-generated hardware. For some applications, it has been shown that HLS-generated circuits can match the performance of hand-designed hardware [120]. HLS is also increasingly being used to design commercial products, with Qualcomm using Catapult C HLS [114] to tape out a chip in production [36].

Despite this, HLS tools remains a hurdle to its broad uptake, particularly to those without hardware skills. There are tasks, such as system integration, which remains a manual process for many HLS tools. System integration can be a challenging task for many software engineers, and in our work described in Chapters 3, 4, and 5, we have addressed this by providing completely automatic system generation flows. We believe another major impeding factor to making HLS easy to use is the use of vendor-specific pragmas, which are needed to drive the generation of HLS hardware. These pragmas are foreign to

software engineers, and even to hardware engineers who are not familiar with the vendor tool. Some pragmas create hardware behaviour that is different from software behaviour, which can be difficult to comprehend for users. To this end, we propose improving the usability of HLS by providing mechanisms within HLS that permit widely used software techniques to be used to control hardware behaviour.

In multi-threaded parallel software programming, a popular development pattern is the *producer-consumer* pattern, wherein concurrently operating threads continuously receive (consume) "work to do" from other threads and also generate (produce) results that are then consumed by subsequent threads. In a typical producer/consumer implementation, queues/buffers are used between the threads as staging areas for work items that have been produced but not yet consumed. We observe an analogy between the producer/consumer pattern in multi-threaded software and *streaming kernels* in hardware, i.e. hardware modules interconnected by FIFO buffers that process their inputs in a pipelined manner and deposit results into output FIFOs. Streaming hardware is popular in applications such as audio/video processing. Commercial HLS tools, such as Xilinx Vivado HLS, create streaming hardware by using vendor-specific pragmas embedded in the source. Conversely, we propose to *automatically* infer streaming hardware behaviour by synthesizing instances of the producer-consumer pattern in software, running on Pthreads, into streaming hardware. This methodology allows streaming hardware to be specified using a well-known software methodology which creates software execution behaviour that closely aligns with the hardware behaviour.

In our approach, each software thread is automatically synthesized into a streaming hardware module. FIFOs between the hardware modules are automatically instantiated, corresponding to the work-queue buffers in the producer/consumer pattern. Exploiting the spatial parallelism available on a large FPGA becomes a matter of forking multiple threads. The proposed approach brings the added benefit that the multi-threaded code can be executed in parallel fashion in both software and hardware. Debugging and visualization can be done in software – software whose parallel execution matches closely with the parallel hardware execution.

7.2 Background

There are a number of HLS tools that can generate streaming hardware. Altera's OpenCL SDK [100] automatically creates deeply pipelined hardware from OpenCL kernels, which can be connected via streaming interfaces. Vivado HLS [138] and Impulse CoDeveloper [123] drive hardware generation with the use of pragmas in the code. In Vivado HLS, an entire function can be compiled to a *pipelined* hardware module by specifying the `HLS pipeline` pragma on the function. Data can be passed into the

kernel as a stream using a Xilinx-specific type from a library, which gets turned into a FIFO in hardware. Impulse CoDeveloper can also pipeline a loop using its vendor pragma, `CO PIPELINE`. It also provides its own APIs to stream inputs/outputs to the pipeline.

We compare our work mostly to Vivado HLS, since it bears the most similarity to LegUp HLS in terms of its programming model and the input language. A code snippet is shown below for Vivado HLS.

```
1:  void func_A(hls::stream<int>& in, hls::stream<int>& temp) {
2:    #pragma HLS pipeline II=1
3:    // read from FIFO
4:    int a = in.read();
5:    // do work
6:    ...
7:    // output to FIFO
8:    temp.write(b);
9:  }
10: ...
11:
12: void top(hls::stream<int>& in, hls::stream<int>& out) {
13:   hls::stream<int> temp1, temp2;
14:   #pragma HLS dataflow
15:   func_A(in, temp1);
16:   func_B(temp1, temp2);
17:   func_C(temp2, out)
18:   ...
19: }
20:
21: int main() {
22:   // declare FIFOs
23:   hls::stream<int> in, out;
24:   ...
25:   for (i=0; i<SIZE; ++i) {
26:     // write to input FIFO
```

```
27:     in.write(in_array[i]);
28:     // invoke top-level function
29:     top(in, out);
30:     // get result from the output FIFO
31:     out_array[i] = out.read();
32:   }
33:   ...
34: }
```

This example creates a streaming circuit for the top-level function `top`, which calls three sub-functions, `func_A`, `func_B`, and `func_C` (lines 12–19). Although only the definition of `func_A` is shown (lines 1–9), each of the sub-functions are fully pipelined with an II (initiation interval) of `1` (line 2), meaning that a new input is received and a new output is produced by the circuit every clock cycle when it is in steady-state. The `HLS dataflow` pragma in the `top` function (line 14) makes the sub-functions execute concurrently and in a pipelined fashion, rather than sequentially one after another (normal software semantics). In other words, `func_A`, `func_B`, and `func_C` operate in a *dataflow* style, commencing execution as soon as their inputs are ready. There are also intermediate FIFOs (such as `temp1` and `temp2` on line 13), which connect the sub-functions together. The `main` function pushes data into the input FIFO, invokes `top`, then fetches results via the output FIFO (lines 27–31).

This methodology is simple and intuitive. However, there are a number of issues with this approach, which make the software behaviour *different* from the generated hardware behaviour[1]. In hardware, a streaming module is always running. It is not invoked a fixed number of times (`SIZE` as shown on line 25). A streaming module simply processes data whenever its input FIFO is non-empty. This differs significantly from the semantics of the software code for Vivado HLS.

A larger discrepancy arises owing to the `HLS dataflow` pragma. This tool-specific feature internally transforms sequential software to parallel hardware. Its parallel execution, however, cannot be compiled or debugged using standard software toolchains, such as `GCC` or `GDB` (the software will just execute sequentially, as it is written). This also means that any existing software needs to be re-written in this style to exploit parallelism, which increases design time. The Xilinx-specific pragmas are foreign concepts that are difficult to comprehend for software engineers, or even for hardware engineers who are not familiar with the tool. Another discrepancy stems from the FIFOs. In Vivado HLS, streams are

[1]HLS tools typically introduce instruction-level parallelism, which allows multiple instructions to be executed in the same cycle. Likewise, pipelined processors also execute multiple instructions at the same time, and out-of-order processors can also re-order instructions. The behaviour that is of concern here is more coarse-grained, at the function/module-level, which we believe is closer to the level of granularity that one uses to visualize software/hardware execution.

assumed to be of infinite size in software [146]. Therefore, it is not possible to validate in the C whether a stream (FIFO) is full. When compiled to hardware, the FIFOs have a default size of 1, unless specified otherwise by the user.

Broadly speaking, the Vivado HLS software streaming specification is different from the actual hardware that is produced. These discrepancies can lead to hardware bugs that cannot be debugged using software methodologies. It also makes the visualization of the generated hardware more difficult for a designer. In our work, we propose a method of writing software for streaming hardware which more closely models the hardware produced. We are not aware of any other HLS tools which, instead of using vendor-specific pragmas/APIs, support the use of the producer-consumer pattern with Pthreads to create streaming hardware.

7.3 Producer-Consumer Threads in Software

The producer-consumer programming pattern comprises a finite-size buffer and two classes of threads, a *producer* and a *consumer* [87]. The producer stores data into the buffer and the consumer takes data from the buffer to process. This *decouples* the producer from the consumer, allowing them to naturally run at different rates, if necessary. The producer must wait until the buffer has space before it can store new data, and the consumer must wait until the buffer is non-empty before it can take data. The *waiting* is usually realized with the use of a software variable, `semaphore`. A semaphore is a `POSIX` standard [59], which allows processes and threads to synchronize their actions. It has an integer value, which must remain non-negative. To *increment* the value by one, the `sem_post` function is used, and to *decrement* the value by one, `sem_wait` function is called [80]. If the value is already zero, the `sem_wait` function will block the process, until another process increases the semaphore value with `sem_post`.

The pseudo-code below (taken from [122]) shows the typical producer-consumer pattern using two threads.

```
1:  producer_thread {
2:    while (1) {
3:      // produce something
4:      item = produce();
5:      // wait for an empty space
6:      sem_wait(numEmpty);
7:      // store item to buffer
8:      lock(mutex);
```

```
9:      write_to_buffer;
10:     unlock(mutex);
11:     // increment number of full spots
12:     sem_post(numFull);
13:   }
14: }
15:
16: consumer_thread {
17:   while (1) {
18:     // wait until buffer has data
19:     sem_wait(numFull);
20:     // get item from buffer
21:     lock(mutex);
22:     read_from_buffer;
23:     unlock(mutex);
24:     // increment number of empty spots
25:     sem_post(numEmpty);
26:     // consume data
27:     consume(item);
28:   }
29: }
```

In a producer-consumer pattern, the independent producer and consumer threads are continuously running, thus they contain infinite loops (line 2 and 17). The buffer is implemented as a circular array. Two semaphores are used, one to keep track of the number of spots available in the buffer (lines 6 and 25), and another to keep track of the number of items in the buffer (lines 12 and 19). Observe that updates to the buffer are within a critical section – i.e. a `mutex` is used enforce mutual exclusion on changes to the buffer itself (lines 8–10 and 21–23).

7.4 Producer-Consumer Threads in Hardware

As mentioned above, we believe that the producer-consumer pattern is an ideal software approach to describe streaming hardware. Streaming hardware is always running, just as the producer-consumer threads shown above. Different streaming hardware modules execute concurrently and independently,

as with the producer-consumer threads. To fork threads, we use Pthreads, which is a standard known by many software programmers. Inputs and outputs are typically passed between streaming modules through FIFOs. The circular buffer described above is essentially a FIFO, with the producer writing to one end, and the consumer reading from the other end.

Using the producer-consumer pattern with Pthreads, we can re-write the example code shown for Vivado HLS as below.

```
1:  void *func_A(PTHREAD_FIFO *in, PTHREAD_FIFO *temp) {
2:    ...
3:    while (1) {
4:      // read from FIFO
5:      int a = pthread_fifo_read(in);
6:      // do work
7:      ...
8:      // output to FIFO
9:      pthread_fifo_write(temp);
10:   }
11: }
12: ...
13: void top(PTHREAD_FIFO *in, PTHREAD_FIFO *out) {
14:   ...
15:   pthread_create(func_A, ...);
16:   pthread_create(func_B, ...);
17:   pthread_create(func_C, ...);
18:   ...
19: }
20:
21: int main() {
22:   // declare and size FIFOs
23:   PTHREAD_FIFO *in =
        pthread_fifo_malloc(/*width*/32, /*depth*/1);
24:   PTHREAD_FIFO *out =
        pthread_fifo_malloc(/*width*/32, /*depth*/1);
```

```
25:   // invoke top-level function
26:   top(in, out);
27:   // fill up the input FIFO, as soon as the FIFO has data
28:   // the hardware executes
29:   for (i=0; i<SIZE; ++i) {
30:     pthread_fifo_write(in, in_array[i]);
31:   }
32:   // get output from the output FIFO
33:   for (i=0; i<SIZE; ++i) {
34:     out_array[i] = pthread_fifo_read(out);
35:   }
36:   // free FIFOs
37:   pthread_fifo_free(in);
38:   pthread_fifo_free(out);
39:   ...
40: }
```

The main differences here are the use of an infinite loop, Pthreads, and `PTHREAD_FIFOs`[2]. The infinite loop (line 3) keeps the loop body of the kernel function continuously running. We *pipeline* this loop, to create a streaming circuit. The advantage of using *loop pipelining*, versus pipelining the entire function, is that there can also be parts of the function that are not streaming (only executed once), such as for performing initializations. The `top` function, which is called only once (line 26), forks a separate thread for each of its sub-functions (lines 15–17). The user does not have to specify the number of times the functions are executed – the threads automatically start executing when there is data in the input FIFO. This closely matches the *always running* behaviour of streaming hardware. In this example, each thread is both a consumer *and* a producer. It consumes data from its previous stage and produces data for its next stage.

The `PTHREAD_FIFO` functions provide users with a software API which they can use to create streaming hardware in HLS. `Pthread_fifo_malloc` sizes the FIFOs in software to be the same as those in hardware (lines 23–24). `Pthread_fifo_write` pushes data into one end of a FIFO (lines 9, 30); previously stored data can be read from the other end with `pthread_fifo_read` (lines 5, 34). The `pthread_fifo_read/write` functions provide the blocking capability with the use of semaphores. This is

[2]LegUp can also create streaming hardware using sequential `C` code without the use of Pthreads, similar to Vivado HLS, and the keyword `FIFO` is reserved for the type of FIFOs used for this style of code.

described in more detail below. `Pthread_fifo_free` frees any memory allocated by `pthread_fifo_malloc` (lines 37–38)[3].

The multi-threaded code above can be compiled, concurrently executed, and debugged using standard software tools. We believe that portability is an important design consideration, and that a design should not be tied to a particular vendor, as is what happens when vendor-specific pragmas are required to produce the desired hardware. Our method aims to keep the HLS source code as a *standard* software program.

7.4.1 `FIFO` Details

This section describes how we create a `PTHREAD_FIFO` and its associated functions. The `PTHREAD_FIFO` is defined as a *struct*:

```
typedef struct {
  // bit-width of the elements stored in the FIFO
  int width;
  // the number of elements that can be stored
  int depth;
  // data array holding the elements
  long long *mem;
  // keeps track of where in the array to write to
  unsigned writeIndex;
  // keeps track of where in the array to read from
  unsigned readIndex;
  // keeps track of the number of occupied spots
  sem_t numFull;
  // keeps track of the number of empty spots
  sem_t numEmpty;
  // mutual exclusion for data array access
  pthread_mutex_t mutex;
} PTHREAD_FIFO;
```

The elements of the *struct* are used to define the storage, its width/depth, and where to read/write from/to in the storage. The data array is used as a *circular* buffer to create the FIFO behaviour.

