12-2019

Using Reduced Graphs for Efficient HLS Scheduling

Stephanie Soldavini
ss1120@rit.edu

Using Reduced Graphs for Efficient HLS Scheduling

STEPHANIE SOLDAVINI

Using Reduced Graphs for Efficient HLS Scheduling

Stephanie Soldavini

December 2019

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in Computer Engineering

R·I·T | Kate Gleason
College of ENGINEERING

Department of Computer Engineering

Using Reduced Graphs for Efficient HLS Scheduling

Stephanie Soldavini

Committee Approval:

Dr. Marcin Łukowiak *Advisor* Date
RIT, Department of Computer Engineering

Dr. Sonia López Alarcón Date
RIT, Department of Computer Engineering

Dr hab. inż. Paweł Śniatała Date
Poznan University of Technology, Department of Computer Science

In loving memory of Gary Soldavini

Acknowledgments

I would like to thank my family, in particular my parents, Amy and Mark, for supporting me in every way possible every day of my life.

I would like to thank my boyfriend of nearly eight years, Toby Lin, for being someone I can explain everything I learn to and for giving me support and words of encouragement, even from 350 miles away.

I would like to thank all of my friends here at RIT, Dakota Folger, Kevin Millar, Max Proskauer, Emily Reynolds, Michael Shullick, and Humza Syed for always being there through all of the shared ups and downs. In particular I would like to thank Andrew Ramsey, who has been my twin since the very beginning, and for keeping me in a constant struggle to keep up with his level of excellence.

I would like to thank my peers and friends in the High Performance Computing lab and in the Applied Cryptography and Information Security lab including Jason Blocklove, Thomas Cenova, Bradley Conn, Tom Guerin, Jerry Kotas, Stephen Lucas, Eri Montano, Braeden Morrison, Yash Nimkar, Prathibha Rama, Cody Tinker, and Eric Scheler. We're all in this together.

I would like to thank our fantastic lab manager, Rick Tolleson, for always having an open chair when I need something or just need a break.

I would like to thank all of my professors, particularly my committee for their advising and support. Thank you to Dr. Sonia López Alarcón for her input and support and for being an excellent woman mentor in a space where that is hard to find, Dr hab. inż. Paweł Śniatała for his incredible hospitality when I was otherwise alone in a completely new country, and Dr. Marcin Łukowiak for always knowing my worth and pushing me to realize my potential over and over again.

Finally, I would like to thank my cat, Crookshanks, for her companionship and love throughout these last few months and for making me smile and laugh when I need it the most.

Abstract

High-Level Synthesis (HLS) is the process of inferring a digital circuit from a high-level algorithmic description provided as a software implementation, usually in C/C++. HLS tools will parse the input code and then perform three main steps: allocation, scheduling, and binding. This results in a hardware architecture which can then be represented as a Register-Transfer Level (RTL) model using a Hardware Description Language (HDL), such as VHDL or Verilog. Allocation determines the amount of resources needed, scheduling finds the order in which operations should occur, and binding maps operations onto the allocated hardware resources. Two main challenges of scheduling are in its computational complexity and memory requirements. Finding an optimal schedule is an NP-hard problem, so many tools use elaborate heuristics to find a solution which satisfies prescribed implementation constraints. These heuristics require the Control/Data Flow Graph (CDFG), a representation of all operations and their dependencies, which must be stored in its entirety and therefore use large amounts of memory.

This thesis presents a new scheduling approach for use in the HLS tool chain. The new technique schedules operations using an algorithm which operates on a reduced representation of the graph, which does not need to retain individual dependency information in order to generate a schedule. By using the simplified graph, the complexity of scheduling is significantly reduced, resulting in improved memory usage and lower computational effort. This new scheduler is implemented and compared to the existing scheduler in the open source version of the LegUp HLS tool. The results demonstrate that an average of 16 times speedup on the time required to determine the schedule can be achieved, with just a fraction of the memory usage (1/5 on average). All of this is achieved with 0 to 6% of added cost on the final hardware execution time.

Contents

List of Figures

List of Tables

Acronyms

ALAP	As Late as Possible
ASAP	As Soon as Possible
AST	Abstract Syntax Tree
BB	Basic Block
BSV	Bluespec System Verilog
CDFG	Control/Data Flow Graph
CFG	Control Flow Graph
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
FDS	Force-Directed Scheduling
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GVN	Global Value Numbering
HDL	Hardware Description Language

HLL	High Level Language
HLS	High-Level Synthesis
ILP	Instruction Level Parallelism
IR	Intermediate Representation
MM	Matrix Multiplication
NTT	Number Theoretic Transform
RAW	Read After Write
RDFG	Reduced Data Flow Graph
RTL	Register-Transfer Level
SDC	System of Difference Constraints
SSA	Single Static Assignment
WAR	Write After Read
WAW	Write After Write

Chapter 1

Introduction

1.1 Motivation

High-Level Synthesis (HLS) is the method of creating a custom digital hardware design from High Level Language (HLL) software code. This is useful for many reasons. For instance, dedicated hardware designs are generally faster and more energy efficient than software implementations of the same functionality. However, hardware design requires a specific skill set with a fairly steep learning curve. HLS facilitates hardware design by making it similar to software development, allowing the user to operate at a higher level of abstraction than traditional hardware design. Another advantage is that in HLS tools, the same software code can easily be used to generate many hardware designs, such as one which is focused on speed or another which is focused on area or yet another which is focused on reduction of energy consumption. In an ideal scenario, circuits implemented through HLS achieve the same objectives (performance, area, energy efficiency) as handcrafted hardware [1], but with significantly less effort.

This design approach has been supported, for example, by Xilinx —one of the leading manufactures of Field Programmable Gate Array (FPGA) devices— in its Vivado HLS, SDAccel and SDSoC tool-sets, and in their recently introduced software-centric development platform, Vitis [2]. The main purpose of this new environment is to provide even greater support for the deployment of hardware accelerated systems at the edge or in the cloud, without the need for hardware design expertise. In addition, Intel, in its acquisition of Altera, has its Intel HLS Compiler. Intel also recently introduced One API [3], which is its unified cross-architecture programming model designed such that a single code base written in a single software language can utilize CPUs, GPUs, AI, and FPGAs.

1.2 High-Level Synthesis

The HLS process can be summarized in a series of steps:

- Parse software code into a format the tool can operate on
- **Allocation**: Identify the necessary components, connectivity, and control logic
- **Scheduling**: Determine the order in which the operations occur
- **Binding**: Map the scheduled operations onto the allocated components
- Generate an Hardware Description Language (HDL) model

The generation of the schedule is a key step in creating quality hardware during HLS process [4, 5, 6, 7]. Scheduling starts by extracting a Control/Data Flow Graph (CDFG) from the software code. This CDFG represents the control and dataflow dependencies between assembly-level software instructions. Traditional graph storage methods include data structures in the form of an adjacency list for each vertex or incidence and adjacency matrices, which for software CDFGs are sparse matrices. All of these require significant amounts of memory when working with nontrivial cases. The scheduling process then examines this graph to create a schedule which guarantees that dependencies are not violated. Finding an optimal schedule is an NP-hard problem [8], so conventional schedulers use heuristics to simplify the process.

The more elaborate schemes generate schedules which result in a shorter xecution time, but take more time to find them.

1.3 Objective

The objective of this research is to demonstrate that it is possible to efficiently and accurately schedule the operations extracted from the HLL description using a simplified graph representation as presented in [9]. The efficiency of our approach was examined using LegUp [10] —an existing open source HLS tool. LegUp's scheduler was used as a baseline for comparison and was replaced with the proposed reduced-graph based custom scheduler.

1.4 Approach

The proposed process for this project was to modify the scheduling portion of the LegUp 4.0 HLS tool such that it used the Reduced Data Flow Graph (RDFG) approach [9]. The original graph extraction was replaced with the extraction of the RDFG and the System of Difference Constraints (SDC) scheduler was replaced with the new scheduling algorithm. Then the original LegUp tool and the modified version were used to generate HDL for several C implementations of common algorithms. First, the generated HDL was simulated to ensure the new scheduler produced functionally-correct hardware. Then, both versions of the HDL were simulated over various input sizes to gather the execution times as the number of clock cycles each version took to complete. Finally, instrumentation was added to both tools to measure schedule generation time and memory usage.

Chapter 2

Background

2.1 High-Level Synthesis

HLS consists of two main steps: the front end and the back end. The front end of HLS turns HLL descriptions into a partially optimized Intermediate Representation (IR) in the form of a CDFG. The back end takes the IR and maps operations onto hardware components and generates HDL. The general flow is shown in Figure 2.1.

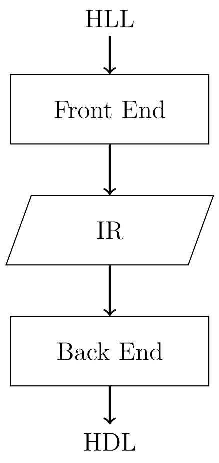


Figure 2.1: Overall flow of HLS

A software compiler is generally organized as a front end, an optimizer, and a back end. The front end is responsible for parsing the source and creating an Abstract Syntax Tree (AST) which is represented in the compiler's IR. The optimizer performs various optimizations on an IR [11]. These optimizations are either machine-independent or machine-dependent. Machine-independent optimizations simplify code no matter what the target architecture is, and are therefore useful in all cases. Machine-dependent optimizations depend on the target architecture, and therefore are only executed when they will be advantageous. After the IR is optimized, the back end generates machine code in the target architecture based on the optimized IR.

HLS is organized in a similar way. The front end is the same as the front end of a software compiler. Machine-independent optimizations are performed, since even though the code will not be compiled into machine-code, these optimizations still simplify the IR and often result in more efficient hardware.