[3] `pthread_fifo_free` is only used in software to prevent memory leaks. It is stripped away when compiled to hardware, as is described in Section 7.4.2.

Its type is a `long long`, making it capable of handling the largest standard `C` data type, though it can also be used to hold anything smaller. When compiled to hardware, the *width* variable is used to parametrize the hardware FIFO, which can be of any arbitrary width. Semaphores are employed to create the producer-consumer behaviour between threads and a mutex is used to ensure atomic access to the shared storage. When `pthread_fifo_malloc` is called, it allocates the data array and initializes all member variables, including the semaphores and the mutex. `pthread_fifo_free` frees all memories which have been allocated.

Using the `struct`, `pthread_fifo_write` follows the logic described in the `producer_thread` of the pseudo-code in Section 7.3, and `pthread_fifo_read` follows the logic of the `consumer_thread`. `Pthread_fifo_write` first waits until there is an empty spot in the FIFO (using `sem_wait` on the `numEmpty` semaphore), then gets the lock, stores the data into the `writeIndex` position of `mem`, updates `writeIndex`, releases the lock, and finally increments `numFull`. `Pthread_fifo_read` waits until the FIFO is non-empty (using `sem_wait` on the `numFull` semaphore), gets the lock, reads the data at the `readIndex` position of `mem`, updates `readIndex`, releases the lock, and finally increments `numEmpty`.

7.4.2 Hardware Architecture

In hardware, a `PTHREAD_FIFO struct` gets compiled to a hardware FIFO. Figure 7.1 shows the interfaces between a FIFO and its producer, module `A`, and its consumer, module `B`. For a FIFO interface, modules use `RVD` (Ready, Valid, Data) signals, which is a typical hand-shaking interface used in streaming architectures. The semaphores of the `PTHREAD_FIFO struct`, which keep track of whether the FIFO is full/empty in software, are simply turned into the `not_full` and `not_empty` signals in hardware. On a call to `pthread_fifo_write` for module `A`, the `not_full` signal is checked, and if it is high, the data is written to the FIFO via the `write_data` signal. If the `not_full` signal is low, meaning the FIFO is already full, the `out_ready` signal of module `A` is de-asserted, which stalls module `A`. The stall logic is described below in more detail in Section 7.4.4. For `pthread_fifo_read` from module `B`, the `not_empty` signal is checked, and if it is high, the data is returned via the `read_data` signal. If the `not_empty` signal is low (FIFO is empty), the `in_valid` signal is de-asserted, which stalls module `B`. This implementation removes any additional hardware overhead for the semaphores/mutex, while still allowing software to be executed like hardware.

Figure 7.1: FIFO interfaces.

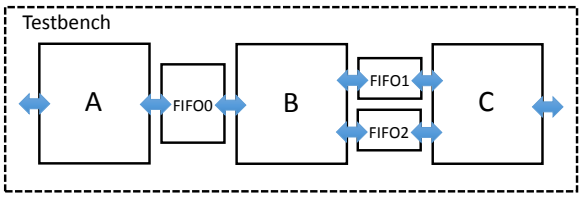
7.4.3 Multiple Software Threads to Multiple Streaming Hardware Kernels

In a streaming architecture, multiple streaming modules may be chained together, transferring data from one streaming module to the next, as shown in Figure 7.2a. This is a typical architecture used in image/video processing applications. We can create this architecture by simply forking a thread for each of `A, B`, and `C`, as described above, and passing in `FIFO0` as an argument to `A` and `B`, and `FIFO1` and `FIFO2` to `B` and `C`. As per Pthread standards, multiple arguments to a thread must be passed by creating a struct which contains all of the arguments, and then passing a pointer to that struct in the `pthread_create()` routine [7]. We use the points-to analysis to automatically determine which FIFOs need to be connected to which hardware modules. The tool also determines whether a module *writes* to the FIFO, or *reads* from the FIFO, and the integrated system generator automatically connects the appropriate input/output FIFO ports to their corresponding streaming module ports.

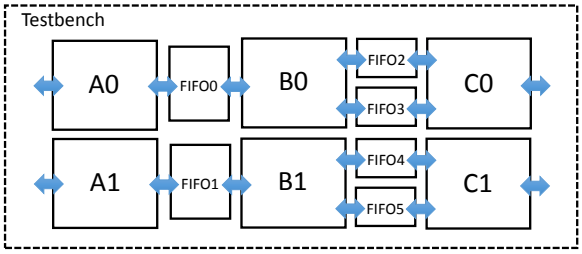
With the producer-consumer threads, all processes, in both software and hardware, start executing as early as possible (i.e. as soon as there is data in the input FIFO). As previously mentioned, the `dataflow` pragma in Vivado HLS achieves a similar effect in hardware, but its parallel execution, which can often be the source of bugs, cannot be debugged in software using standard debugging tools. Since we only use standard software methodologies, all of our code, including the FIFO functions, can be compiled with `GCC` and debugged with `GDB`. As such, most of the design effort can be spent at the software stage.

Another advantage of using Pthreads is that one can also easily replicate streaming hardware. In LegUp, each thread is mapped to a hardware instance by default, hence forking multiple threads of the same function creates replicated hardware instances. For instance, if the application shown in Figure 7.2a is completely parallelizable (say data-parallel), one can exploit spatial hardware parallelism by forking two threads for each function, to create the architecture shown in Figure 7.2b[4]. This methodology therefore allows exploiting *both* spatial (replication) and pipelined hardware parallelism all from software. We think that this is easier and more intuitive than manually instantiating a synthesized core multiple times using HDL, or using a system generator tool such as the Vivado IP Integrator [143], which uses a

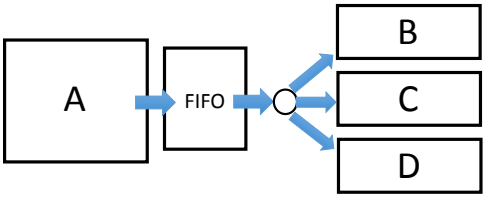
[4]It is not only limited to massively parallelizable architectures. Depending on the nature of the application, one can select which function to parallelize.



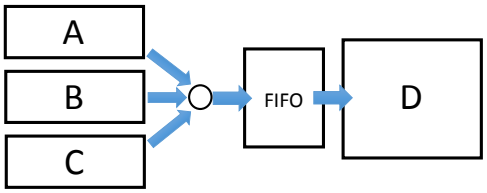
(a) Multiple streaming modules.



(b) Multiple replicated streaming modules.



(c) One-to-many streaming modules.



(d) Many-to-one streaming modules.

Figure 7.2: Multiple streaming modules connected through FIFOs.

schematic-like block design entry to connect modules by drawing wires and pins. Both approaches would be foreign concepts to those coming from the software domain. Our methodology uses purely software concepts to automatically create and connect multiple parallel streaming modules together.

Our approach is also able to handle more complex architectures, where multiple consumers receive data from a single producer through a single FIFO, as shown in Figure 7.2c, and where multiple producers can feed data to a single consumer through a single FIFO, as shown in Figure 7.2d. The former architecture can be useful for applications with a *work queue*, where a producer writes to the work queue, and multiple workers (consumers), when ready, take work-items from the queue to process. The latter architecture can be used for applications such as *mapReduce* [25], where multiple mappers can map to the same reducer. Both architectures can be created from software by giving the same FIFO argument to the different threads. We automatically synthesize arbiters to handle contention that may occur when multiple modules try to access the same FIFO in the same clock cycle – modules may stall if not given immediate access. Code examples on how to create the architectures shown in Figures 7.2a, 7.2b, 7.2c, and 7.2d, are presented in Appendix F. We believe this one-to-many, or many-to-one FIFO architecture, with automatic synthesis of arbitration logic, is a unique aspect of our work, as both Vivado HLS and Altera's OpenCL SDK require there to be a sole *single* writer and a reader to/from a FIFO.

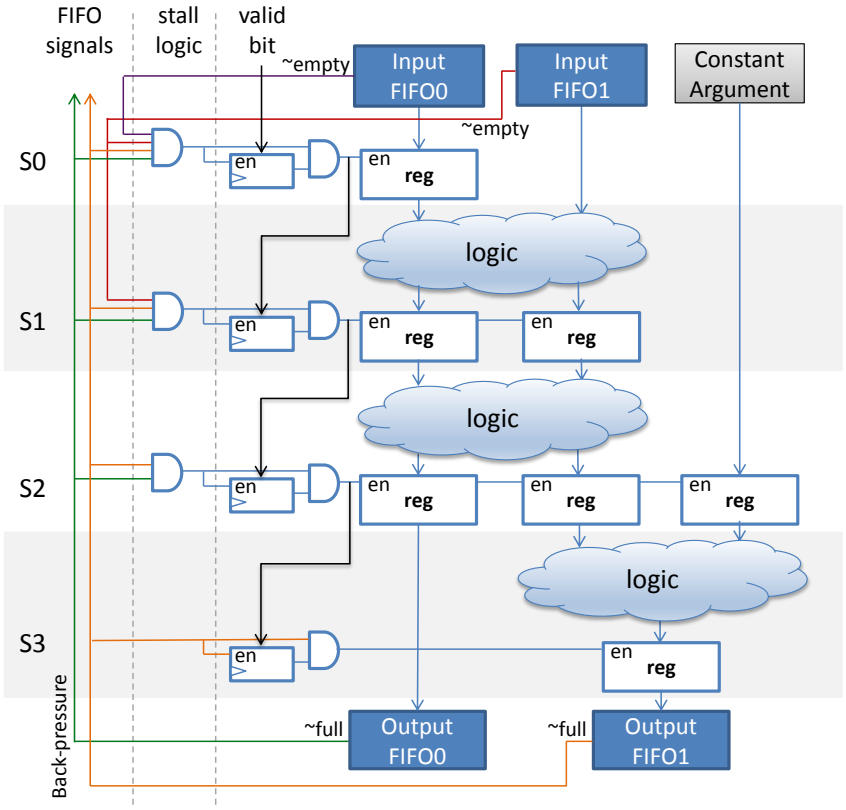


Figure 7.3: Streaming circuit data-path and stall logic.

7.4.4 Streaming Datapath and Stall Logic

The datapath and its stall logic for streaming hardware is shown in Figure 7.3. The figure illustrates how we handle different types of inputs/outputs, which can be received/outputted at different pipeline stages of a streaming circuit, as well as how our stall logic is used to only stall the pipeline stages that absolutely need to stall. There are two input FIFOs, a non-FIFO argument input, and two output FIFOs. The `S`'s denote pipeline stages, with registers at each stage to pipeline data. The valid bits are used to indicate which stages of the pipeline contain valid data. The streaming circuit is a straight-line datapath, without any control flow. Like other HLS tools, we remove any diverging branches with if-conversion and back edges by unrolling any internal loops (those residing inside the while loop). Any sub-functions called within the while loop are inlined. During if-conversion, we also predicate operations with side effects or those that can cause memory contention (i.e. load/store, FIFO read/write) so that they only trigger for the correct if/else conditions.

The stall logic ensures that the hardware can stall appropriately and produce a functionally correct result. It directly impacts the QoR (quality-of-result) of the circuit, as stalls increase circuit latency, and the stall logic affects circuit area and Fmax. It is desirable to stall only when necessary, and also to

minimize the stall circuitry. For the architecture shown in Figure 7.3, there are two scenarios wherein the circuit can stall: 1) When any of the input FIFO becomes *empty*, and 2) when any of the output FIFOs become *full*. In both cases, a stall does not necessarily stall the *entire* pipeline, but only those pipeline stages which absolutely need to stall. For instance, in the case of `Input FIFO0`, its data is required in `S0` (pipeline stage 0). Consequently, if this FIFO becomes empty, only `S0` stalls. Data from `Input FIFO1` is needed in `S1`, so if this FIFO is empty, `S1` and `S0` stall. `S0` also needs to stall in this case since its next stage is stalled (allowing it to continue would overwrite valid data in `S1`). `Output FIFO0` is written from `S2`, hence when this FIFO is full, it stalls `S2`, `S1`, and `S0`. When `Output FIFO1` is full, the entire pipeline stalls. In general, a FIFO being full/empty stalls the first pipeline stage where its data is written/read from, and all of the prior pipeline stages. This architecture allows the later pipeline stages to continue making forward progress, even when a FIFO becomes empty/full. For instance, when `S0` stalls due to `Input FIFO0` only, `S1`, `S2`, `S3` can continue. When `Output FIFO0` is full, valid data in `S3` can continue and be written to the `Output FIFO1` (given that it is not full).

There are also scenarios where stall circuitry is unnecessary. For instance, a *constant* argument (such as an integer value), is stored in registers when the module starts and remains unchanged during its execution. We do not create any stall logic for this argument, as it will not be overwritten during the execution. This helps to reduce circuit area and the fan-out of the stall signals, which can become large when there are many FIFOs and pipeline stages.

In summary, there are three conditions for a pipeline stage to be enabled: 1) Its valid bit must be asserted to indicate there is valid data, 2) any input FIFOs, from which its data is needed in this or a downstream pipeline stage, must not be empty, and 3) any output FIFOs, which are written to from this or a downstream pipeline stage, must not be full. A FIFO can also be shared between multiple modules through an arbiter, as was shown in Figs. 7.2c and 7.2d. In such cases, we stall in the same manner, depending on whether it is an input or an output FIFO[5]. It is worth noting that, although we primarily discuss FIFO memories in this work, streaming hardware can also access non-FIFO RAMs, with arbitration and stall logic created as described in Chapter 6.

7.5 Experimental Study

In this section, we first discuss the streaming benchmarks which use the producer-consumer pattern with Pthreads, as well as their resulting hardware. We use four different applications from various fields, including image processing, mathematics/finance and data mining. For each benchmark, we create two

[5]For an input FIFO, the `grant` signal from the arbiter is `AND`'ed with the `not_empty` FIFO signal, and this output goes to the stall logic. For an output FIFO, the `grant` signal is `AND`'ed with the `not_full` FIFO signal.

versions, a *pipelined-only* version and a *pipelined-and-replicated* version. In the pipelined-only version, there are one or more functions which are connected together through FIFOs, as in Figure 7.2a, but no modules are replicated. For the pipelined-and-replicated version, we parallelize each benchmark with one or more functions (modules) executing on multiple threads, yielding architectures similar to Figs. 7.2b and 7.2d. In both versions, all kernel functions are fully pipelined with multiple pipeline stages, and receive/output new data every clock cycle (II=1).

Each benchmark also includes golden inputs and outputs to verify correctness. Each generated circuit was synthesized into the Altera Stratix V FPGA (5SGSMD8K1F40C2) with Quartus 15.0. For performance and area comparison, we also use a commercial HLS tool to synthesize one of the pipelined-only benchmarks, Canny, targeting the Xilinx Virtex 7 FPGA (XC7VX980TFFG1930-2). The Virtex 7 is on the same technology node as the Stratix V (28 nm) [139, 104], and both FPGAs have the second fastest speed grade[6]. The commercial tool does not support replicating hardware from software, thus none of the pipelined-and-replicated benchmarks were used for this tool. For both LegUp and the commercial HLS tool, a 3`ns` (333MHz) clock period constraint was supplied, which can be given as a configuration parameter to each tool (given as a setting in the GUI for Vivado HLS, and as a `Tcl` parameter in LegUp). This is used by the scheduling stage of HLS create a circuit that aims to meet the target frequency.

7.5.1 Benchmarks

Mandelbrot is an iterative mathematical benchmark which generates a fractal image. For each pixel in a 512×512 image, it iteratively computes whether it is bounded (inside the Mandelbrot set) or diverges to infinity (outside the Mandelbrot set), and displays its colour accordingly. Computations are done in fixed-point for this benchmark. Each pixel is independent from others, hence this application is easily parallelizable. In the pipelined-and-replicated version with four threads, each thread processes a quadrant of the image.

The *Black-Scholes* benchmark estimates the price of European-style options. It uses Monte Carlo simulation to compute the price trajectory for an option using random numbers. Ten thousand simulations are conducted, with 256 time steps per simulation. The system diagram for the pipelined-only version is shown in Figure 7.4 as a dot graph. This dot graph, automatically created by our system generator, shows the different modules, as well as the connections between them. White boxes are hardware modules; blue ovals are FIFOs; dark-blue overalls (appearing later) are memory blocks. This

[6]Altera and Xilinx may not have the same *binning* process used for deciding speed grades, however, this is the closest comparison we could use.

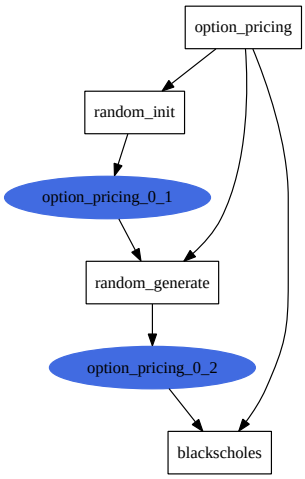


Figure 7.4: System diagram for the Black-Scholes option pricing benchmark for the pipelined-only architecture.

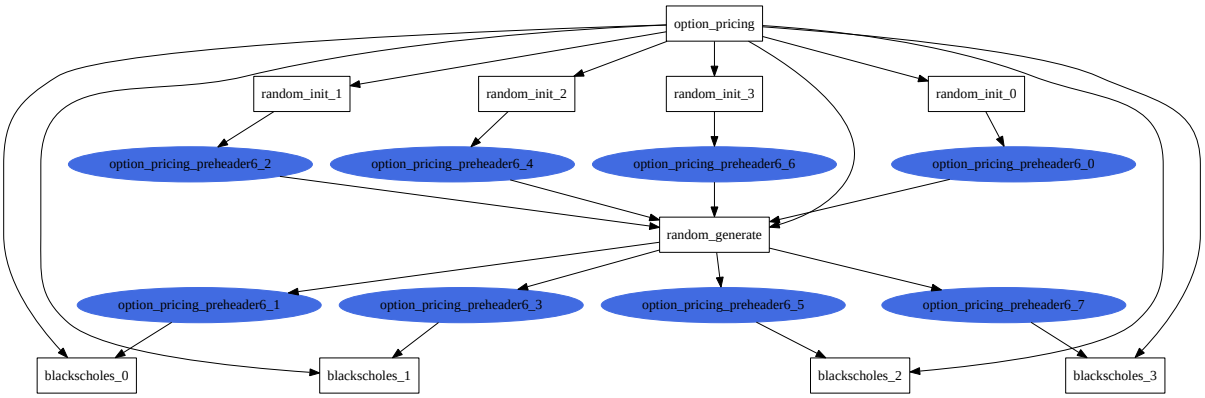


Figure 7.5: System diagram for the Black-Scholes option pricing benchmark for the pipelined-and-replicated architecture.

benchmark consists of three kernel functions, `random_init`, `random_generate`, and `blackscholes`, and the wrapper function, `option_pricing`, which creates the necessary intermediate FIFOs between the kernel functions and forks their threads. The `random_init` and `random_generate` are an implementation of the *Mersenne twister* [130], which is a widely used pseudo-random number generator. These two kernels were adapted from [108], originally written in OpenCL. The init function initializes the random number generator in the generate function. The blackscholes function uses the random numbers to price a European option using the Black-Scholes formula. In the pipelined-and-replicated version, shown in Figure 7.5, we parallelize the initialization and the Black-Scholes functions, each with four threads. For the generate function, we modify its logic so that it can receive four initializations from the initialization threads, and generate four random numbers concurrently. Each random number is used by an independent Black-Scholes' thread, with four threads concurrently computing four prices.

The *Canny* benchmark implements the well-known Canny edge detection algorithm [12] for a 512×512 image. The multi-stage algorithm is implemented with four kernel functions, `gaussian filter`, `sobel filter`, `nonmaximum suppression`, and `hysteresis`, as well as its wrapper function `canny`, as shown in Figure 7.6. The Gaussian filter first smooths the input image to remove noise. The Sobel filter then finds the intensity gradients. The non-maximum suppression removes pixels not considered to be part of an edge. Then finally, hysteresis finalizes the edges by suppressing all the other weak edges. Every clock cycle, each kernel receives a new pixel from the previous kernel stage and outputs a pixel to its next-stage kernel.

In the pipelined-and-replicated version, we parallelize each kernel function with four threads. We again divide the image into four sections (this time with 128 rows each), with each section to be processed by a set of replicated modules (i.e. rows 0–127 are processed by a first set of copies of the Gaussian, Sobel, non-maximum suppression, and hysteresis kernel modules). The data required by each set of modules, however, is not completely mutually exclusive, since each kernel uses either a 5×5 or a 3×3 filter. For instance, the Gaussian filter, which uses a 5×5 filter, requires up to two rows outside of its assigned section. For example, when working on row 127, values of pixels in rows 128 and 129 are needed, which belong to the next section of rows. To manage this, pixel values for border rows are communicated between adjacent copies of the kernels. Moreover, to minimize stall time arising from needed data in border rows, even-numbered sections (containing rows 0–127 and rows 256–383) proceed from the bottom row to the top; odd-numbered sections (containing rows 128–255 and rows 384–511) proceed from the top row to the bottom. The architecture for this parallelized version is shown in Figure 7.7

The `k-means` benchmark implements the k-means clustering algorithm [47] used in data mining. It partitions `n` data points into one of `k` clusters defined by centroids. Our version has 1,000 data points with four clusters. We use the *mapReduce* programming paradigm to implement k-means. A `mapper` iteratively maps each data point to a cluster, and a `reducer` updates the centroids with each data point. In the pipelined-only version, there is a single mapper and a single reducer. The mapper maps all data points to one of the clusters, and the reducer updates the centroids for all clusters. In the pipelined-and-replicated version, there are four mappers and four reducers. Each mapper maps to a single cluster, and each reducer updates the centroid for a single cluster. The architecture for the parallelized version is shown in Figure 7.8[7]. The figure shows nine hardware modules (four mappers, four reducers, and the wrapper function), eight memories (four x and y centroid coordinates stored in registers), and many

[7]The purpose of showing the dot graphs is to illustrate the general architecture and the complexity of the benchmarks. Details such as the *names* of the modules are not significant.

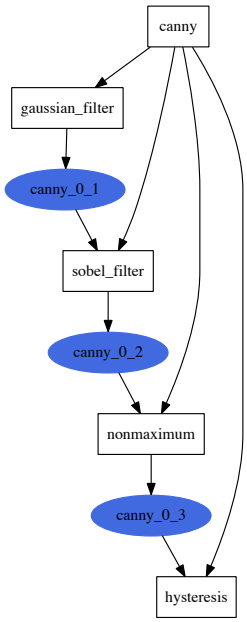


Figure 7.6: System diagram for the Canny benchmark for the pipelined-only architecture.

FIFOs. These FIFOs are used to send data inputs to each mapper, pass data from each mapper to each reducer (each mapper can write to any of the reducer FIFOs using the architecture shown in Figure 7.2d), and also indicate when a mapper or a reducer is done computing for an iteration. For each iteration, a mapper needs to know when all reducers have finished (updated the centroids), so that it can start the next iteration using the updated centroids, and a reducer also needs to know when all mappers have finished so that it can average the accumulated centroid value. We use a 1-bit-wide depth-of-1 FIFO, which is implemented in registers, to send the *done* signal. Although the figure looks very complicated, the actual source code is a standard mapReduce algorithm, which is pretty simple. All of the modules, memories, and FIFOs are automatically generated and connected, which is the beauty of using HLS.

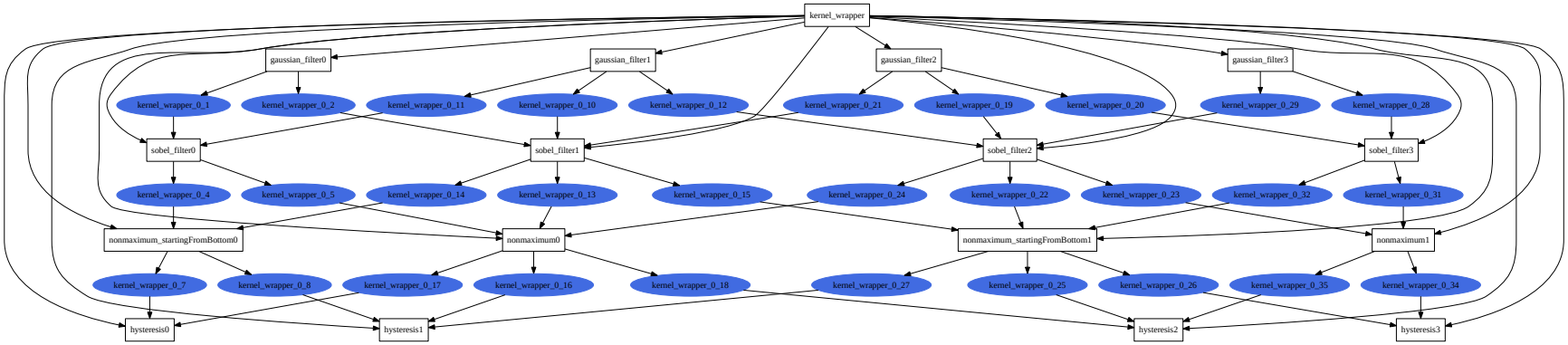


Figure 7.7: System diagram for the Canny benchmark for the pipelined-and-replicated architecture.

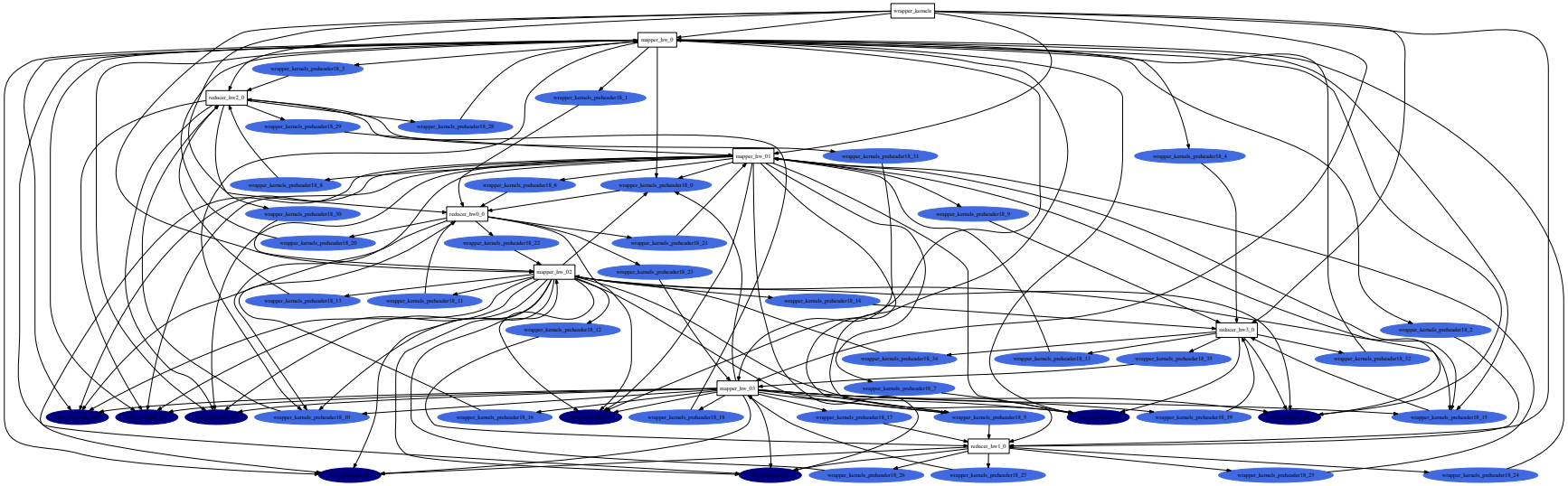


Figure 7.8: System diagram for the k-means benchmark for the pipelined-and-replicated architecture.