Some machine-independent optimizations include dead-code elimination, strength reduction, and constant propagation and folding [12]. Dead-code elimination is the removal of any code that would never be executed. Strength reduction is the conversion of more expensive operators to less expensive equivalents, such as a multiplication or division by a power of two can easily become a single bit shift. Constant propagation and folding is the precomputation of any operations done on constants.

A particular example of a software compiler used as the front end in several open source HLS tools, is the LLVM Compiler Infrastructure [13]. In LLVM the front ends, optimizer, and back ends are decoupled and organized as a collection of libraries. Because of this decoupling, LLVM supports many front ends for various software languages and many back ends for various CPU architectures. Also, this makes LLVM easily extensible for more front or back ends. A flow diagram of LLVM is shown in Figure 2.2. The output of any LLVM based front end is the LLVM IR. This IR can completely represent input code and is the only interface between decoupled components of LLVM. The LLVM optimizer is a collection of optimization iterations, known as “passes”, each implemented as a C++ class which take in IR input and produce IR output which has been modified according to the optimization. This means that passes can be chosen to fit the application.

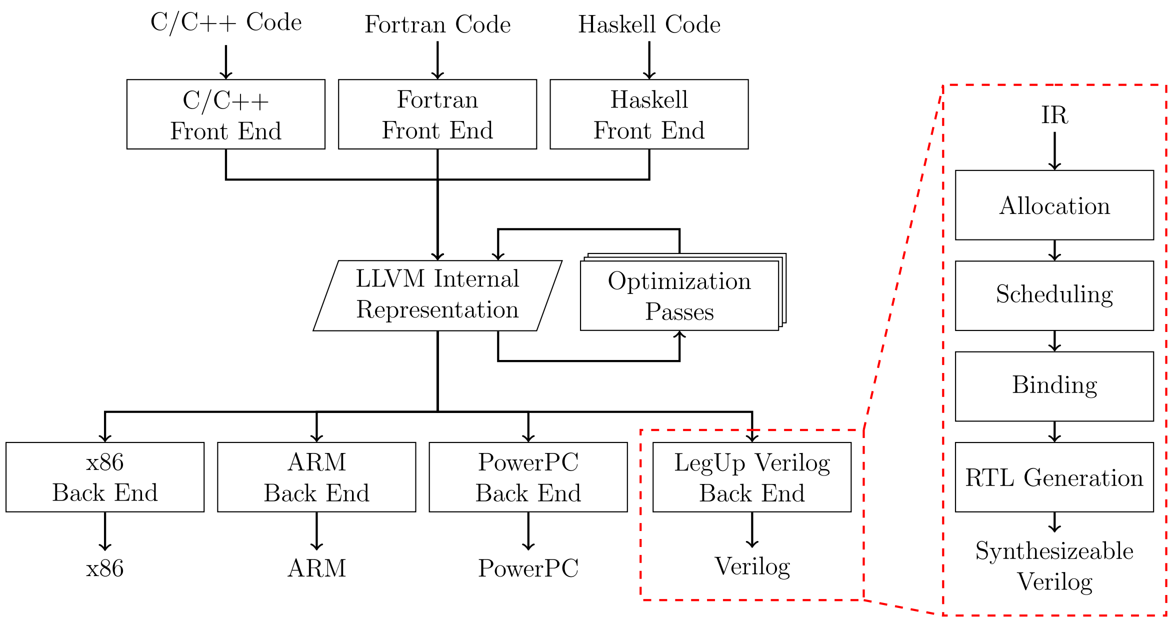


Figure 2.2: Flow chart of LLVM, with the LegUp Verilog back end added

The IR of a compiler can often be traversed as a CDFG, and this is true of the LLVM IR [12]. The control flow of a program can be presented in a Control Flow Graph (CFG) where the nodes are the Basic Blocks (BBs) of the program and the edges are the control dependencies between them. A BB is a section of code with one entry point and one exit point such that once the program enters the BB, all instructions in the block are executed in order until the end of the BB where the program conditionally branches to a different BB. Within a BB, the data flow between operations is represented by a Data Flow Graph (DFG) where the nodes are the operations and the edges are the data dependencies such that an edge is drawn from operation i to operation j if j requires the result of i. The CDFG is constructed by inserting the DFGs of each BB into the nodes of the CFG such that a graph of the control and data dependencies of the entire program is represented.

Given a quadratic equation, $ax^2 + bx + c = 0$ where $a \neq 0, b, c \in \mathbb{R}$, the solutions for x, known as the "roots", can be found using the quadratic formula, $x_{1,2} = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Listing 2.1 is an example function for solving the quadratic formula. For the sake of simplicity, if the roots fall in the complex domain, $x \in \mathbb{C}$, the function returns 1 to indicate a failure, otherwise the roots are stored in `r1` and `r2` and the return value is 0. The BBs for this function are annotated on the C code

```
int qsolve(double a, double b, double c, double* r1, double* r2) {
    int ret = 0;
    double disc = (b*b) - (4*a*c);      // discriminant          BB0
    if (disc > 0) {                     // unique real roots
        *r1 = (-b + sqrt(disc)) / (2*a);
                                                                 BB1
        *r2 = (-b - sqrt(disc)) / (2*a);
    } else if (disc == 0) {             // double real roots     BB2
        *r1 = *r2 = -b / (2*a);                                  BB3
    } else {                            // complex roots
                                                                 BB4
        ret = 1;
    }                                                            BB5
    return ret;                                                  BB6
}
```

Listing 2.1: C code for a quadratic formula solver with the BBs partitioned

as BB0 through BB6.

Figure 2.3 visualizes the CFG of this function where a diamond node indicates a BB with a conditional branch and the edges leaving that node are labeled with “T” or “F” to indicate which path is taken based on a true or false evaluation of the branch respectively. A rectangle node indicates a BB with an unconditional branch and the edge leaving the node goes to the branch target. The DFG for BB1 of the quadratic formula solver function, where roots are computed if there are two unique roots, is shown in Figure 2.4. The variables used in the BB which are live on entry are `a`, `b`, and `disc`, shown in rectangles at the top of the graph. Also needed for the calculations are the constant literals 0 and 2, which are also inherently “ready” at the beginning of the BB. The circle nodes represent individual operations needed for the calculations. Shown in rectangles at the bottom of the graph are the variables `r1` and `r2`, which are produced in the BB and are needed in later BBs, in this case outside of the function. The CDFG of this function could be represented by inserting all of

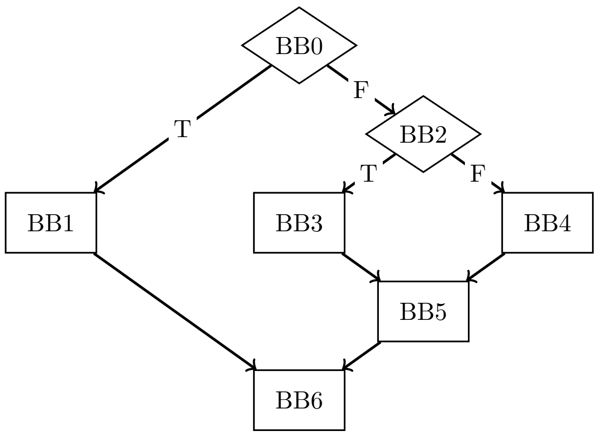


Figure 2.3: CFG of quadratic formula solver function. Diamond nodes are BBs with conditional branched and rectangle nodes are BBs with unconditional branches.

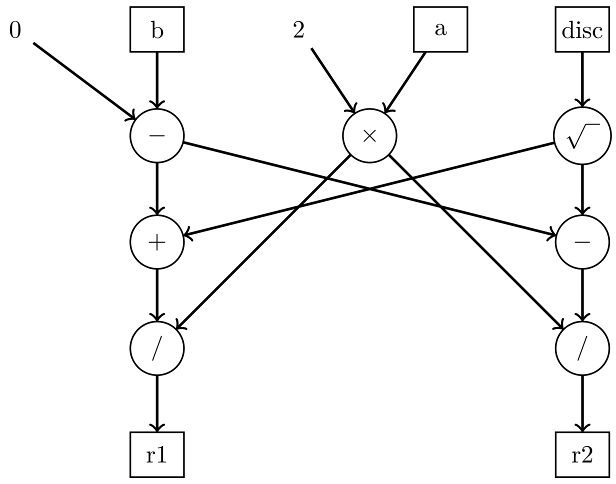


Figure 2.4: DFG of BB1 of the quadratic formula solver function where two unique roots are computed

the individual DFGs of the BBs into the nodes of the matching BB in the CFG.

The HLS back end uses the CDFG to map operations into a hardware design [12]. This is done in four main steps: allocation, scheduling, binding, and HDL generation. The first three of these steps are all interdependent and can be solved in any order and any HLS implementation chooses the order to solve these steps which best suits their goals. Allocation determines the amount of resources in three groups: Functional Units (FUs) (i.e. adders, multipliers), routing resources (i.e multiplexers, buses), and storage resources (i.e. registers, memory) [12]. Allocation is generally done in one of two ways, by constraining resource usage to the target FPGA or by choosing a minimally necessary set of FUs.

Scheduling can either be done for the DFG of each BB or over the program's entire CDFG [12]. For a nontrivial program, scheduling the entire CDFG can be impractical due to memory or time constraints. Examples of typical scheduling algorithms employed in HLS are as follows. The As Soon as Possible (ASAP) algorithm schedules operations in the first cycle they are ready to execute. The As Late as Possible (ALAP) algorithm schedules operations in the latest cycle they can execute such that their results are available in time for the operations that depend on them and such that the overall schedule is the same length as the ASAP schedule. The Force-Directed Scheduling (FDS) algorithm uses a heuristic to optimize resource usage while still maintaining the length of the ASAP and ALAP schedules. The System of Difference Constraints (SDC) algorithm schedules by representing constraints in a system of equations to be solved. This algorithm is flexible in types of scheduling and types of constraints, but it cannot schedule across BB boundaries. The List Scheduling (LS) algorithm is a resource-constrained scheduling algorithm which, given a set of FUs, schedules all operations which are ready to execute onto the available FUs. Once there are no ready operators which can be scheduled onto the available hardware, the algorithm moves on to the next control step.