Table 7.1: Performance and area results for *pipelined-only* benchmarks for LegUp HLS.

Benchmark	Time (μs)	Cycles	Fmax(MHz)	ALMs	Registers	DSPs	M20Ks
Mandelbrot	738.6	262208	355	1101	2746	112	0
Black-Scholes	16736.7	2560714	153	8575	28963	45	5
Canny	787.95	264752	336	1246	2415	0	10
K-means	70.4	20908	297	8499	20681	16	115
Geomean	910.01	246910.57	271.33	3162.11	7938.86	16.85	8.71

Table 7.2: Performance and area results for *Canny* benchmark for a commercial HLS tool.

Benchmark	Time (μs)	Cycles	Fmax (MHz)	LUTs	Registers	DSP48s	BRAMs
Canny	792.64	264743	334	1427	1948	0	5
Ratio vs. LegUp	1.006 (0.994×)	1.00 (1.00×)	0.99	1.15	0.81	1	0.5

7.5.2 Results

Table 7.1 shows the performance and area results for all the pipelined-only benchmarks compiled with LegUp HLS. There are three performance metrics (total wall-clock time (# cycles × clock period), total number of clock cycles, and Fmax) and four area metrics (number of ALMs, registers, DSPs, and M20Ks). As previously mentioned, all circuits have an II=`1`, and were given a clock period constraint of 3`ns` (333 MHz), except for Black-Scholes, which was given 6`ns` (167 MHz). All circuits run roughly within +/-10% of the target frequency. For Black-Scholes, due to a recurrence in the benchmark, we had to lower the clock period constraint supplied to LegUp to meet II=`1`.

Table 7.2 shows the commercial HLS tool's result for the Canny benchmark. The performance results are nearly identical to that of LegUp HLS, with the total wall-clock time 0.6% higher than LegUp. Targetting the Virtex 7 FPGA, the area is reported in terms in LUTs, registers, DSP48s, and 18KB Block RAMS. The circuit generated by the commercial tool uses 15% more LUTs, but it also uses 19% less registers and half the number of RAMs. For this performance/area comparison, we note that there are differences in the FPGA architectures and the vendor FPGA CAD tools that can lead to different results. For example, although Virtex 7 and Stratix V are fabricated in the same 28 nm TSMC process, Stratix V uses fracturable 6-LUTs that are more flexible than Virtex 7's fracturable 6-LUTs. Likewise, we expect that two vendor's FPGA CAD tools employ different RTL/logic synthesis, place-and-route algorithms. Despite these potential sources of error, the similarity in the performance results for the Canny benchmark gives us confidence that LegUp HLS produces a reasonably good-quality implementation. In LegUp's case, this is achieved through software-only methodologies requiring no special pragmas.

Table 7.3 shows the results for LegUp HLS, for the pipelined-and-replicated benchmarks. Compared to pipelined-only, we see a geometric mean speedup of 2.8× in terms of total wall-clock time. Clock cycle

Table 7.3: Performance and area results for *pipelined-and-replicated* benchmarks for LegUp HLS.

Benchmark	Time (μs)	Cycles	Fmax (MHz)	ALMs	Registers	DSPs	M20Ks
Mandelbrot	231.8	65606	283	4192	11006	448	0
Black-Scholes	4297	640252	149	19182	55843	180	20
Canny	264.8	70706	267	7396	14232	48	76
K-means	42.2	10712	254	11218	25919	64	120
Geomean	324.81	75102.66	231.25	9037.68	21820.8	125.46	20.25
Ratio vs. Table 7.1	0.36 (2.80×)	0.30 (3.29×)	0.85	2.86	2.75	7.45	2.32

improvement is higher with 3.29×, but Fmax drops 15% on average, due to higher resource utilization and more complex hardware architectures. On a per benchmark basis, Black-Scholes shows close to linear speedup in wall-clock time: 3.89×. Mandelbrot also shows linear speedup in clock cycles, but Fmax drops due to the use of 448 DSP blocks. Canny shows 3.74× speedup in clock cycles, and 2.98× speedup in wall-clock time. For k-means, the work load for each mapper/reducer, and thus the speedup from parallelization, is dependant on the initial coordinates of the centroids and the data points. We initialize each centroid to be at the centre of each quadrant of the entire x/y space, and randomly generate the initial data point coordinates. With this, the four mappers/reducers obtain 1.95× speedup in clock cycles and 1.67× in wall-clock time.

In terms of area, the pipelined-and-replicated benchmarks show average increases of 2.86×, 2.75×, 7.45×, and 2.32×, in ALMs, registers, DSPs, and M20Ks, respectively. For DSP usage, all benchmarks increased linearly by a factor of four, with the exception of Canny. In the pipelined-only case, the compiler was able to optimize multiplications with constant filter coefficients to shifts and adds, however this optimization did not occur in the replicated case, due to the structural code changes, utilizing 48 DSP blocks. For ALMs, the biggest relative increase was with Canny, which again, for the replicated scenario, the compiler was not to optimize the program as effectively as the pipelined-only, and we also had added additional logic and FIFOs to allow communication of the border rows. The smallest relative increase was with k-means, where most of the ALMs and M20Ks were used by eight dividers, used to average the x and y coordinates for the four centroids. Eight dividers were also needed in the pipelined-only case to meet II=1. In the pipelined-and-replicated case, each reducer handled one cluster, with two dividers each, thus the total number of dividers remained the same.

Overall, our methodology allows the synthesis of a diverse space of streaming hardware architectures that can be pipelined or pipelined *and* replicated, all from software. For massively parallel applications, replication of streaming hardware is as easy as forking multiple software threads. For the Canny benchmark, our streaming hardware showed very competitive results to that of a commercial tool. Our pipelined-only circuits provide high throughput, with an II=1, while the pipelined-and-replicated circuits further improve performance, at the expense of FPGA resources.

7.6 Summary

In this chapter, we discussed our methodology that allows standard software techniques to specify pipeline and spatial FPGA hardware parallelism. Our work allows software-threaded programs to model streaming hardware more accurately than the existing solution from Vivado HLS. The closer alignment between software and hardware allows a designer to better understand the generated hardware. It also enables more debugging to happen in software, which is much less difficult and time consuming than hardware debugging. Using Pthreads can open up many options, such as creating multiple streaming kernels that work concurrently. Our work also permits the creation of circuit architectures that are not feasible to realize in other HLS tools, such as a FIFO with multiple writers, where the arbitration is also automatically generated.

This work has been published in the 2016 IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) [16].

Chapter 8

Conclusions

8.1 Summary of Contributions

With the end of CPU clock speed scaling, multi-core processing has become a necessity for achieving higher performance in software. There has been considerable effort in the software domain on providing efficient methods to harness the power of multi-cores, via new programming languages, standards, libraries, and compilers. On the contrary, there has been comparatively little work on providing a standard approach to exploit parallel custom hardware. Much of it still remains a manual process, requiring an engineer to understand and specify the parallelism in hardware. Some advancements have been made, with a select few tools providing vendor-specific annotations to drive the generation of parallel hardware. Such approaches, however, add another level of complexity and perplexity to a process that is already overwhelming to those not familiar with hardware or HLS.

We believe that the correct approach to specifying hardware parallelism from software in HLS is to use a *standard* software parallel programming methodology that is already widely known and used by many engineers, such as Pthreads and OpenMP. With both standards being in use for almost two decades, Pthreads and OpenMP are intuitive to use and understood by even relatively novice software engineers. This dissertation presents new methodologies that allow one to use Pthreads and OpenMP to perform multi-core processing in hardware on an FPGA. All of our work is implemented within the LegUp HLS framework. We summarize our contributions below.

Chapter 3 described the synthesis of a multi-threaded software program with Pthreads and OpenMP to a processor/parallel-accelerator hybrid platform, where threads are compiled to concurrent accelerators, with the sequential portion of software executed on a soft MIPS processor. We described the

automated generation of an SoC that allows one to create an *entire* system from software, with tasks such as software/hardware partitioning, off-chip memory setup, and interconnect generation, all handled by LegUp. This enables those without hardware knowledge to benefit from multi-core processing in hardware, and with its ease-of-use, even hardware engineers can take advantage of significantly reduced design time. As multi-threaded programming often requires synchronization of threads, we also described our HLS support for mutexes and barriers. This work has been published in the IEEE International Conference on Field-Programmable Technology (FPT) [14].

Chapter 4 discussed the support for using the *hard* ARM processor on the Altera Arria V SoC FPGA, which improves the performance and the applicability of our processor-accelerator hybrid systems by harnessing the power of the 1.05 GHz dual-core CPU. We outlined the flow of using the ARM processor with and without an OS. We also discussed our support for enabling DMA transfers between hardware accelerators and off-chip memory, with the option of using our Double Buffering Module to further improve performance and memory bandwidth. We showed that our ARM hybrid systems can significantly outperform the MIPS, the ARM, and even two other `x86` processors. This work is to be submitted to IEEE Transactions on Very Large Scale Integration Systems (TVLSI).

Chapter 5 showed that we can also create a parallel hardware system *without* requiring a processor in the system. In the *hardware-only* flow of LegUp, the entire multi-threaded software program is compiled to hardware, with parallel-threaded modules executing concurrently within a larger hardware system. We believe this flow to be beneficial for designs that are either constrained by FPGA resources, or for cases where all computations need to be performed in hardware. We showed that this methodology can bring significant benefits in speed, power, and area-delay product, compared to sequential hardware. This work is to be submitted to IEEE Transactions on Very Large Scale Integration Systems (TVLSI), along with the work in Chapter 4.

Chapter 6 investigated two different circuit topologies, the nested topology and the flat topology, as well as three different memory architectures. We showed that the flat topology, together with our system generator, enables efficient sharing of modules, memories, and functional units. We also discussed the dead-lock prevention logic automatically inserted by our system generator, which allows concurrent hardware modules to execute properly without deadlocks. For the memory architectures, we described the global, local, and shared-local memories, with additional options available for memory replication and memory-to-register conversion. We showed that the flat topology combined with the improvements to the memory architecture can yield substantial performance and area benefits. This work has been published in the 2015 IEEE International Conference on Field-Programmable Technology (FPT) [15].

Chapter 7 detailed one of our efforts to improve the usability of HLS in the context of parallel

hardware by providing support for using a commonly known multi-threaded programming method to *infer* streaming hardware. Specifically, we showed that we can use the producer-consumer pattern with Pthreads to create streaming hardware in LegUp. This allows us to use standard software to produce parallel hardware, where the software execution behaviour closely models the generated hardware behaviour. This contrasts to other HLS tools, which employ vendor-specific pragmas to *transform* sequential software to streaming hardware under the hood. Our methodology allows one to compile and debug the parallel execution behaviour in software using standard software tools, making it possible for more of the design and debugging to happen in the software abstraction layer. By using Pthreads, creating multiple streaming hardware modules becomes a matter of forking more threads, and we showed that we can create more complex parallel hardware structures than what are currently supported by other HLS tools. As a point of comparison, we showed that the performance of our streaming hardware generated with the producer-consumer pattern with Pthreads can match the performance of streaming hardware generated by a state-of-the-art commercial HLS tool. We also showed that by replicating streaming hardware modules with multiple threads, we can achieve higher performance. This work has been published in the 2016 IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) [16].

8.2 Future Work

8.2.1 Automated DMA Hardware Generation

As shown in Chapter 4, using DMA transfers between hardware accelerators and off-chip memory can greatly improve memory bandwidth, making it possible for FPGA hardware to outperform CPUs that are running more than an order of magnitude faster in clock frequencies than the FPGA hardware. In LegUp, although the generation of a hybrid system is currently automated, the instantiations of DMA cores and the Double Buffering Module remain a manual process. This can be a burdensome task for software engineers and we would like to automate this process in the future. This automation would work very similar to the current hybrid SoC generation flow. One would start by using the functions in our DMA software library in the user program. Then when the program is compiled to LLVM IR with Clang, LegUp can examine the IR and detect the DMA functions to determine how many DMA cores are needed, and which DMA cores need to be connected to which accelerators. This information can then be passed as `Tcl` commands to Qsys, to generate the complete SoC, including the DMA cores. The user can use a `Tcl` parameter to specify double buffering for an accelerator, which can be detected

in LegUp's Hardware Backend, to instantiate the Double Buffering Module.

8.2.2 Direct Accelerator-to-Accelerator communication

Currently in a hybrid system, an accelerator cannot communicate with another accelerator without going through the processor or the shared memory space (on-chip cache/off-chip memory). However, for streaming architectures, such as those shown in Chapter 7, direct communication links between accelerators may be needed. For instance, there can be a hybrid system with multiple streaming hardware accelerators, where data is directly streamed from one accelerator to the next, with the processor handling tasks such as setting up the accelerators. If data needs to be written to memory from one accelerator and read back by the next accelerator every time, the system would quickly become memory bandwidth limited. To create a high-performance system where multiple streaming accelerators which are connected together can continuously execute, the first accelerator should perform burst DMA transfers (possibly with double buffering) to receive a large chunk of data from off-chip memory, process the data, and pass it directly to the next accelerator by way of a streaming interface, such as a FIFO, or an Avalon Streaming Interface [4]. The next accelerator can then continuously receive, process, and transfer the data in the same manner, with the last streaming accelerator writing the output data back to off-chip memory via burst DMA transfers (possibly with double buffering). This proposed system can be thought of as combining the hybrid system with DMA transfers described in Chapter 4, with the multiple streaming kernels connected through FIFOs, which was described in Chapter 7. This new hybrid architecture would allow LegUp-generated hybrid systems to be applied to more complex streaming-style applications, such as those in video/image processing, which can require many filters to be applied to an image one after the next.

8.2.3 Peripheral Component Interconnect Express (PCIe) Support

LegUp's processor-accelerator hybrid systems are currently limited to architectures where the both the processor and accelerators reside on the FPGA. For high-performance computing, however, an `x86` processor connected to the FPGA over PCIe can be a more suitable platform. Previously, the LegUp research group had built a prototype application which involved an `x86` processor communicating with LegUp-generated hardware accelerators over PCIe. While the PCIe support worked for this particular application, and it showed that our accelerators can work with an `x86` processor, additional effort is required to make the PCIe support high-performance and robust. Having a PCIe support that is robust and readily available will allow LegUp to be used for a wider range of applications, such as those that

are used in data centres. With the PCIe support, we can also compare the performance and power consumption of LegUp-generated accelerators with GPU implementations.

8.3 Closing Remarks

We believe that much can be learned from looking at the past trends in software and hardware design. For example, in the past, engineers programmed microprocessors with assembly code. Nowadays however, compilers have matured to the extent that programming in assembly is typically only necessary for certain limited circumstances, such as for handling interrupts or communicating with I/Os. Even though hand optimizing assembly could lead to better performance, it is simply too cumbersome and time consuming to do. In the hardware domain, manually laying out transistors nowadays is only done for high-performance chips with extremely high volumes, such as Intel processors. Even in these chips, only the most critical parts of the chip are manually laid out. Although a full-custom chip could provide higher performance, it is simply too costly to do. In both cases, which are at very different abstraction layers, the level of design abstraction has migrated higher, to make software/hardware design more easier and efficient, even at the cost of some performance/area degradations.

We believe that HLS compilers are also on the same trajectory. With continuous improvements in HLS technologies, leading to better quality-of-results and ease-of-use, we are already seeing an increasing adaption of HLS in the hardware design community. The holy grail of HLS, is to *replace* HDL design, or to minimize the need for it to certain limited applications – just as software compilers minimized the need for assembly programming. We think that for this to become a reality, we must improve the *usability* of HLS. One must be able to use software to create a complete working system, without concerning over tasks such as setting up peripherals or off-chip memory interfaces. One must also be able to create high-performance hardware using a methodology that is intuitive and can be easily understood by even those without hardware knowledge.

This dissertation provided two key steps towards improving the usability of HLS, which allows one to: 1) create not just a core, but an entire SoC from software, and 2) use standard multi-threaded programming methodologies to create high-performance parallel hardware. Exploiting hardware parallelism on an FPGA is crucial for achieving high performance, just as multi-core processing is essential for modern CPUs. We have implemented all of our work within the LegUp high-level synthesis framework, which is open-source and freely downloadable by researchers around the world. With this dissertation, we are optimistic that our work will make a positive contribution for HLS to achieve its grand goal of replacing manual hardware design.

Appendix A

A Sample Code for Using MMAP to Map a Hardware Accelerator in Linux

This appendix provides an example code to illustrate how to use the `mmap` function to map and unmap a hardware accelerator in a Linux environment.

```
1:  #include <sys/mman.h>
2:  // open memory device file
3:  fd = open("/dev/mem", O_RDWR | O_SYNC);
4:  if (fd < 0) {
5:      perror("open");
6:      exit(EXIT_FAILURE);
7:  }
8:  ...
9:  // map physical address to virtual address
10: void *accel_mmap_ptr = mmap(NULL, accel_addr_size, PROT_WRITE,  MAP_SHARED,
                                fd, accel_physical_addr);
11: // check if the mapping succeeded
12: assert(accel_mmap_ptr != MAP_FAILED);
13: // cast pointer to the proper type to be used for communicating with accelerator
```

```
14: volatile unsigned long *accel_addr = (volatile unsigned long*) accel_mmap_ptr;
15: ...
16: // free the mapping
17: if (munmap(accel_mmap_ptr, accel_addr_size) < 0) {
18:     perror("munmap");
19: }
20: // close memory device file
21: close(fd)
```

The example maps to virtual memory an accelerator which has a memory-mapped address of `accel_physical_addr`, and is taking up `accel_addr_size` of memory-mapped space (the `status`, `data`, and `argument` memory-mapped registers described in Section 3.4.2 are mapped within this range). To use `mmap`, one must first open the memory device file to get the file descriptor (line 3), and it should be checked that this operation has succeeded (lines 4–7). This file descriptor is given to `mmap`, along with the starting physical address (`accel_physical_addr`), and the size of memory in bytes to be mapped (`accel_addr_size`) from the starting address (line 10). `PROT_WRITE` is an argument for describing the desired memory protection of the mapping, where it indicates that pages may be written for the mapping. `MAP_SHARED` is a flag indicating that this mapping is to be *shared*, so that updates to the mapping are visible to other processes that map to the same memory device file. Again, one should check that the `mmap` call has succeeded (line 12), then cast the mapped pointer to the proper type to be used for reading/writing to the accelerator (line 14). Any mapping done with `mmap` should be *unmapped* with `munmap`, and it also is recommended to check that this operation has succeeded (line 18). Finally, the device file should also be closed (line 21).

Appendix B

The Arria V SoC Preloader Generation and Modification Procedures for Bare Metal Execution

This appendix describes the steps required to generate and modify the preloader, so that one can use the ARM HPS on the Arria V SoC FPGA in bare metal mode.

When using the *stock* preloader, the ARM boot sequence attempts to find the bootloader in the SD card, but using the SD card requires lots of user intervention, as described in the RocketBoard tutorial [92], and most importantly, the bare metal tutorial with the SD card does not work properly (also noted in the tutorial itself as with issues). Thus we needed to edit the preloader so that the ARM HPS can execute an application downloaded onto the HPS DDR3 memory in a much more streamlined fashion.

The preloader is generated by the bsp-editor, which is part of the Altera SoC EDS tool. First, download and install the SoC EDS tool from Altera. Once it is installed, run the `embedded_command_shell.sh` script inside the installation directory (altera/quartus_version/embedded/). Then open up bsp-editor, and click on File, New HPS BSP, then select the preloader directory, which is in our case is in:

`design_directory/Qsys_project_directory/hps_isw_handoff/legup_system_Arm_A9_HPS/`

Use the default settings for the preloader, but check *off* everything in spl.boot (BOOT_FROM_QSPI/

SDMMC/NAND/RAM), and *enable* checking SDRAM_SCRUBBING, then click Generate. Once it is generated, go to the software/spl_bsp directory and type `make`. This generates a bunch of the preloader code and some of the preloader code must be changed in order to get the baremetal ARM working. Special thanks to Kevin Nam at the Altera University Program for helping with this. The steps are described below.

1. Open ./software/spl_bsp/uboot-socfpga/arch/arm/cpu/armv7/socfpga/spl.c
   This file is generated inside the Quartus project directory where the
   bsp-editor was run.
2. Find where it says, "puts("\nNo boot device selected. Just loop here...\n");"
3. Before that line (also above the #if statements) put the following assembly code.

```
   // Switch to ARM mode then break
   asm volatile("mov r1,pc\n" : : );
   // This instruction is duplicated for padding purposes
   asm volatile("mov r1,pc\n" : : );
   asm volatile("bx r1\n" : : );
   asm volatile(".code 32\n" : : );
   asm volatile("stop:\n" : : );
   asm volatile("bkpt\n" : : );
   asm volatile("bkpt\n" : : );
```

   This will switch the processor from THUMB mode to ARM mode. The break instructions (bkpt) do not actually have any functionality, but are used later to find the point in the preloader where the preloader is done and the software application can start to execute.

4. Compile the preloader code again by first running "make clean" then "make"
inside ./software/spl_bsp, which will generate "u-boot-spl" inside
   software/spl_bsp/uboot-socfpga/spl
5. The "u-boot-spl" file needs to be converted to the binary format (.srec) that can be used by quartus_hps and the ARM processor. The conversion can be done by running the following:

```
   arm-altera-eabi-objcopy -O srec u-boot-spl u-boot-spl.srec
6. Now, "u-boot-spl" file needs to be disassembled to find the address of the "bkpt"
   instruction that was added above. Get the disassembly by running:
   arm-altera-eabi-objdump -D u-boot-spl > u-boot-spl.src
   Open u-boot-spl.src, and find the bkpt instruction. Get the address of the
   second bkpt instruction.
   This is the address we give as the --preloaderaddr
   argument to quartus_hps, the hps programmer.
```

We have now described all steps which are necessary to generate and modify the preloader. The ARM HPS can then be configured to execute a software binary with the following:

```
quartus_hps --cable=JTAG_CABLE -o GDBSERVER --gdbport0=PORT_NUM
--preloader=u-boot-spl.srec --preloaderaddr=BKPT_ADDR --source=SW_BINARY.srec
```

This invokes `quartus_hps` to use the `JTAG_CABLE` to download the preloader `u-boot-spl.srec` to the FPGA and execute it on the HPS. The value of `JTAG_CABLE` can be obtained by running `jtagconfig` (on DE1-SoC, we used `DE-SoC [3-4]`). `Quartus_hps` then stalls the HPS at the preloader address specified by `BKPT_ADDR`, which is the address found above from the preloader disassembly. To execute the software binary `SW_BINARY.srec`, one must connect remotely to the ARM HPS with `gdb` using the port specified by `PORT_NUM` to give the *continue* command. This allows the ARM HPS to start executing the software binary. In LegUp this is handled by a script, so that the process downloading the software binary and executing it on the HPS can be done with a single makefile target.

Appendix C

The Complete Benchmark Results for Chapter 4

This appendix presents the complete circuit-by-circuit results for Chapter 4 described in Section 4.5.2.

Table C.1: Benchmark results for the MIPS processor-only architecture (Arch. 0).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	209,972.96	26,920,633	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	465,722,121.52
Mandelbrot	37,393.44	4,794,213	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	82,939,024.85
Option Pricing	55,401.22	7,102,991	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	122,880,470.07
Gaussian Filter	54,243.15	6,954,514	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	120,311,844.61
Dfsin	113,031.20	14,491,730	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	250,704,329.28
Dfdiv	34,596.90	4,435,669	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	76,736,277.97
Geomean	67,221.74	8,618,499.64	128.21	9,629	12,444	6	76	1,339.35	325.94	2,218.01	149,098,497.70

Table C.2: Benchmark results for the MIPS single-threaded processor-accelerator hybrid architecture (Arch. 1).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	2,201.61	271,326	123.24	21,814	34,171	46	76	1,340.23	477.08	2,323.35	5,115,102.74
Mandelbrot	9,953.37	1,249,646	125.55	11,333	16,627	70	81	1,339.79	422.52	2,268.34	22,577,634.47
Option Pricing	2,229.01	284,243	127.52	12,575	20,015	33	79	1,340.00	449.32	2,295.37	5,116,396.29
Gaussian Filter	6,465.85	803,188	124.22	11,371	18,135	56	81	1,339.83	427.59	2,273.46	14,699,853.41
Dfsin	2,256.78	273,183	121.05	19,715	32,782	88	84	1,341.25	595.22	2,442.51	5,512,203.30
Dfdiv	2,025.95	259,666	128.17	14,383	24,058	54	80	1,339.97	445.01	2,291.03	4,641,512.02
Geomean	3,361.97	420,025.54	124.93	14,684	23,364	55	80	1,340.18	466.13	2,314.93	7,782,738.66

Table C.3: Benchmark results for the MIPS multi-threaded processor-accelerator hybrid architecture (Arch. 2).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	837.30	102,990	123	20,814	35,131	46	76	1,340.09	461.17	2,307.30	1,931,902.29
Mandelbrot	3,710.40	422,984	114	14,919	24,888	198	91	1,341.70	645.18	2,492.92	9,249,730.37
Option Pricing	785.60	98,198	125	19,244	35,253	87	85	1,342.18	703.65	2,551.86	2,004,741.22
Gaussian Filter	2,681.90	348,647	130	15,190	29,022	156	91	1,341.63	636.52	2,484.18	6,662,322.34
Dfsin	794.40	106,443	134	42,125	75,650	252	88	1,345.71	1112.1	2,963.84	2,354,474.50
Dfdiv	803.40	104,439	130	26,640	50,463	150	88	1,343.69	878.79	2,728.52	2,192,092.97
Geomean	1,269.07	159,685.16	125.83	21,611	38,806	129	86	1,342.50	711.96	2,579.92	3,274,099.39

Table C.4: Benchmark results for the MIPS multi-threaded and pipelined processor-accelerator hybrid architecture (Arch. 3p).