Binding depends on the allocation method [12]. If allocation was simply constraining to the resource availability of the target FPGA, binding must only ensure these constraints are met. Depending on the goal of the HLS tool, binding may optionally attempt to minimize area. If a minimal set of FUs were allocated, binding is responsible for assigning operations in each control step to the FUs and for adding the necessary routing and control logic.

The HDL generation process takes the results of allocation, scheduling, and binding, and generates HDL code, typically VHDL or Verilog, which can then be synthesized and implemented by hardware design tools such as Xilinx Vivado or Intel Quartus Prime [12].

2.2 Popular HLS Tools

The existing HLS tools can be divided into two categories: commercial and open source [6]. The most prevalent commercial HLS tools are Xilinx Vivado HLS [14] and Intel HLS Compiler [15]. Other notable commercial HLS tools are Handel-C [16] and the Bluespec BSV Compiler [17]. Bluespec uses the Bluespec System Verilog (BSV) language as an input rather than an existing software programming language. Open source HLS tools include Bambu [18], Gaut [19], LegUp [10], MyHDL [20], ROCCC [21], and Trident [22]. All of them, with the exception of MyHDL, use C as an input language and are built using various compiler frameworks (GCC, LLVM, Eclipse) as their front ends. A summary of these tools is shown in Table 2.1.

Table 2.1: Summary of current popular HLS tools

License	Tool Name	Input	Output	Front End	FPGAs
Commercial	Vivado HLS [14]	C/C++/ System C	VHDL/Verilog	Eclipse	Xilinx
	Intel HLS [15]	C/C++	Verilog	-	Intel
	Handel-C [16]	C	Verilog/VHDL	-	Xilinx/Altera
	Bluespec [17]	BSV	Verilog/ SystemC	-	Device agnostic
Open source	Bambu [18]	ANSI C	Verilog/VHDL	GCC	Xilinx
	Gaut [19]	C/C++	VHDL	LLVM	Xilinx/Altera
	LegUp [10]	ANSI C	Verilog	LLVM	Altera
	MyHDL [20]	Python	Verilog/VHDL	Python	Device agnostic
	ROCCC [21]	C	VHDL	Eclipse	Device agnostic
	Trident [22]	C	VHDL	LLVM	Device agnostic

2.2.1 Xilinx HLS Tools

Xilinx Vivado HLS [14] is a commercial HLS tool owned by Xilinx Inc, originally acquired from AutoESL where it was known as AutoPilot. Vivado HLS is tightly integrated with the rest of the Xilinx suite of tools. The use cases for the HLS tool are to take a C program and generate a fully hardware design, or to take a C function within a larger program and turn it into a hardware block to be integrated into a software and hardware co-design solution. Along with the C code, other inputs to the tool include constraints, such as clock period and FPGA target, and directives, extra commands to guide the tool towards a particular behavior or optimization.

Xilinx SDSoC is an HLS tool specifically for Xilinx's Zynq SoCs and Zynq Ultrascale+ MPSoCs. It abstracts some of the more manual and complicated portions of using Vivado HLS away, such as the data-mover configuration. Xilinx SDAccel makes the same abstractions, but instead of targeting the Zynq architecture, it is for PC systems with an FPGA connected via PCIe.

2.2.2 LegUp

LegUp is an open source HLS tool developed at the University of Toronto [10]. It uses the existing LLVM front-end for C code and implements a Verilog back end to perform the allocation, scheduling, binding, and HDL generation. For allocation, LegUp reads a configuration Tcl file specifying the target FPGA and the resource constraints of the device. The default synthesized architecture will try to exploit hardware parallelism with the goal of achieving desired performance. For scheduling, the earlier version of LegUp used ASAP scheduling and the most recent open source version 4.0 uses SDC [23]. The SDC algorithm schedules by representing constraints in a system of equations to be solved. For binding, LegUp uses a weighted bipartite matching heuristic with the goal of minimizing shared FUs.

In LegUp, each step in the HLS process is coded in a modular fashion such that replacing the scheduler with a new one does not require full understanding of the entire code base. For this reason, the LegUp 4.0 infrastructure was chosen to implement the reduced graph scheduling method described in [9] in an HLS context. The time and memory performance of this scheduler was tested against an unchanged version of the LegUp 4.0 scheduler.

LegUp can be configured for several different flows, including software-only, hardware-only, and hybrid. The software-only flow compiles the input application as software which can simulate as running on a processor. The hardware-only flow generates hardware from the entire input application. The hybrid flow generates a hardware coprocessor for a single function, which is chosen using a configuration Tcl file, and the remaining input code is compiled as software. This system is then simulated as a soft core processor running the code which sends data to the coprocessor of the accelerated function.

2.3 The LLVM Compiler Infrastructure

2.3.1 LLVM IR

The LLVM IR is designed such that the internal bitcode is one-to-one to a human readable assembly-like representation, similar to how machine code maps to assembly language. The IR is an Single Static Assignment (SSA) based representation of the input code which has the important properties that every variable is assigned exactly one time and every variable is defined before it is used. Oftentimes, in software code, a variable is assigned to several times. In the IR, to maintain SSA form, this variable would be split into versions where each version is given a different name. However, if a variable is assigned different values depending on the control flow of the program, such as in Figure 2.5a, a PHI node is necessary to resolve the value. The PHI node assigns a variable a value based on the preceding BB the control flow passed through. An SSA form version of this example is shown in Figure 2.5b where the variable `b2` is assigned `b0` if the `if` portion was executed, and `b1` if the `else` portion was executed.

The LLVM IR is designed to be at a low-level where high-level code can easily be mapped to it, analyses and transformations can be performed, and then IR can easily be mapped to machine code of the target processor. LLVM IR instructions are sorted into a few categories: binary instructions, terminator instructions, memory

```
if (a == 0) {
    b = c;
} else {
    b = a;
}
```

(a) Original if-tree

```
if (a == 0) {
    b0 = c;
} else {
    b1 = a;
}
b2 = phi [b0, if], [b1, else];
```

(b) SSA if-tree with PHI node

Figure 2.5: PHI node example

instructions, and other instructions. The most common type of LLVM IR instruction is a binary instruction, or an instruction with two input operands. An example of a binary instruction is shown in Figure 2.6.

$$\underbrace{\%\text{mul1}}_{\text{identifier}} = \underbrace{\text{fmul}}_{\text{operation}} \underbrace{\text{double}}_{\text{type}} \underbrace{4.000000\text{e}{+}00}_{\text{operand 1}}, \underbrace{\%\text{a}}_{\text{operand 2}}$$

Figure 2.6: A typical LLVM IR instruction for a binary operation

The structure of most LLVM IR instructions is similar to this example. If the instruction produces a result, it starts with an identifier for that result and an equality operator. Next is the type of operation, in this case "fmul" or a floating point multiply. After that is the type of the result and operands, which here is "double." Other common types are "float", "i" followed by a number —which can be arbitrarily large— (such as "i1," "i32," or "i65536") indicating an integer of that many bits, arrays ([4 x i32] indicates an array of four 32-bit integers), or pointers to any other type indicated by a type followed by an asterisk (*). After the type, the operands or inputs to the operation are specified in a comma separated list, in this case a literal double-precision floating-point '4' and an identifier for the variable 'a'. Other typical binary instructions are for addition or subtraction, division, or bitwise operations such as bitwise-and or bitwise-or.

Identifiers, the names given to variables or expressions, either begin with '@' if they are global or with '%' if they are local and they can either be strings derived from the code or simply numerical values. In Figure 2.6, the identifier “%mul1” is local and the name was derived from the “mul” variable in the C code. “%a” is also an identifier for the input variable “a”.

Terminator instructions are the last instruction in any basic block and indicate the control flow of the program. These instructions are usually branches (“br”), which indicate a jump to another basic block within the function or returns (“ret”), which indicate control flow returns to the caller.

The most common memory instructions are shown in Figure 2.7.

$$\underbrace{\text{\%a.addr}}_{\text{address}} = \text{alloca } \underbrace{\text{double}}_{\text{type}}\ \underbrace{\text{align 8}}_{\text{alignment}}$$

$$\text{store } \underbrace{\text{double}}_{\text{type}}\ \underbrace{\text{\%a}}_{\text{id}},\ \underbrace{\text{double*}}_{\text{addr type}}\ \underbrace{\text{\%a.addr}}_{\text{address}},\ \underbrace{\text{align 8}}_{\text{alignment}}$$

$$\underbrace{\text{\%8}}_{\text{id}} = \text{load } \underbrace{\text{double}}_{\text{type}},\ \underbrace{\text{double*}}_{\text{addr type}}\ \underbrace{\text{\%a.addr}}_{\text{address}},\ \underbrace{\text{align 8}}_{\text{alignment}}$$

Figure 2.7: Allocate, store, and load instructions

The “alloca” instruction allocates memory based on the type and alignment and assigns the address of that memory to the identifier on the left hand side. The “store” instruction writes the value represented by the first identifier to the address represented by the second identifier. The “load” instruction reads the value at the address parameter and assigns the value to the left hand side identifier.

The “phi” instruction is unusual in that it does not have a counterpart in actual assembly language. This instruction is used to represent the PHI node in the SSA graph and if present must always be the first instruction in a BB. An example is shown in Figure 2.8.

This instruction assigns a value to the left hand identifier based on the preceding BB which terminated in a branch into the current BB. In this example, `%ret` is assigned 0 if the control flow came from the `%if.then` BB and 1 if the control flow

$$\underbrace{\%\text{ret}}_{\text{id}} = \text{phi}\ \underbrace{\text{i32}}_{\text{type}}\ [\underbrace{\%\text{a}}_{\text{value 1}}, \underbrace{\%\text{if.then}}_{\text{label 1}}], [\underbrace{\%\text{b}}_{\text{value 2}}, \underbrace{\%\text{if.else}}_{\text{label 2}}]$$

Figure 2.8: A LLVM phi instruction

came from the `%if.else` BB. This is useful in compiler IR to maintain the property of SSA where each identifier represents one value. In actual machine code, the same register would be used for the value from each predecessor BB (here to hold either 0 or 1) as the identifier (here `%ret`) such that the correct value simply propagates through. In hardware, however, this is simply represented as a multiplexer with inputs from the predecessor BBs. The instruction shown in Figure 2.8 is shown as a multiplexer in Figure 2.9.

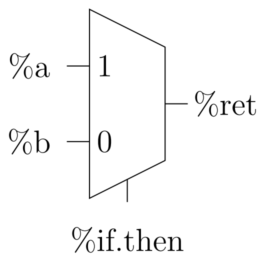


Figure 2.9: Hardware representation of a PHI node as a multiplexer

2.3.2 LLVM Infrastructure Classes

The LLVM represents the code to be compiled by several related classes. The most relevant ones and their relationships are described here.

The `Module` class represents the overall structure of the input program. It contains a list of Functions, GlobalVariables, and a SymbolTable.

The `Value` class represents any sort of typed value, which can be `Constants`, `Arguments`, `Instructions`, and `Functions`. This class keeps a list of all of the `Users` of the `Value`, which are any other nodes which consume the `Value`. An SSA variable and the operation which produced it are represented as one in the same `Value`, which means the `User`s of a `Instruction` are the other `Instruction`s which use the result as an operand.

The `User` class is the base class for any LLVM node that may 'use' other `Value`s. It holds a list of `Value`s which are its operands. This class is a subclass of the `Value` class.

The `Instruction` class is the base class for all types of instructions. It holds the opcode and the `BasicBlock` it is in. Also, since it is a subclass of the `User` class, its operands can easily be accessed. There are subclasses for all kinds of instructions, such as `BinaryOperator`, `PHINode`, and `ReturnInst`, and enums defined to easily identify the exact operation of the `Instruction`.

The `Function` class represents functions in the `Module`. It keeps track of a list of its `BasicBlock`s, a list of its `Argument`s, and a `SymbolTable`.

The `BasicBlock` class represents BBs in the code. It has a list of the `Instruction`s which make up the BB, and the last `Instruction` is always a terminator instruction. The class also keeps track of its parent `Function`. This class is a subclass of the `Value` because they can be used by branches as a destination.

2.4 Related Work

There have been many attempts to optimize portions of the HLS process. One example was an autotuner for the input parameters to LegUp HLS which targets the weighted normalized sum of a variety of metrics presented in [24]. These input parameters included the operation latencies, resource constraints, and resource sharing patterns. The weighted normalized sum is defined as a sum of each metric multiplied with a weight for how much it affects the desired optimization scenario. For instance, when trying to optimize for area, the number of LUTs, Registers, BRAMs, and DSPs affect the area the most and are weighted by a multiplication of 8. Clock cycles and maximum frequency have a smaller effect and are weighted by a multiplication of 2.

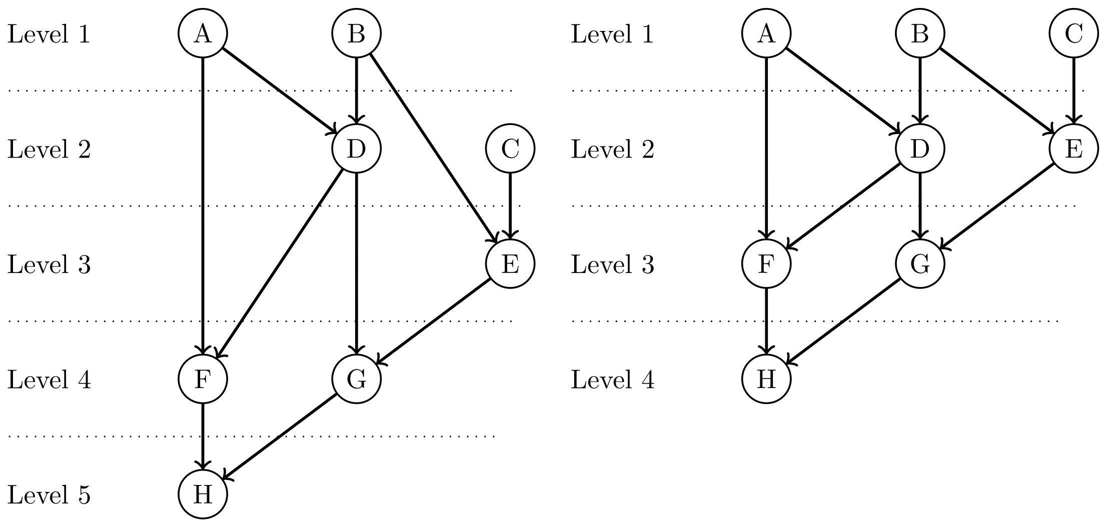
When attempting to optimize for latency, the number of registers, clock cycles, and maximum frequency have a large impact and are weighted by 8. The number of other resources has a lower impact and is weighted by 2. By minimizing this function the autotuner successfully reduces the weighted normalized sum in scenarios attempting to optimize for area, performance, latency, and a balance.

Optimizations to the scheduling portion itself include using an adaptive genetic algorithm to perform a design space exploration on the optimal scheduling of a CDFG presented in [25]. In this paper, a novel encoding for the chromosomes used in the genetic algorithm was proposed which consisted of a ‘datapath string’ and ‘auxiliary string’. The encoding scheme enables effective design space exploration. As the process continues, the ‘datapath string’ evolves to eventually yield a satisfactory configuration for the resource array and unrolling factor. The proposed approach saw improvement in cost and runtime as compared to previous approaches.

Another scheduling optimization is the FALLS lookahead algorithm presented in [26] which attempts to reserve FUs for operations in the critical path rather than greedily scheduling. The List Scheduling (LS) algorithm initially pre-allocates insufficient FUs and must post-allocate greedily as it continues to schedule operations. The proposed FALLS algorithm uses a lookahead technique such that there is as much resource sharing as possible, and then uses a binary search to estimate the actual required FUs such that they can be accurately pre-allocated for maximum resource sharing. All of this maintains similar complexity to the LS algorithm, so the FALLS algorithm remains just as scalable. This approach was successful in reducing the number of FUs and had a shorter run time than other state-of-the-art algorithms.

2.5 Graph Based Optimization of Pipelined Architectures

A method for analyzing pipelined architectures and optimizing the schedule and control logic of operations is presented in [9]. Given a computation with a pipelined architecture, the first step is to create a DFG. Because the entire DFG of an application requires more memory than is practical to store all vertices and edges, only a reduced representation of the graph is stored. This reduced representation only holds the number of operations executed at each level of the graph in the *minimized configuration.* The *minimized configuration* of a graph is defined to be where all edges from operations in level L_i go to operations in a later level L_{i+j}, for $j > 0$, and that an operation in level L_i cannot be in level L_{i-1} due to precedence constraints. An example of this is shown in Figure 2.10. In this example, it can be seen that in Fig-



(a) Not minimized configuration **(b)** Minimized configuration

Figure 2.10: In minimized configuration, nodes are placed in the earliest level after all of their inputs

ure 2.10a several nodes are violating minimized configuration. For instance, node C has no inputs and therefore can be in level 1. Also, node F has inputs from nodes A and D, so it can be in the first level after both of them, level 3. In Figure 2.10b the graph is in minimized configuration because all nodes are placed as early as possible. The nodes without inputs— nodes A, B, and C —are in level 1. Nodes D and E only have A, B, or C as input and therefore are in level 2. The latest inputs of nodes F and G are in level 2, and therefore they are in level 3. Finally, the inputs to node H are in level 3 and therefore node H is in level 4.

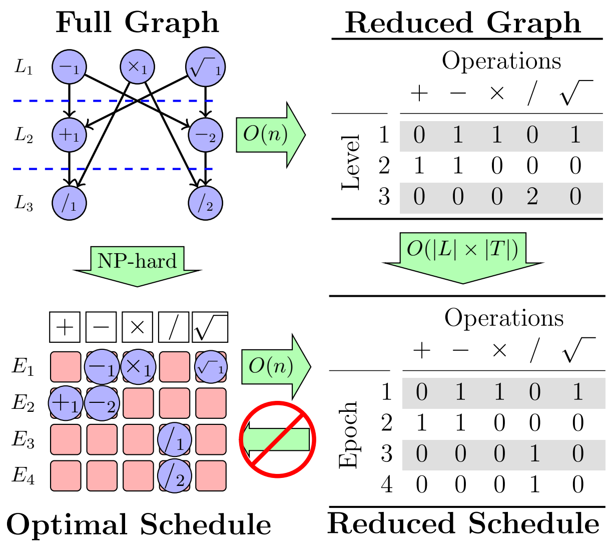
This reduced graph can be constructed without storing all nodes at once by only accessing nodes in the current level and each node can be freed when all of its successors are found.

Because of the constraints placed on the *minimized configuration*, even though precedence information is lost in the reduced representation, it can still be inferred due to placing operations in levels. This is enough to be able to create the *reduced schedule*. An iterative algorithm, shown in Algorithm 1, is used to schedule the operations and calculate the *span*, or number of *epochs* needed to execute the schedule. An *epoch* is the set of operations that are executed concurrently in the same amount of time (i.e. the same clock cycle). For each level of the reduced graph, the operations are *matched* to pipelines and each *use* of the pipeline is scheduled. A *matching* is a grouping of operations into *uses*. A *use* of a pipeline is the set of operations that correspond to the FUs in the pipeline. The number of epochs for the level is the maximum of all the uses. The number of epochs in the level is added to the span, except when the number of epochs in a level is zero, then 1 is added to account for pipeline latency.