Benchmark	Time (us)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	169.19	20,010	118.27	19,242	28,542	46	289	1,345.04	954.16	2,805.24	474,616.15
Mandelbrot	339.57	40,619	119.62	17,813	35,284	198	321	1,346.09	1,058.88	2,911.01	988,482.82
Option Pricing	112.81	12,200	108.15	22,708	46,299	87	273	1,345.72	1,035.69	2,887.45	325,722.52
Gaussian Filter	665.81	66,015	99.15	23,233	53,313	156	168	1,344.22	907.47	2,757.74	1,836,129.16
Geomean	256.30	28,444.26	110.98	20,621.39	39,706.97	105.44	255.40	1,345.27	987.15	2,839.69	727,806.05

Table C.5: Benchmark results for the ARM processor-only architecture (Arch. 0).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	HPS Power	FPGA Power	Total Power (mW)	Energy (nJ)
Hash	2,158.58	2,266,507	1,050	1,650	2,551	0	2,048	1,666.20	334.30	2,000.50	4,318,235.48
Mandelbrot	3,013.07	3,163,725	1,050	1,650	2,551	0	2,048	1,789.61	354.83	2,144.43	6,461,335.71
Option Pricing	955.53	1,003,310	1,050	1,650	2,551	0	2,048	1,758.39	340.34	2,098.72	2,005,401.30
Gaussian Filter	3,738.87	3,925,811	1,050	1,650	2,551	0	2,048	1,813.55	330.53	2,144.08	8,016,425.45
Dfsin	387.93	407,323	1,050	1,650	2,551	0	2,048	1,762.32	341.60	2,103.92	816,165.96
Dfdiv	1,280.32	1,344,337	1,050	1,650	2,551	0	2,048	1,741.97	343.29	2,085.26	2,669,807.05
Geomean	1,503.28	1,578,439.36	1,050.00	1,650	2,551	0	2,048	1,754.73	340.73	2,095.59	3,150,248.68

Table C.6: Benchmark results for the ARM single-threaded processor-accelerator hybrid architecture (Arch. 1).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	HPS Power	FPGA Power	Total Power (mW)	Energy (nJ)
Hash	2,561.71	2,689,792	130	9,988	20,241	40	4	1,713.40	467.10	2,180.50	5,585,801.39
Mandelbrot	7,311.97	7,677,564	170	3,684	6,702	64	9	1,742.32	444.85	2,187.17	15,992,486.96
Option Pricing	1,978.35	2,077,272	140	5,031	10,094	27	7	1,788.20	482.90	2,271.10	4,493,040.42
Gaussian Filter	8,708.97	9,144,415	150	3,870	8,339	50	9	1,774.40	417.27	2,191.67	19,087,177.13
Dfsin	2,142.86	2,249,999	130	11,504	22,821	82	8	1,649.57	462.01	2,111.58	4,524,822.68
Dfdiv	1,831.15	1,922,704	130	6,812	14,089	48	8	1,772.54	414.99	2,187.54	4,005,702.19
Geomean	3,289.21	3,453,671.54	140.96	6,188	12,420	49	7	1,739.42	447.47	2,187.77	7,196,046.44

Table C.7: Benchmark results for the ARM multi-threaded processor-accelerator hybrid architecture (Arch. 2).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	HPS Power	FPGA Power	Total Power (mW)	Energy (nJ)
Hash	907.47	952,841	125	10,063	20,090	40	4	1,679.34	472.21	2,151.55	1,952,462.59
Mandelbrot	3,896.76	4,091,593	120	11,730	31,003	192	25	1,801.19	534.02	2,335.21	9,099,761.20
Option Pricing	660.44	693,459	140	12,193	25,827	81	13	1,665.28	534.42	2,199.70	1,452,761.09
Gaussian Filter	4,300.61	4,515,640	125	8,184	19,775	150	19	1,860.57	508.54	2,369.11	10,188,635.46
Dfsin	1,537.84	1,614,728	120	32,649	65,641	246	13	1,789.99	694.10	2,484.09	3,820,120.22
Dfdiv	1,408.95	1,479,394	125	18,788	40,845	144	16	1,742.71	622.55	2,365.26	3,332,531.83
Geomean	1,670.90	1,754,443.41	125.66	13,904	30,795	122	13	1,755.17	556.22	2,314.79	3,867,785.49

Table C.8: Benchmark results for the ARM multi-threaded and pipelined processor-accelerator hybrid architecture (Arch. 3p).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	HPS Power	FPGA Power	Total Power (mW)	Energy (nJ)
Hash	40.80	42,845	130	15,477	24,069	40	794,366	1,756.87	665.86	2,422.73	98,858.75
Mandelbrot	139.18	146,143	160	12,143	28,022	192	562,916	1,779.02	757.58	2,536.60	353,053.05
Option Pricing	25.39	26,659	140	16,591	38,283	81	229,022	1,662.20	658.10	2,320.30	58,911.31
Gaussian Filter	163.52	171,691	150	18,518	48,699	150	699,338	1,877.58	795.78	2,673.36	437,135.15
Geomean	69.68	73,167.36	144.57	15,501.36	33,486.58	98.28	517,317.47	1,767.26	716.93	2,484.79	173,147.96

Appendix D

The Complete Benchmark Results for Chapter 5

This appendix presents the complete circuit-by-circuit results for Chapter 5 described in Section 5.3.1.

Table D.1: Benchmark results for the single-threaded hardware-only system (Arch. 1).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	1,606.51	227,353	141.52	7,356	10,098	40	51	1,329.71	167.31	1,502.31	2,413,472.90
Mandelbrot	6,322.08	1,228,569	194.33	1,626	3,506	64	4	1,328.45	30.28	1,364.01	8,623,374.68
Option Pricing	1,737.86	273,417	157.33	3,132	6,569	27	3	1,328.42	27.79	1,361.50	2,366,091.94
Gaussian Filter	2,878.62	706,587	245.46	1,837	5,244	50	4	1,328.51	30.82	1,364.62	3,928,227.62
Dfsin	1,764.88	248,248	140.66	10,309	19,689	82	4	1,328.69	58.71	1,392.69	2,457,930.52
Dfdiv	862.44	121,233	140.57	5,207	10,812	48	4	1,328.74	49.57	1,383.59	1,193,261.48
Geomean	2,064.10	342,846.46	166.10	3,931.72	7,987.11	48.86	5.83	1,328.75	48.26	1,393.94	2,877,236.28

Table D.2: Benchmark results for the multi-threaded hardware-only system (Arch. 2).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	542.06	76,821	141.72	7,151	14,670	40	1	1,329.48	159.97	1,494.73	810,236.05
Mandelbrot	2,150.06	409,543	190.48	5,366	11,494	192	12	1,328.91	84.21	1,418.40	3,049,641.91
Option Pricing	677.45	92,180	136.07	9,778	21,061	81	9	1,328.83	75.62	1,409.73	955,015.15
Gaussian Filter	1,243.62	278,335	223.81	6,624	18,603	150	20	1,328.92	74.62	1,408.82	1,752,039.30
Dfsin	801.22	99,263	123.89	35,507	66,849	246	10	1,329.60	169.12	1,504.00	1,205,033.11
Dfdiv	314.67	40,825	129.74	16,158	30,972	144	56	1,329.70	146.49	1,481.47	466,170.90
Geomean	792.40	121,838.05	153.76	10,609.17	22,698.89	122.05	10.32	1,329.24	111.13	1,452.28	1,150,782.13

Table D.3: Benchmark results for the multi-threaded and pipelined hardware-only system (Arch. 3p).

Benchmark	Time (μs)	Cycles	Fmax	ALMs	Registers	DSPs	M10Ks	Static Power	Dyn. Power	Total Power (mW)	Energy (nJ)
Hash	21.97	3,155	144	8,246	17,490	40	3	1,328.98	93.41	1,427.67	31,366.98
Mandelbrot	52.50	9,695	185	6,699	16,813	192	17	1,329.77	188.81	1,523.87	80,001.73
Option Pricing	7.43	1,124	151	11,986	30,708	81	57	1,330.02	202.98	1,538.31	11,432.56
Gaussian Filter	26.59	6,330	238	11,826	38,305	150	30	1,329.85	182.44	1,517.58	40,355.74
Geomean	21.85	3,840.87	175.78	9,406.79	24,251.33	98.28	17.18	1,329.65	159.86	1,501.21	32,802.37

Appendix E

The Complete Benchmark Results for Chapter 6

This appendix presents the complete circuit-by-circuit results for Chapter 6 described in Section 6.6.2.

Table E.1: Benchmark results for Arch. 1.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	55.7	10,090	181	1,474	0	46
Barrier	326.8	64,052	196	1,076	0	12
MCML	2,895.9	474,924	164	18,752	388	76
Alphablend	10.6	2,038	193	1,883	0	18
Black Scholes	556.8	99,662	179	20,082	260	76
Box Filter	800.6	161,716	202	2,581	48	34
DF	384.3	59,186	154	22,531	142	68
Division	529.6	90,034	170	4,556	0	140
Dot Product	48.3	9,036	187	1,607	8	44
Hash	504.3	87,756	174	3,380	14	28
Line of Sight	1,085.2	207,276	191	4,362	8	26
Mandelbrot	4,328.9	839,810	194	2,792	160	20
Matrix Multiply	16.8	2,054	122	2,360	44	16
Histogram	369.4	49,504	134	4,398	0	97
Vector Add	54.9	10,040	183	1,823	0	46
Geomean	262.86	45,562.53	173.34	3,727.72	11.03	39.38

Table E.2: Benchmark results for Arch. 2.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	47.8	10,090	211	1,247	0	46
Barrier	297.9	64,052	215	1,012	0	12
MCML	2,810.2	474,924	169	21,827	388	76
Alphablend	10.0	2,040	204	1,746	0	18
Black Scholes	559.9	99,660	178	20,056	260	76
Box Filter	804.6	161,716	201	2,500	48	34
DF	391.8	59,158	151	22,384	142	68
Division	481.5	90,036	187	4,763	0	140
Dot Product	48.6	9,036	186	1,475	8	44
Hash	464.4	87,764	189	3,086	14	28
Line of Sight	1,050.9	207,022	197	4,142	8	26
Mandelbrot	4,396.9	839,810	191	2,698	160	20
Matrix Multiply	17.0	2,060	121	2,199	44	16
Histogram	330.3	49,538	150	4,071	0	97
Vector Add	47.8	10,038	210	1,620	0	46
Geomean	250.37	45,570.97	182.00	3,563.40	11.03	39.38

Table E.3: Benchmark results for Arch. 3.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	47.8	10,090	211	1,247	0	46
Barrier	297.9	64,052	215	1,012	0	12
MCML	3,025.0	474,924	157	11,144	388	35
Alphablend	10.0	2,040	204	1,746	0	18
Black Scholes	600.4	99,662	166	9,436	260	35
Box Filter	804.6	161,716	201	2,500	48	34
DF	386.7	59,160	153	21,462	142	113
Division	466.5	90,036	193	2,473	0	92
Dot Product	48.6	9,036	186	1,475	8	44
Hash	525.5	87,764	167	1,993	14	28
Line of Sight	1,050.9	207,022	197	4,142	8	26
Mandelbrot	4,396.9	839,810	191	2,698	160	20
Matrix Multiply	17.0	2,060	121	2,199	44	16
Histogram	330.3	49,538	150	4,071	0	97
Vector Add	47.8	10,038	210	1,620	0	46
Geomean	254.10	45,571.14	179.33	3,004.11	11.03	35.72

Table E.4: Benchmark results for Arch. 4.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	47.8	10,090	211	1,247	0	46
Barrier	297.9	64,052	215	1,012	0	12
MCML	2,913.6	474,924	163	13,786	97	35
Alphablend	10.0	2,040	204	1,746	0	18
Black Scholes	579.5	99,666	172	10,779	65	35
Box Filter	812.6	161,716	199	3,234	8	34
DF	389.2	59,164	152	22,320	44	113
Division	466.5	90,036	193	2,473	0	92
Dot Product	46.8	9,036	193	1,631	2	44
Hash	504.4	87,764	174	2,149	10	28
Line of Sight	1,145.6	207,360	181	4,264	2	26
Mandelbrot	4,263.0	839,814	197	2,932	40	20
Matrix Multiply	16.9	2,060	122	2,785	20	16
Histogram	330.3	49,538	150	4,071	0	97
Vector Add	47.8	10,038	210	1,620	0	46
Geomean	252.65	45,576.44	180.39	3,247.12	5.29	35.72

Table E.5: Benchmark results for Arch. 5.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	42.6	10,044	236	1,236	0	44
Barrier	61.9	16,038	259	748	0	10
MCML	2,845.2	472,304	166	13,379	97	32
Alphablend	6.2	1,782	289	1,337	0	12
Black Scholes	569.5	99,658	175	9,184	65	30
Box Filter	301.4	79,878	265	2,972	8	26
DF	339.2	59,016	174	21,235	44	100
Division	326.2	90,030	276	2,273	0	90
Dot Product	31.6	9,032	286	1,444	2	42
Hash	348.0	82,824	238	1,848	10	26
Line of Sight	794.1	207,266	261	4,004	2	17
Mandelbrot	4,263.0	839,810	197	2,505	40	10
Matrix Multiply	10.0	2,042	205	2,170	12	14
Histogram	235.7	49,500	210	4,307	0	94
Vector Add	37.9	10,034	265	1,467	0	43
Geomean	170.30	39,079.69	229.66	2,885.29	5.11	29.72

Table E.6: Benchmark results for Arch. 6.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	38.3	10,036	262	1,010	0	36
Barrier	65.4	20,016	306	563	0	2
MCML	2,740.2	471,322	172	13,194	97	26
Alphablend	4.9	1,532	312	1,270	0	6
Black Scholes	533.6	99,254	186	9,001	65	24
Box Filter	233.6	61,912	265	2,872	8	17
DF	341.7	57,740	169	21,092	44	94
Division	338.0	87,530	259	2,195	0	84
Dot Product	27.6	7,530	273	1,365	2	36
Hash	425.9	79,224	186	1,758	10	20
Line of Sight	703.0	191,210	272	4,011	2	11
Mandelbrot	4,220.1	839,808	199	2,443	40	4
Matrix Multiply	8.0	1,650	207	2,112	12	8
Histogram	217.2	44,082	203	3,365	0	75
Vector Add	41.1	10,030	244	1,387	0	35
Geomean	160.48	36,858.47	229.71	2,680.04	5.11	19.59

Table E.7: Benchmark results for Arch. 7.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	31.8	7,536	237	879	0	132
Barrier	65.4	20,016	306	563	0	2
MCML	3,184.6	471,322	148	13,075	97	38
Alphablend	4.9	1,532	312	1,270	0	6
Black Scholes	522.4	99,254	190	8,986	65	30
Box Filter	239.0	61,912	259	2,507	8	92
DF	387.5	57,740	149	20,924	44	105
Division	338.0	87,530	259	2,195	0	84
Dot Product	28.6	7,530	263	1,174	2	132
Hash	425.9	79,224	186	1,758	10	20
Line of Sight	703.0	191,210	272	4,011	2	11
Mandelbrot	4,220.1	839,808	199	2,443	40	4
Matrix Multiply	7.8	1,640	211	2,051	24	20
Histogram	219.0	40,956	187	3,061	0	430
Vector Add	24.3	6,034	248	1,208	0	159
Geomean	156.10	34,771.55	222.74	2,556.74	5.35	36.12

Table E.8: Benchmark results for Arch. 8.