Algorithm 1 $EstimateSchedule(L, T, Ops, S, p)$

```
1: span := 0
2: sch[][] := {{}}
3: for all i := 1 to $|L|$ do
4:     uses[] := $matching(T, Ops, p, i)$
5:     epochs := max{uses[]}
6:     sch[][] := $addToSchedule(Sch, T, Ops, S, p,$ epochs$, i)$
7:     $Ops := removeNodes(T, Ops, S, p,$ epochs$, i)$
8:     span := span + max{1, epochs}
9: end for
10: return sch, span
```

Finding an optimal schedule from the CDFG is NP-hard. The reduced graph method, however, scales with the number of levels in the graph and the number of types of operations. These relative complexities are shown in Figure 2.11.



Level	+	−	×	/	√
1	0	1	1	0	1
2	1	1	0	0	0
3	0	0	0	2	0

Epoch	+	−	×	/	√
1	0	1	1	0	1
2	1	1	0	0	0
3	0	0	0	1	0
4	0	0	0	1	0

Figure 2.11: Complexity of standard scheduling compared to reduced-graph scheduling.

By creating this reduced schedule, an *achievable performance goal* is acquired in the form of the span and an *execution plan* is acquired in the form of the reduced schedule itself. The new architecture can be compared to the original architecture to determine if speedup was obtained. The results from [9] showed speedup of up to 10.7 as compared to the original architectures.

2.6 Contribution

Traditional HLS schedulers use advanced heuristics to generate schedules, due to the fact that finding an optimal schedule is an NP-hard problem. These heuristics are computationally intensive and therefore take significantly longer to find a schedule on larger problems. Additionally, these schedulers need access to individual dependencies in the CDFG which means the scheduler must store all of this information. With traditional graph storage methods, storing the entire graph takes a exponential amount of memory based on the number of nodes.

The goal of this research was to implement and evaluate a scheduler based on the reduced graph approach and determine if quality schedules can be produced with significant time and memory savings.

Chapter 3

LLVM Front End

LegUp is built as an extension to LLVM. It uses the existing LLVM C front end to parse C input code into IR and implements a Verilog back end to perform allocation, scheduling, binding, and HDL generation. In order to understand the IR that LegUp works with, the LLVM optimizer was studied.

3.1 LLVM Optimization Passes

The LLVM optimization stage works by running "passes" over the IR to analyze and transform the code. These passes analyze the IR or perform transformations based on the analysis. The goal of this process is to generate the most efficient machine code as possible. Different sets of passes can be chosen for different use cases. For instance, some passes are useful in reducing the code size but others may duplicate code to help optimize execution time of the final program. While these passes were designed for software optimization, many of them are useful for hardware optimization as well. An example flow of passes that could be used to optimize input C code with the goal of becoming efficient hardware is presented in Figure 3.1.

All passes are subclasses of the `Pass` class, and there are several classes to inherit from, depending on what the pass needs to do, such as the `FunctionPass`, `LoopPass`, or `BasicBlockPass` classes. A function pass will run a transformation or analysis on each function, likewise a loop or basic block pass will run a transformation or analysis

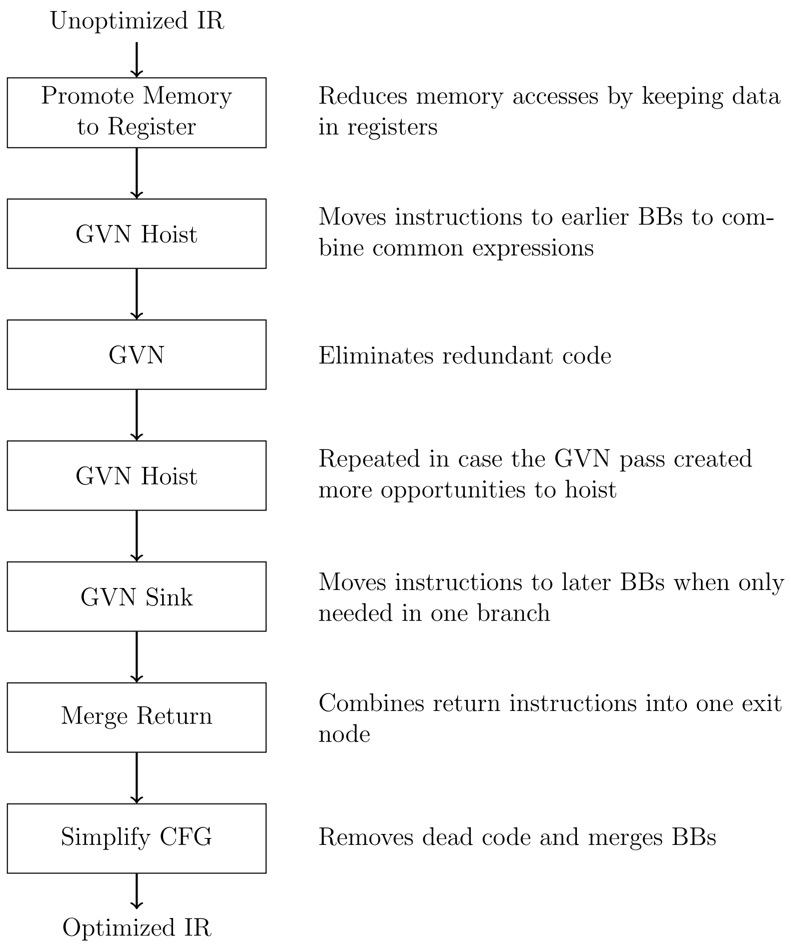

Unoptimized IR
Promote Memory to Register
Reduces memory accesses by keeping data in registers
GVN Hoist
Moves instructions to earlier BBs to combine common expressions
GVN
Eliminates redundant code
GVN Hoist
Repeated in case the GVN pass created more opportunities to hoist
GVN Sink
Moves instructions to later BBs when only needed in one branch
Merge Return
Combines return instructions into one exit node
Simplify CFG
Removes dead code and merges BBs
Optimized IR

Figure 3.1: Example flow of LLVM optimization passes

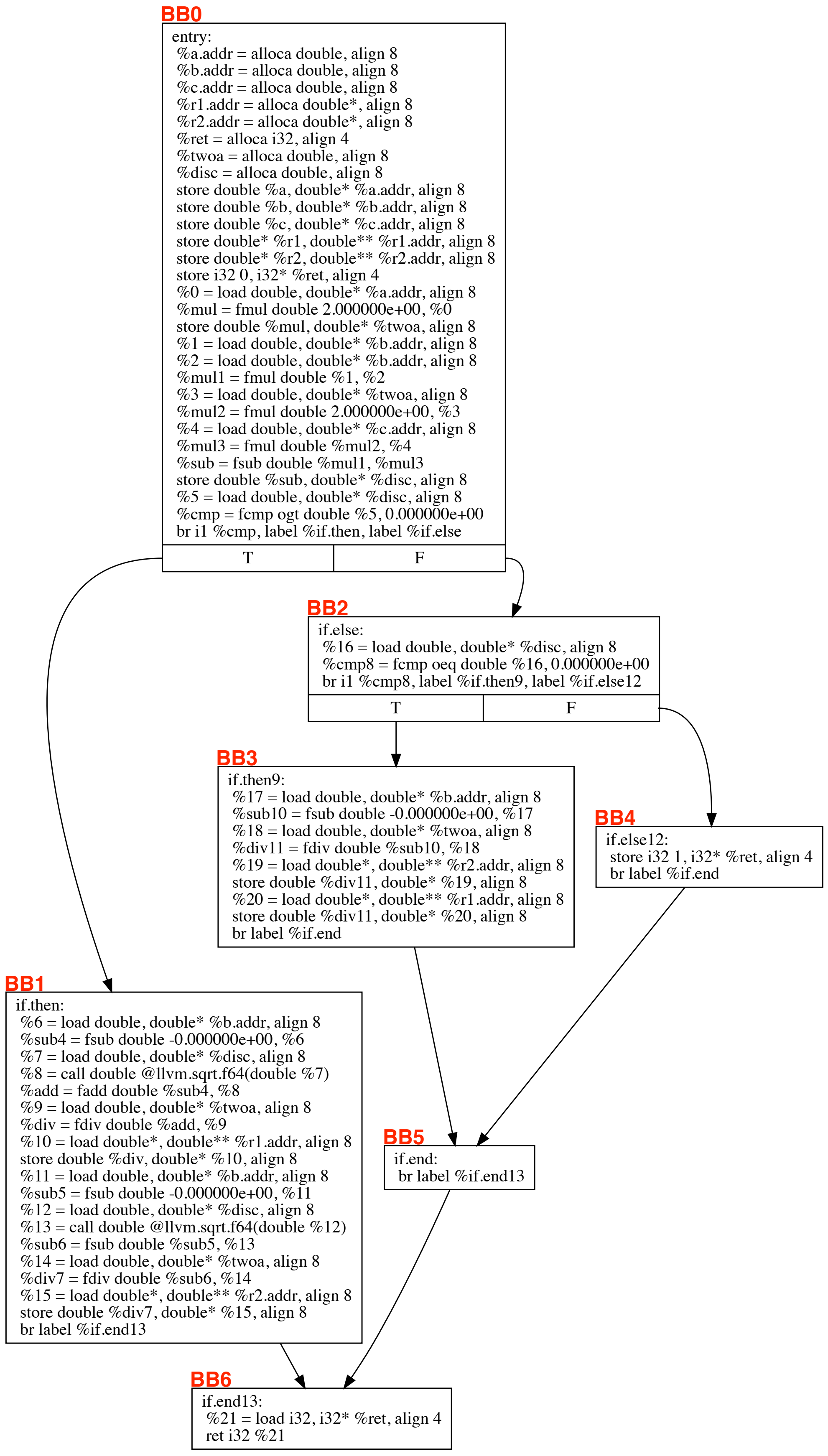
on each loop or basic block.