Benchmark	**Time (μs)**	**Cycles**	**Fmax (MHz)**	**Logic Util.**	**DSPs**	**M20Ks**
Mutex	31.8	7,536	237	879	0	132
Barrier	70.2	20,016	285	568	0	2
MCML	3,163.2	471,322	149	10,300	388	38
Alphablend	4.8	1,532	322	1,263	0	6
Black Scholes	506.4	99,254	196	7,585	260	30
Box Filter	197.8	61,912	313	1,746	48	92
DF	395.5	57,742	146	20,142	142	105
Division	324.2	87,530	270	2,190	0	84
Dot Product	26.8	7,530	281	1,011	8	132
Hash	430.6	79,224	184	1,583	14	20
Line of Sight	667.5	190,894	286	3,860	8	11
Mandelbrot	4,351.3	839,806	193	2,202	160	4
Matrix Multiply	8.0	1,640	205	1,388	48	20
Histogram	219.0	40,956	187	3,061	0	430
Vector Add	24.3	6,034	248	1,208	0	159
Geomean	153.54	34,767.79	226.56	2,299.00	11.09	36.12

Appendix F

Code Examples for Creating Streaming Hardware with LegUp

This appendix provides code examples that illustrate how one can use LegUp to create the different streaming hardware architectures described in Section 7.4.3.

1. Creating streaming hardware modules with two kernels, `A` and `B`, which are connected together via two FIFOs, `fifo0` and `fifo1`. The data flows from `A` to `B`. This is similar to the architecture shown in Figure 7.2a.

```
// create the FIFOs
PTHREAD_FIFO *fifo0 = pthread_fifo_malloc(/*width*/32, /*depth*/2);
PTHREAD_FIFO *fifo1 = pthread_fifo_malloc(/*width*/32, /*depth*/2);

// store FIFOs in a struct (Pthreads require multiple arguments to be passed in as a struct)
// kernel_data is a struct type that we assume to have defined already
kernel_data data;
data.fifo0 = fifo0;
data.fifo1 = fifo1;

// fork the kernels
pthread_t threadA, threadB;
pthread_create(&threadA, NULL, A, (void *)&data);
pthread_create(&threadB, NULL, B, (void *)&data);
```

2. Creating *replicated* streaming hardware modules with two kernels, `A` and `B`, where each kernel executes on two threads. `Fifo0` and `fifo1` are used to connect the first instance of `A` and `B`, and `fifo2` and `fifo3` are used to connect the second instance of `A` and `B`. The data flows from the first instance of `A` to the first instance of `B`, and also flows from the second instance of `A` to the second instance of `B`. This is similar to the architecture shown in Figure 7.2b

```
#define NUM_FIFOS 4
#define NUM_INSTANCES 2
// create the FIFOs
PTHREAD_FIFO *fifo[NUM_FIFOS];
for (i=0; i<NUM_FIFOS; ++i) {
  fifo[i] = pthread_fifo_malloc(/*width*/32, /*depth*/2);
}

// store FIFOs in structs
kernel_data data[NUM_INSTANCES];
data[0].fifo0 = fifo[0];
data[0].fifo1 = fifo[1];
data[1].fifo0 = fifo[2];
data[1].fifo1 = fifo[3];

// fork the kernels
pthread_t threadA[NUM_INSTANCES], threadB[NUM_INSTANCES];
for (i=0; i<NUM_INSTANCES; ++i) {
  pthread_create(&threadA[i], NULL, A, (void *)&data[i]);
  pthread_create(&threadB[i], NULL, B, (void *)&data[i]);
}
```

3. Creating a one-to-many architecture with three kernels, `A`, `B`, and `C`, which are connected via a common FIFO, `fifo`. The data flows from `A` to `fifo`, which can be accessed by both `B` and `C`. This is similar to the architecture shown in Figure 7.2c. Here we also show a code snippet of the kernel functions.

```
void top() {
```

```
  ...
  // create the FIFO
  PTHREAD_FIFO *fifo = pthread_fifo_malloc(/*width*/32, /*depth*/2);

  // fork the kernels
  pthread_t threadA, threadB, threadC;
  pthread_create(&threadA, NULL, A, (void *)&fifo);
  pthread_create(&threadB, NULL, B, (void *)&fifo);
  pthread_create(&threadC, NULL, C, (void *)&fifo);
}

// kernel A, which writes to the FIFO
void *A(void *arg) {
  ...
  while (1) {
    ...
    // write data to the fifo that was passed in as the argument
    pthread_fifo_write(fifo, data);
  }
}

// kernel B, which reads from the FIFO
void *B(void *arg) {
  ...
  while (1) {
    // read data from the fifo that was passed in as the argument
    data = pthread_fifo_read(fifo);
    ...
  }
}

// kernel C, which also reads from the FIFO
void *C(void *arg) {
```

```
  ...
  while (1) {
    // read data from the fifo that was passed in as the argument
    data = pthread_fifo_read(fifo);
    ...
  }
}
```

LegUp automatically determines which kernel *writes* to the FIFO, or *reads* from the FIFO, and automatically create arbitration logic as necessary. In this example, LegUp finds that `A` writes to the FIFO, and both `B` and `C` reads from the FIFO, hence an arbiter is created to handling *read* contentions from `B` and `C`.

4. Creating a many-to-one architecture with three kernels, `A`, `B`, and `C`, which are connected via a common FIFO, `fifo`. The data flows from both `A` and `B` to `fifo`, which can be accessed by `C`. This is similar to the architecture shown in Figure 7.2d. Here we also show a code snippet of the kernel functions.

```
void top() {
  ...
  // create the FIFO
  PTHREAD_FIFO *fifo = pthread_fifo_malloc(/*width*/32, /*depth*/2);

  // fork the kernels
  pthread_t threadA, threadB, threadC;
  pthread_create(&threadA, NULL, A, (void *)&fifo);
  pthread_create(&threadB, NULL, B, (void *)&fifo);
  pthread_create(&threadC, NULL, C, (void *)&fifo);
}

// kernel A, which writes to the FIFO
void *A(void *arg) {
  ...
  while (1) {
    ...
```

```
    // write data to the fifo that was passed in as the argument
    pthread_fifo_write(fifo, data);
  }
}

// kernel B, which writes to the FIFO
void *B(void *arg) {
  ...
  while (1) {
    ...
    // write data to the fifo that was passed in as the argument
    pthread_fifo_write(fifo, data);
  }
}

// kernel C, which reads from the FIFO
void *C(void *arg) {
  ...
  while (1) {
    // read data from the fifo that was passed in as the argument
    data = pthread_fifo_read(fifo);
    ...
  }
}
```

In this example, LegUp finds that `A` and `B` write to the FIFO, and `C` reads from the FIFO, hence an arbiter is created to handling *write* contentions from `A` and `B`.

Bibliography

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL*, IWOCL '14, pages 4:1–4:9, New York, NY, USA, 2014. ACM.

[2] Algorithm and Programming. *Box Filtering. (http://tech-algorithm.com/articles/boxfiltering/).*

[3] Altera, Corp., San Jose, CA. *TriMatrix Embedded Memory Blocks in Stratix IV Devices*, 2011.

[4] Altera, Corp., San Jose, CA. *Avalon Interface Specifications*, 2015.

[5] L. O. Andersen. Program analysis and specialization for the c programming language. In *Ph.D. Thesis*. University of Cophenhagen, 1994.

[6] E. Anderson et al. Enabling a uniform programming model across the software/hardware boundary. In *IEEE FCCM*, pages 89–98, April 2006.

[7] B. Barney. *POSIX Threads Programming*. Lawrence Livermore National Laboratory, Auguest 13, 2015.

[8] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 1965.

[9] A. Canis, S.D. Brown, and J.H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *IEEE FPL*, pages 1–8, Sept. 2014.

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

[11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, , and Jason H. Anderson. Legup: An open source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM TECS*, 2012.

[12] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679 – 698, November 1986.

[13] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, Jan 2016.

[14] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *IEEE FPT*, pages 270–277, December 2013.

[15] J. Choi, S. Brown, and J. Anderson. Resource and memory management techniques for performance and area of parallel hardware in high-level synthesis for fpgas. In *IEEE FPT*, December 2015.

[16] J. Choi, R. Lian, S. Brown, and J. Anderson. A unified software approach to specify pipelineand spatial parallelism in fpga hardware. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2016.

[17] J. Choi, K. Nam, A. Canis, J.H. Anderson, S.D. Brown, and T. Czajkowski. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *IEEE FCCM*, pages 17–24, 2012.

[18] Jongsok Choi, Stephen Brown, and Jason Anderson. Enabling hardware/software co-design in high-level synthesis. In *M.A.Sc dissertation, Electrical and Computer Engineering, University of Toronto*, Toronto, Canada, Nov. 2012.

[19] J. Cong and W. Jiang. Pattern-based behavior synthesis for fpga resource reduction. In *ACM/SIGDA FPGA*, pages 107–116, 2008.

[20] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *ACM DAC*, volume 43, pages 433–438, 2006.

[21] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):1–29, 2009.

[22] Cortex-A9 Technical Reference Manual (Revision: r3p0). *Chapter 11: Performance Monitoring Unit.*

[23] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin. GAUT a free and open source high-level synthesis tool. In *IEEE DATE*, 2010.

[24] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-enabled loop pipelining for high-level synthesis. In *DAC*, pages 1–6, San Francisco, CA, June 2014.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 365–376, 2011.

[27] A. Cilardo et. al. Efficient and scalable OpenMP-based system-level design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 988 – 991, 2013.

[28] A. Papakonstantinou et. al. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *IEEE Symposium on Application Specific Processors*, pages 35–42, 2009.

[29] D. Cabrera et. al. OpenMP extensions for FPGA accelerators. In *IEEE Systems, Architecture, Modeling and Simulation (SAMOS)*, pages 17–24, 2009.

[30] J. Cong et. al. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[31] Y.Y Leow et. al. Generating hardware from OpenMP programs. In *IEEE FPT*, pages 73–80, 2006.

[32] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 120–129, Aug 2014.

[33] M. Gort and J. H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 773–779, Jan 2013.

[34] S. Hadjis, A. Canis, J.H. Anderson, J. Choi, K. Nam, S.D. Brown, and T. Czajkowski. Impact of FPGA architecture on resource sharing in high-level synthesis. In *ACM/SIGDA FPGA*, pages 111–114, 2012.

[35] Robert J. Halstead and Walid Najjar. Compiled multithreaded data paths on fpgas for dynamic workloads. In *IEEE Compilers, Architectures and Synthesis for Embedded Systems*, 2013.

[36] D. Hansen, J. Li, S. Balatsos, and X. Liu. *Calypto White Paper: Designing ASIC IP at Higher Level of Abstraction*. `https://www.mentor.com/hls-lp/success/qualcomm-inc`.

[37] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CH-Stone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[38] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S.D. Brown, and J.H. Anderson. The effect of compiler optimizations on high-level synthesis for FPGAs. In *IEEE FCCM*, pages 89–96, Seattle, WA, 2013.

[39] A. Ismail et al. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *IEEE FCCM*, pages 170–177, May 2011.

[40] D. Ku and G. D. Micheli. *High Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Norwell, MA, 1992.

[41] C. E. LaForest, J. Anderson, and J. G. Steffan. Approaching overhead-free execution on fpga soft-processors. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 99–106, Dec 2014.

[42] Jaehwan Lee. Hardware/software deadlock avoidance for multiprocessor multiresource system-on-a-chip. In *Ph.D dissertation, School of Electrical and Computer Engineering, Georgia Institute of Technology*, Toronto, Canada, Nov. 2004.