The quadratic solver function shown in Listing 2.1 is used to illustrate the functionality of these passes in detail. Figure 3.2 shows the CFG generated by LLVM with no optimization passes run with the human readable LLVM instructions inside each BB. The BBs are labelled the same as in Listing 2.1 and Figure 2.3, but in other figures the BBs will be referred to by their label in the code, at the top of the BB.

The *Promote Memory to Register* (`-mem2reg`) pass removes unnecessary loads and stores to memory when the same access can be placed into a register. By default, LLVM will allocate memory for every variable, store the initialization value into that allocated location, then load it into a register when the value is needed. Running this pass to remove these unnecessary memory accesses is useful to software because register accesses are much faster than memory accesses, and in many instruction sets data must be loaded into a register before it can be operated on anyway. When using the software model for HLS, eliminating unnecessary memory accesses simplifies the process of finding data dependencies, and the act of deciding if data belongs in a "register" (flip-flops) or "memory" (block RAM or external memory) occurs later, during the allocation step. Figure 3.3 shows the quadratic solver function code after this pass. All of the allocations, stores, and loads which could be simplified to registers were removed.

The *Global Value Numbering (GVN) Hoist* (`-gvn-hoist`) pass moves code to earlier BBs where possible to combine common expressions. This is beneficial both to reduce code size and expose more Instruction Level Parallelism (ILP). When translated to hardware, eliminating common expressions reduces the area of the circuit.

The *GVN* (`-gvn`) pass performs global value numbering which is the process of assigning numbers to expressions so that equivalent expressions have the same number. This is used to find and eliminate redundant code. This pass may create more opportunities to hoist code so the GVN Hoist pass is run again after GVN. Figure 3.4



```
entry:
 %a.addr = alloca double, align 8
 %b.addr = alloca double, align 8
 %c.addr = alloca double, align 8
 %r1.addr = alloca double*, align 8
 %r2.addr = alloca double*, align 8
 %ret = alloca i32, align 4
 %twoa = alloca double, align 8
 %disc = alloca double, align 8
 store double %a, double* %a.addr, align 8
 store double %b, double* %b.addr, align 8
 store double %c, double* %c.addr, align 8
 store double* %r1, double** %r1.addr, align 8
 store double* %r2, double** %r2.addr, align 8
 store i32 0, i32* %ret, align 4
 %0 = load double, double* %a.addr, align 8
 %mul = fmul double 2.000000e+00, %0
 store double %mul, double* %twoa, align 8
 %1 = load double, double* %b.addr, align 8
 %2 = load double, double* %b.addr, align 8
 %mul1 = fmul double %1, %2
 %3 = load double, double* %twoa, align 8
 %mul2 = fmul double 2.000000e+00, %3
 %4 = load double, double* %c.addr, align 8
 %mul3 = fmul double %mul2, %4
 %sub = fsub double %mul1, %mul3
 store double %sub, double* %disc, align 8
 %5 = load double, double* %disc, align 8
 %cmp = fcmp ogt double %5, 0.000000e+00
 br i1 %cmp, label %if.then, label %if.else
```

```
if.else:
 %16 = load double, double* %disc, align 8
 %cmp8 = fcmp oeq double %16, 0.000000e+00
 br i1 %cmp8, label %if.then9, label %if.else12
```

```
if.then9:
 %17 = load double, double* %b.addr, align 8
 %sub10 = fsub double -0.000000e+00, %17
 %18 = load double, double* %twoa, align 8
 %div11 = fdiv double %sub10, %18
 %19 = load double*, double** %r2.addr, align 8
 store double %div11, double* %19, align 8
 %20 = load double*, double** %r1.addr, align 8
 store double %div11, double* %20, align 8
 br label %if.end
```

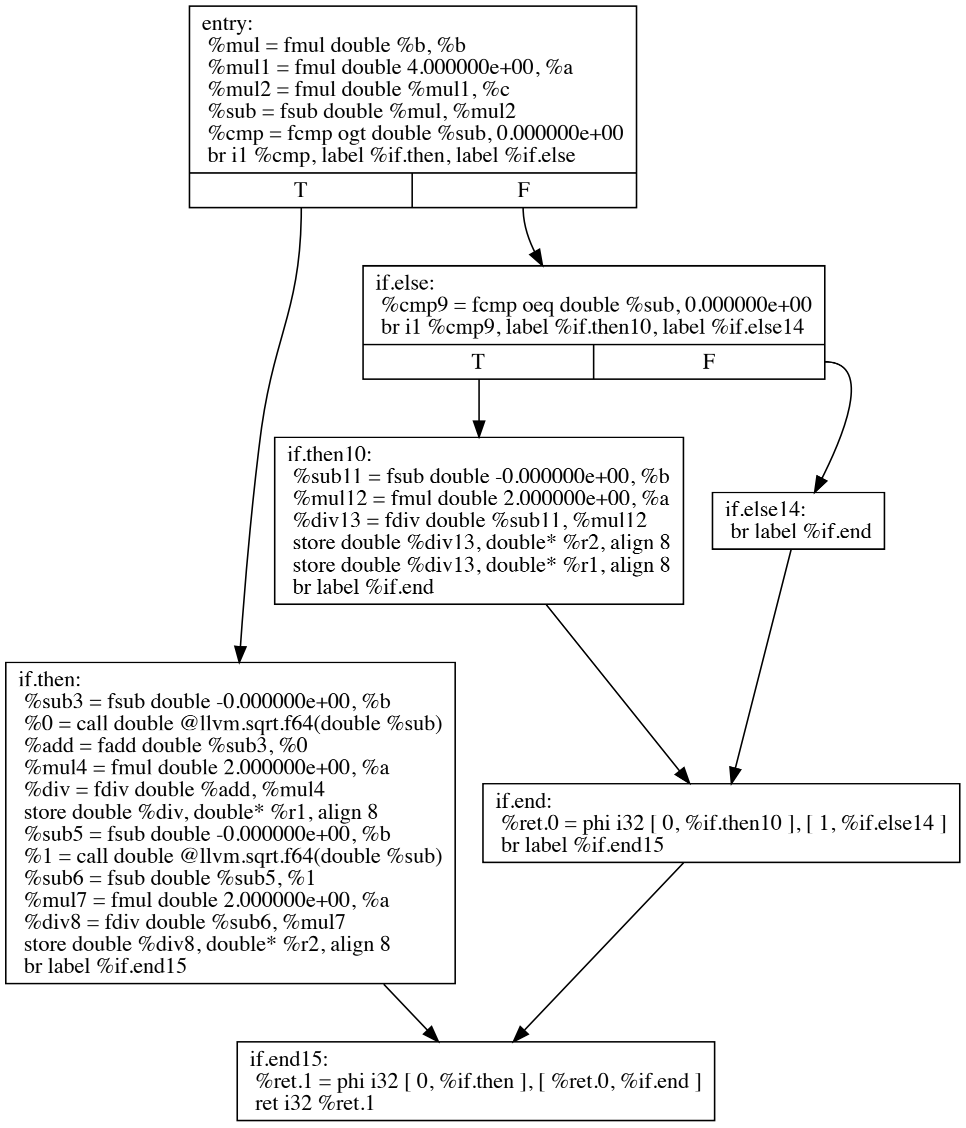
```
if.else12:
 store i32 1, i32* %ret, align 4
 br label %if.end
```

```
if.then:
 %6 = load double, double* %b.addr, align 8
 %sub4 = fsub double -0.000000e+00, %6
 %7 = load double, double* %disc, align 8
 %8 = call double @llvm.sqrt.f64(double %7)
 %add = fadd double %sub4, %8
 %9 = load double, double* %twoa, align 8
 %div = fdiv double %add, %9
 %10 = load double*, double** %r1.addr, align 8
 store double %div, double* %10, align 8
 %11 = load double, double* %b.addr, align 8
 %sub5 = fsub double -0.000000e+00, %11
 %12 = load double, double* %disc, align 8
 %13 = call double @llvm.sqrt.f64(double %12)
 %sub6 = fsub double %sub5, %13
 %14 = load double, double* %twoa, align 8
 %div7 = fdiv double %sub6, %14
 %15 = load double*, double** %r2.addr, align 8
 store double %div7, double* %15, align 8
 br label %if.end13
```

```
if.end:
 br label %if.end13
```

```
if.end13:
 %21 = load i32, i32* %ret, align 4
 ret i32 %21
```

Figure 3.2: CFG for quadratic solver function with no optimization passes run



```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 br i1 %cmp, label %if.then, label %if.else
```

```
if.else:
 %cmp9 = fcmp oeq double %sub, 0.000000e+00
 br i1 %cmp9, label %if.then10, label %if.else14
```

```
if.then10:
 %sub11 = fsub double -0.000000e+00, %b
 %mul12 = fmul double 2.000000e+00, %a
 %div13 = fdiv double %sub11, %mul12
 store double %div13, double* %r2, align 8
 store double %div13, double* %r1, align 8
 br label %if.end
```

```
if.else14:
 br label %if.end
```