[43] Jaehwan Lee and Vincent John Mooney. A novel deadlock avoidance algorithm and its hardware implementation. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, pages 200–205, 2004.

[44] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. Resource-aware throughput optimization for high-level synthesis. In *ACM/SIGDA FPGA*, pages 200–209, 2015.

[45] W. Lo et al. Hardware acceleration of a monte carlo simulation for pdt treatment planning. *J. Biomed. Opt.*, 14(1), Jan-Feb 2009.

[46] E. Lubbers and M. Platzner. A portable abstraction layer for hardware threads. In *IEEE FPL*, pages 17–22, Sept 2008.

[47] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press.*, pages 281 – 297, 1967.

[48] A. Martin, D. Jamsek, and K. Agarwal. FPGA-Based Application Acceleration: Case Study with GZIP Compression/Decompression Streaming Engine. In *Interational Conference on Computer-Aided Design (ICCAD)*, Nov. 2013.

[49] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[50] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen, and R. Krishnamurthy. 340 mv x2013;1.1 v, 289 gbps/w, 2090-gate nanoaes hardware accelerator with area-optimized encrypt/decrypt gf(2 4) 2 polynomials in 22 nm tri-gate cmos. *IEEE Journal of Solid-State Circuits*, 50(4):1048–1058, April 2015.

[51] Microsoft, `http://download.microsoft.com/download/8/2/9/8297F7C7-AE81-4E99-B1DB-D65A01F7A8EF/Microsoft_Cloud_Infrastructure_Datacenter_and_Network_Fact_Sheet.pdf`. *Microsoft's Cloud Infrastructure: Datacenters and Network Fact Sheet*, June, 2015.

[52] S. Mochizuki, K. Matsubara, K. Matsumoto, C. L. P. Nguyen, T. Shibayama, K. Iwata, K. Mizumoto, T. Irita, H. Hara, and T. Hattori. 4.4 a 197mw 70ms-latency full-hd 12-channel video-processing soc for car information systems. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 78–79, Jan 2016.

[53] C. Moore. Data processing in exascale-class computing systems. In *Salishan Conference on High Speed Computing*, April, 27 2011.

[54] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 19 1965.

[55] A. Morvan, S. Derrien, and P. Quinton. Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion. In *IEEE FPT*, pages 1–10, Dec. 2011.

[56] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y.T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, PP(99), December 2015.

[57] R. Nane, V.M. Sima, J. van Someren, and K.L.M. Bertels. Dwarv: A hdl compiler with support for scheduling custom ip blocks. In *DAC*, San Diego, USA, June 2011.

[58] R. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *In IEEE/ACM MEMOCODE*, pages 69–70, 2004.

[59] The Open Group. *POSIX.1 FAQ*, October 5, 2011.

[60] Oregon Medical Laser Center. *Monte Carlo Simulations. (http://omlc.ogi.edu/software/mc/)*, 2007.

[61] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*, pages 1–4, 2013.

[62] Fred J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[63] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[64] R. Rashid, J. G. Steffan, and V. Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 20–27, Dec 2014.

[65] D. Salomon, G. Motta, and D. Bryant. *Data Compression: The Complete Reference*. Springer, molecular biology intelligence unit edition, 1992.

[66] L. Semeria and G. De Micheli. Spc: synthesis of pointers in c application of pointer analysis to the behavioral synthesis from c. In *IEEE/ACM ICCAD 98. Digest of Technical Papers.*, pages 340–346, November 1998.

[67] B. Steensgaard. Points-to analysis in almost linear time. In *ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 32–41, 1996.

[68] G. Stitt and F. Vahid. Thread warping: a framework for dynamic synthesis of thread accelerators. In *IEEE/ACM CODES+ISSS*, pages 93–98, 2007.

[69] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded pipeline synthesis for data-parallel kernels. *ICCAD*, 2014.

[70] United States Bureau of Labor Statistics. *Occupational Outlook Handbook 2010-2011 Edition*, 2010.

[71] University of Cambridge, `http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html`. *The Tiger "MIPS" processor.*, 2010.

[72] `http://dragonegg.llvm.org/`. *DragonEgg - Using LLVM as a GCC backend.*

[73] `http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf`. *Nvidia Tesla GPU Accelerators.*

[74] `http://legup.eecg.utoronto.ca/wiki/doku.php?id=running_linux_os_on_arm-based_soc_fpgas`. *LegUp wiki: Linux OS on ARM SoC FPGAs.*

[75] `http://linuxcommand.org/man_pages/taskset1.html`. *Linux Users Manual : TASKSET.*

[76] `http://linux.die.net/man/8/picocom`. *picocom(8) - Linux man page.*

[77] `http://llvm.org/`. *The LLVM Compiler Infrastructure.*

[78] `http://llvm.org/docs/Passes.html`. *LLVMs Analysis and Transform Passes*, 2015.

[79] `http://man7.org/linux/man-pages/man2/mmap.2.html`. *Linux Programmer's Manual : MMAP.*

[80] `http://man7.org/linux/man-pages/man7/sem_overview.7.html`. *Linux Programmer's Manual.*

[81] `http://manpages.ubuntu.com/manpages/xenial/man8/turbostat.8.html`. *Ubuntu Manuals : turbostat.*

[82] `http://news.synopsys.com/index.php?item=123168`. *Synopsys Acquires High-level Synthesis Technology from Synfora, Inc.*

[83] `http://nvidianews.nvidia.com/news/nvidia-delivers-massive-performance-leap-for-deep-learning-hpc-applications-with-nvidia-tesla-p100-accelerators`. *NVIDIA Delivers Massive Performance Leap for Deep Learning, HPC Applications With NVIDIA Tesla P100 Accelerators*, April 5 2013.

[84] http://openmp.llvm.org/. *OpenMP: Support for the OpenMP language.*

[85] http://openmp.org/wp/. *The OpenMP API specification for parallel programming.*

[86] https://community.cadence.com/cadence_blogs_8/b/ii/archive/2014/02/18/cadence-acquisition-of-forte-boosts-high-level-synthesis. *How Cadence Acquisition of Forte Boosts High-Level Synthesis.*

[87] https://docs.oracle.com/cd/E19455-01/806-5257/sync-31/index.html. *Oracle: Multithreaded Programming Guide.*

[88] https://gcc.gnu.org/onlinedocs/gcc-4.2.4/libgomp/Implementing-PARALLEL-construct.html. *Implementing PARALLEL construct.*

[89] https://github.com/grievejia/andersen. *Andersen's inclusion-based pointer analysis re-implementation in LLVM.*

[90] https://msdn.microsoft.com/en-us/library/hh872235.aspx. *Auto-Parallelization and Auto-Vectorization.*

[91] https://rocketboards.org/. *RocketBoards.org.*

[92] https://rocketboards.org/foswiki/view/Documentation/SoCSWWS1IntroToAlteraSoCDevicesLab3BareMetalApplication. *SoC SW WS1: Intro To Altera SoC Devices - Lab 3 - Bare Metal Application.*

[93] https://software.intel.com/sites/default/files/m/d/4/1/d/8/4-1-ProgTools_-_Automatic_Parallelization_with_Intel_C2_AE_Compilers.pdf. *Automatic Parallelization with Intel Compilers.*

[94] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/backgrounder/spectra-q-engine-backgrounder.pdf. *Spectra-Q Engine.*

[95] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_c5soc_getting_started.pdf. *Altera SDK for OpenCL Cyclone V SoC Getting Started Guide.*

[96] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/ug_aocl_c5soc_devkit_platform.pdf. *Altera SDK for OpenCL: Cyclone V SoC Development Kit Reference Platform Porting Guide.*

[97] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_av_soc_dev_board.pdf. *Arria V SoC Development Board Reference Manual.*

[98] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_av_soc_dev_kit.pdf. *Arria V SoC Development Kit User Guide.*

[99] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf. *Embedded Peripherals IP User Guide.*

[100] https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. *Altera SDK for OpenCL.*

[101] https://www.altera.com/products/design-software/embedded-software-developers/soc-eds/ds-5-toolkit.html. *ARM DS-5 Altera Edition.*

[102] https://www.altera.com/products/design-software/embedded-software-developers/soc-eds/overview.html. *Altera's SoC Embedded Design Suite (EDS).*

[103] https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html. *Qsys - Alteras System Integration Tool.*

[104] https://www.altera.com/products/fpga/stratix-series/stratix-v/overview.highResolutionDisplay.html. *Stratix V FPGAs.*

[105] https://www.altera.com/products/processors/overview.html. *Nios II Processor: The World's Most Versatile Embedded Processor.*

[106] https://www.altera.com/products/soc/overview.html. *Altera SoC Overview.*

[107] https://www.altera.com/products/soc/portfolio/arria-v-soc/overview.html. *Altera Arria V SoC.*

[108] https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html. *Monte Carlo Pricing of Asian Options on FPGAs Using OpenCL.*

[109] https://www.arm.com/products/system-ip/amba-specifications.php. *AMBA Specifications.*

[110] https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html. *Stratus High-Level Synthesis.*

[111] https://www.khronos.org/opencl/. *The open standard for parallel programming of heterogeneous systems.*

[112] https://www.maxeler.com/products/software/maxcompiler/. *MaxCompiler.*

[113] https://www.mentor.com/company/news/mentor-acquires-calypto-design-systems. *Mentor Graphics Acquires Calypto Design Systems.*

[114] https://www.mentor.com/hls-lp/catapult-high-level-synthesis/. *Catapult High-Level Synthesis.*

[115] https://www.mentor.com/hls-lp/success/google-inc. *Google Develops WebM Video Decompression Hardware IP Using Technology Independent Sources and High-Level Synthesis.*

[116] https://www.mentor.com/products/fpga/handel-c/dk-design-suite/. *DK Design Suite.*

[117] https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx. *Synphony C Compiler.*

[118] http://www.ace.nl. *ACE - Associated Compiler Experts. CoSy.*

[119] http://www.bdti.com/InsideDSP/2011/02/28/Xilinx. *Xilinx Buys AutoESL, Securing High-Level Synthesis Capabilities.*

[120] http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot. *BDTI Certified Results for the AutoESL AutoPilot High-Level Synthesis Tool.*

[121] http://www.bluespec.com. *Bluespec: The Synthesizable Modeling Company.*

[122] http://www.csee.wvu.edu/~jdm/classes/cs550/notes/tech/mutex/pc-sem.html. *A Semaphore Solution to the Producer-Consumer Problem.*

[123] http://www.impulseaccelerated.com. *Impulse CoDeveloper – Impulse accelerated technologies.*

[124] http://www.intc.com/releasedetail.cfm?ReleaseID=915707. *Intel to Acquire Altera.*

[125] http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html. *Intel Hyper-Threading Technology.*

[126] http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html. *Intel Xeon Phi Product Family: Product Brief.*

[127] http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html. *The Story of the Intel 4004: Intel's First Microprocessor.*

[128] http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html. *Intel Xeon Processor E7 Family.*

[129] http://www.intel.com/content/www/us/en/support/processors/000005523.html. *Intel Turbo Boost Technology Frequency Table.*

[130] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html. *Mersenne Twister Home Page.*

[131] http://www.nec.com/. *NEC: Orchestrating a brighter world.*

[132] http://www.nec.com/en/global/prod/cwb/index.html. *Cyberworkbench: Pioneering C-based LSI Design.*

[133] http://www.nvidia.ca/object/cuda_home_new.html. *CUDA Parallel Computing Platform.*

[134] http://www.prnewswire.com/news-releases/cadence-announces-stratus-high-level-synthesis-platform-300038873.html. *Cadence Announces Stratus High-Level Synthesis Platform.*

[135] http://www.xilinx.com/products/design-tools/microblaze.html. *MicroBlaze Soft Processor Core.*

[136] http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. *SDAccel Development Environment.*

[137] http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html. *SDSoC Development Environment.*

[138] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. *Vivado High-Level Synthesis.*

[139] http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html. *Virtex-7.*

[140] http://www.xilinx.com/products/silicon-devices/soc.html. *Expanding the All Programmable SoC Portfolio.*

[141] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable. *Zynq-7000 All Programmable SoC.*

[142] `http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf`. *Xilinx: Xcell Journal, Issue 86*, 2014.

[143] `http://www.xilinx.com/publications/prod_mktg/vivado/Vivado_IP_Integrator_Backgrounder.pdf`. *Vivado IP Integrator: Accelerated Time to IP Creation and Integration*.

[144] `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug1027-intro-to-sdsoc.pdf`. *SDSoC Environment User Guide*, 2015.

[145] `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug1185-sdsoc-release-notes.pdf`. *SDSoC Development Environment Release Notes*, July 26, 2015.

[146] `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug902-vivado-high-level-synthesis.pdf`. *Xilinx: Vivado Design Suite User Guide - High-Level Synthesis*, November 2015.

[147] `www.intel.com/software/pcm`. *Intel Performance Counter Monitor - A better way to measure CPU utilization*.

[148] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 281–294, Berkeley, CA, USA, 2008.

[149] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 5–14, New York, NY, USA, 2011.

[150] Y Explorations (XYI), San Jose, CA. *eXCite C to RTL Behavioral Synthesis 4.1(a)*, 2010.

[151] P. Yiannacouras, J. G. Steffan, and J. Rose. Exploration and customization of fpga-based soft processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):266–277, Feb 2007.

[152] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Vespa: Portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 61–70, 2008.

[153] T. Yuki, A. Morvan, and S. Derrien. Derivation of efficient fsm from loop nests. In *IEEE FPT*, pages 286–293, Kyoto, Japan, December 2013.

[154] W. Zhang, V. Betz, and J. Rose. Portable and scalable fpga-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(1):6–1–6–26, March 2012.