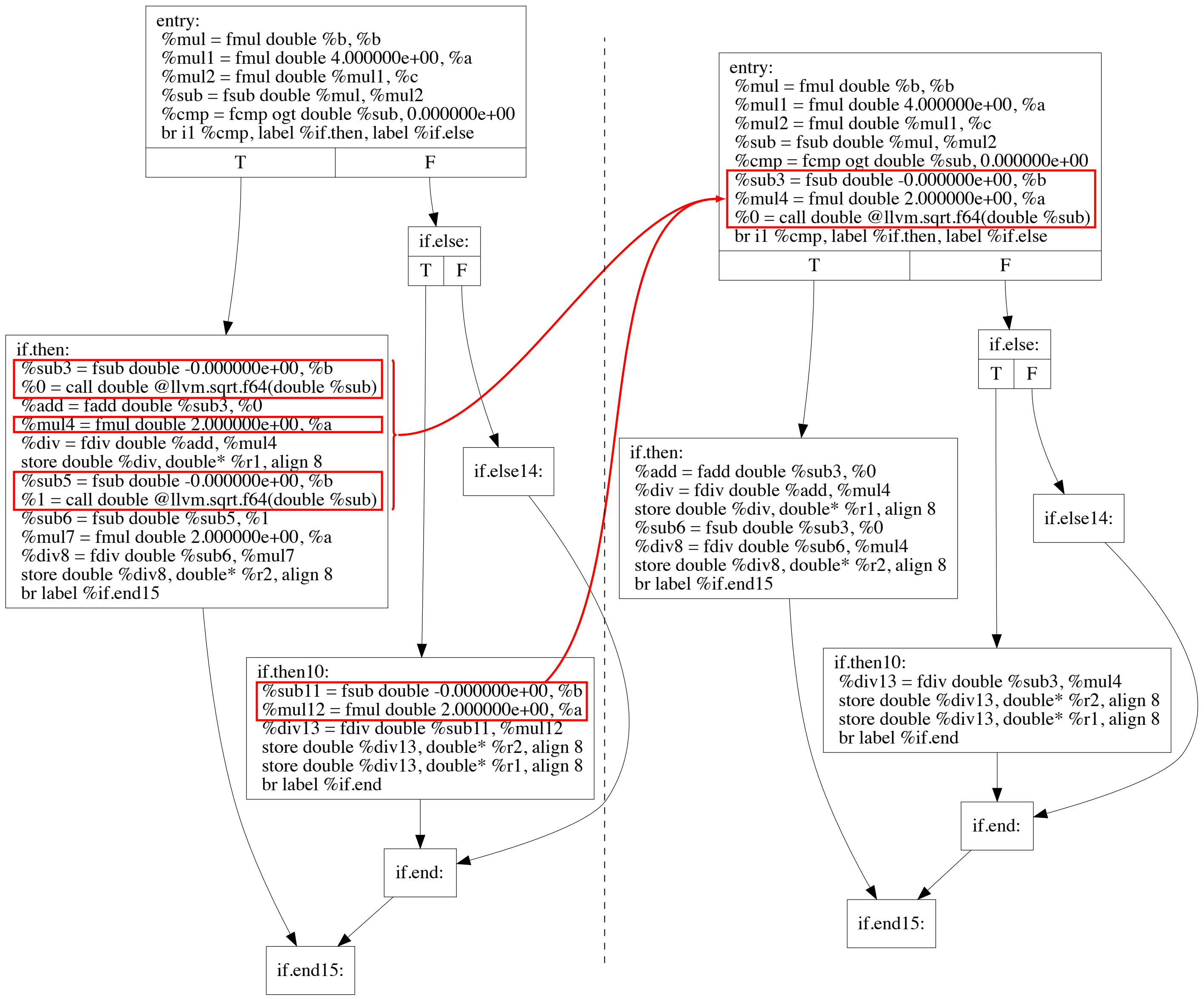
```
if.then:
 %sub3 = fsub double -0.000000e+00, %b
 %0 = call double @llvm.sqrt.f64(double %sub)
 %add = fadd double %sub3, %0
 %mul4 = fmul double 2.000000e+00, %a
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub5 = fsub double -0.000000e+00, %b
 %1 = call double @llvm.sqrt.f64(double %sub)
 %sub6 = fsub double %sub5, %1
 %mul7 = fmul double 2.000000e+00, %a
 %div8 = fdiv double %sub6, %mul7
 store double %div8, double* %r2, align 8
 br label %if.end15
```

```
if.end:
 %ret.0 = phi i32 [ 0, %if.then10 ], [ 1, %if.else14 ]
 br label %if.end15
```

```
if.end15:
 %ret.1 = phi i32 [ 0, %if.then ], [ %ret.0, %if.end ]
 ret i32 %ret.1
```

Figure 3.3: CFG for quadratic solver function after the memory to register pass. The only remaining memory accesses are the necessary storage of the results to `r1` and `r2`

shows the quadratic solver function code after the GVN Hoist pass, GVN pass, and the GVN Hoist pass again have been run.



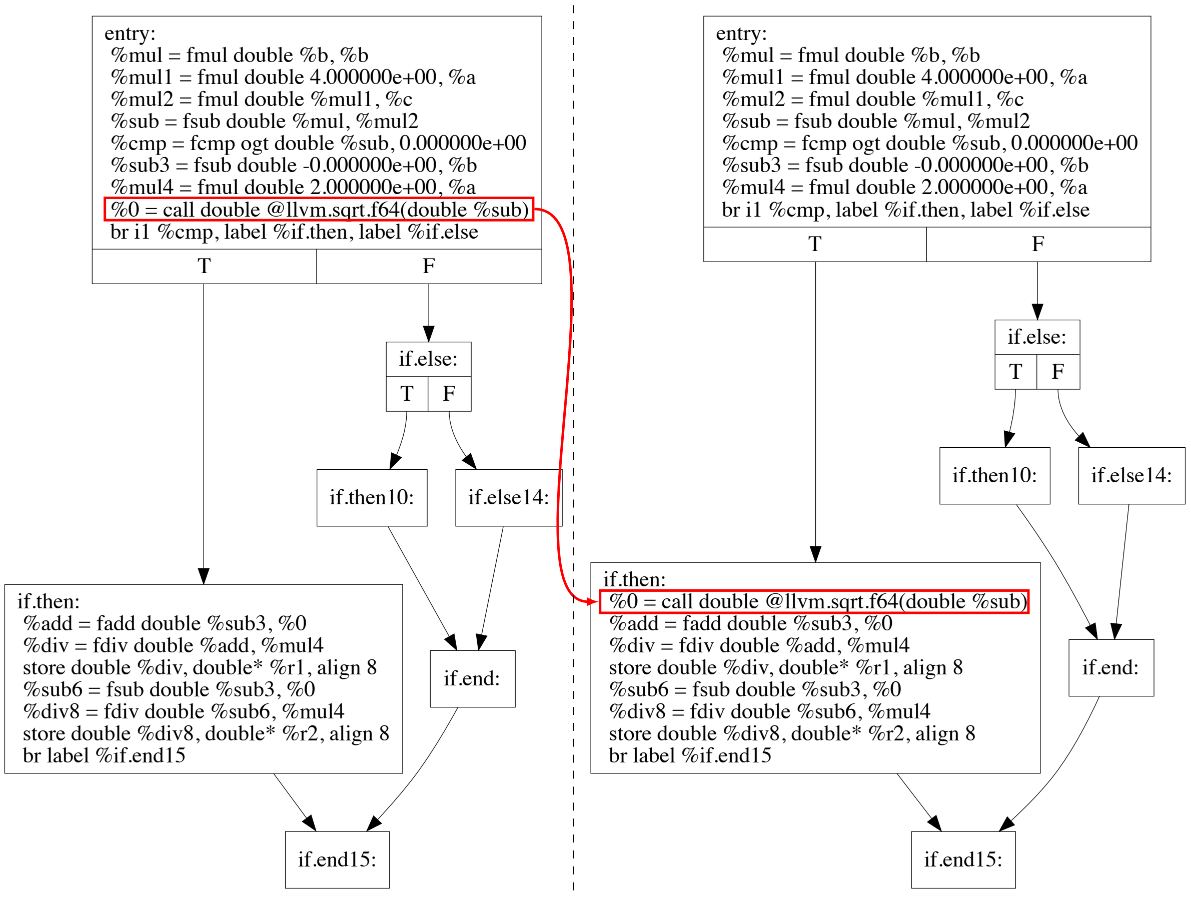
```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 br i1 %cmp, label %if.then, label %if.else

if.then:
 %sub3 = fsub double -0.000000e+00, %b
 %0 = call double @llvm.sqrt.f64(double %sub)
 %add = fadd double %sub3, %0
 %mul4 = fmul double 2.000000e+00, %a
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub5 = fsub double -0.000000e+00, %b
 %1 = call double @llvm.sqrt.f64(double %sub)
 %sub6 = fsub double %sub5, %1
 %mul7 = fmul double 2.000000e+00, %a
 %div8 = fdiv double %sub6, %mul7
 store double %div8, double* %r2, align 8
 br label %if.end15

if.else:

if.else14:

if.then10:
 %sub11 = fsub double -0.000000e+00, %b
 %mul12 = fmul double 2.000000e+00, %a
 %div13 = fdiv double %sub11, %mul12
 store double %div13, double* %r2, align 8
 store double %div13, double* %r1, align 8
 br label %if.end

if.end:

if.end15:
```

```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 %sub3 = fsub double -0.000000e+00, %b
 %mul4 = fmul double 2.000000e+00, %a
 %0 = call double @llvm.sqrt.f64(double %sub)
 br i1 %cmp, label %if.then, label %if.else

if.then:
 %add = fadd double %sub3, %0
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub6 = fsub double %sub3, %0
 %div8 = fdiv double %sub6, %mul4
 store double %div8, double* %r2, align 8
 br label %if.end15

if.else:

if.else14:

if.then10:
 %div13 = fdiv double %sub3, %mul4
 store double %div13, double* %r2, align 8
 store double %div13, double* %r1, align 8
 br label %if.end

if.end:

if.end15:
```

Figure 3.4: The instructions which the GVN Hoist and GVN passes found to Hoist are shown on the left in their original locations (in the `if.then10` BB and `if.then` BB) and on the right in their new location (in the `entry` BB). Redundant instructions were eliminated.

Before these passes, there were three separate instructions (`%sub11, %sub3, %sub5`) computing $0 - b$ (negative 'b'), two (`%mul12, %mul4`) computing $2 \times a$, and two (`%0, %1`) calling the square root function on `%sub`. After the passes, the redundant code was removed and hoisted to the common dominating BB.

The *GVN Sink* (`-gvn-sink`) pass moves code to later BBs when the code is only used in one of the branches. This can reduce code size, enable if-conversion, or ensure that code is only executed when the result is used. Figure 3.5 shows the quadratic solver function code after the GVN Sink pass. The call to the square root function, which was previously hoisted because there were redundant calls, was sunk to a lower BB because the other branch did not use the result.



```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 %sub3 = fsub double -0.000000e+00, %b
 %mul4 = fmul double 2.000000e+00, %a
 %0 = call double @llvm.sqrt.f64(double %sub)
 br i1 %cmp, label %if.then, label %if.else

if.then:
 %add = fadd double %sub3, %0
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub6 = fsub double %sub3, %0
 %div8 = fdiv double %sub6, %mul4
 store double %div8, double* %r2, align 8
 br label %if.end15
```

```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 %sub3 = fsub double -0.000000e+00, %b
 %mul4 = fmul double 2.000000e+00, %a
 br i1 %cmp, label %if.then, label %if.else

if.then:
 %0 = call double @llvm.sqrt.f64(double %sub)
 %add = fadd double %sub3, %0
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub6 = fsub double %sub3, %0
 %div8 = fdiv double %sub6, %mul4
 store double %div8, double* %r2, align 8
 br label %if.end15
```

Before GVN Sink pass After GVN Sink pass

Figure 3.5: The instruction which the GVN Sink pass found to move is shown on the left in its original location (in the `entry` BB) and on the right in its new location (in the `if.then` BB).

The *Merge Return* (`-mergereturn`) pass combines return instructions such that there is only one exit node for the function. In the quadratic solver example, there is already only one return instruction so this pass changes nothing.

The *Simplify CFG* (`-simplifycfg`) pass removes dead code and merges BBs. This pass cleans up BBs with no predecessors, combines BBs which are each other's only predecessor and successor, removes BBs which only contains an unconditional branch, and removes PHI nodes for BBs with one predecessor.

Figure 3.6 shows the quadratic solver function code after the Simplify CFG pass. The BBs which were unchanged have their code removed for clarity.

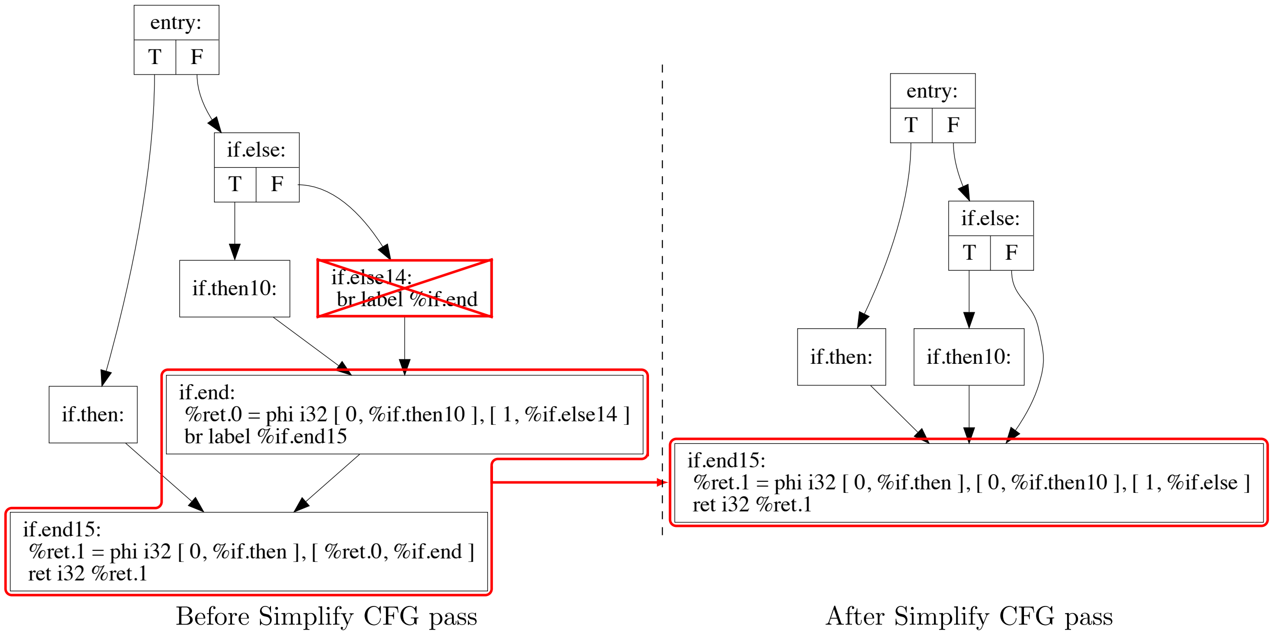
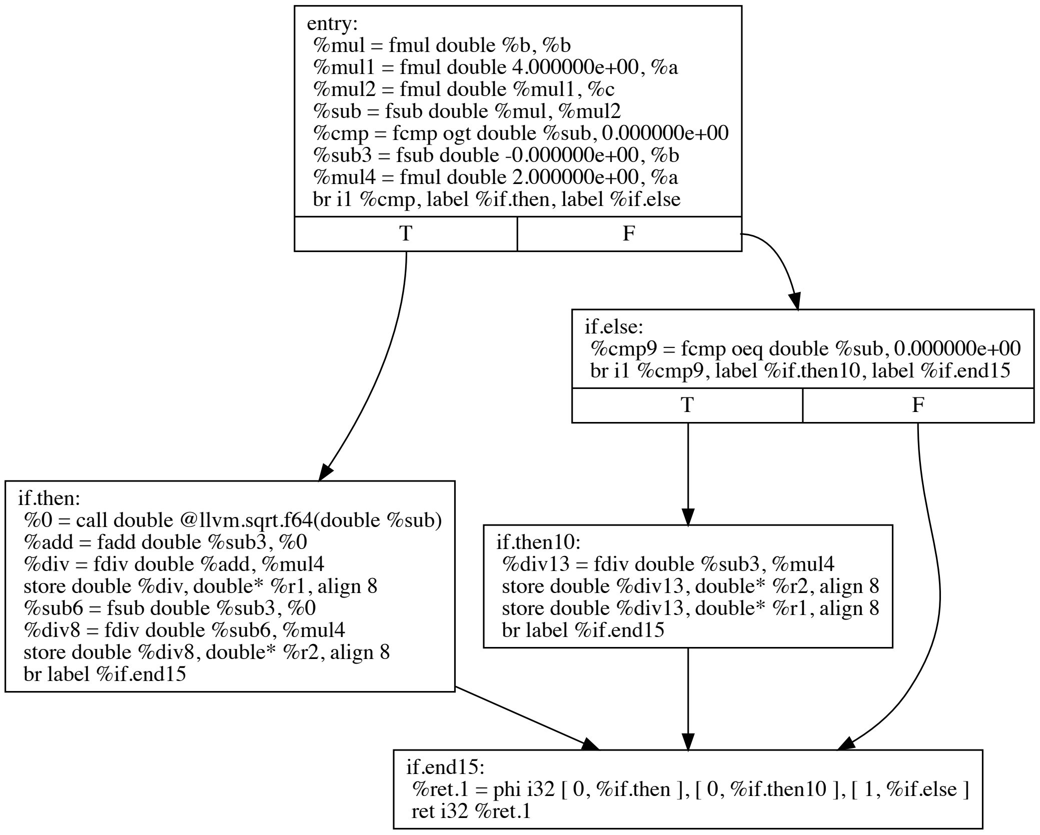


Figure 3.6: The BBs which were removed or combined by the Simplify CFG pass are shown on the left and the final result is shown on the right.

The `if.else14` BB was removed entirely because it was only a single unconditional branch statement. The `if.end` and `if.end15` BBs were combined because the `if.end` BB only contained a PHI node and an unconditional branch, so the PHI nodes from both BBs could be combined into a single PHI node.

The final result of all of these passes is shown in Figure 3.7. The code size is significantly smaller (24 instructions as opposed to 65), which in software is beneficial for many reasons, but in hardware translates to using less resources to produce the same result. In addition, reducing the number of BBs simplifies the control flow



```
entry:
 %mul = fmul double %b, %b
 %mul1 = fmul double 4.000000e+00, %a
 %mul2 = fmul double %mul1, %c
 %sub = fsub double %mul, %mul2
 %cmp = fcmp ogt double %sub, 0.000000e+00
 %sub3 = fsub double -0.000000e+00, %b
 %mul4 = fmul double 2.000000e+00, %a
 br i1 %cmp, label %if.then, label %if.else
```

```
if.else:
 %cmp9 = fcmp oeq double %sub, 0.000000e+00
 br i1 %cmp9, label %if.then10, label %if.end15
```

```
if.then:
 %0 = call double @llvm.sqrt.f64(double %sub)
 %add = fadd double %sub3, %0
 %div = fdiv double %add, %mul4
 store double %div, double* %r1, align 8
 %sub6 = fsub double %sub3, %0
 %div8 = fdiv double %sub6, %mul4
 store double %div8, double* %r2, align 8
 br label %if.end15
```

```
if.then10:
 %div13 = fdiv double %sub3, %mul4
 store double %div13, double* %r2, align 8
 store double %div13, double* %r1, align 8
 br label %if.end15
```

```
if.end15:
 %ret.1 = phi i32 [ 0, %if.then ], [ 0, %if.then10 ], [ 1, %if.else ]
 ret i32 %ret.1
```

Figure 3.7: The final form of the quadratic solver code and CFG after the optimization passes.

in both software and hardware. In software, this reduces the number of jumps and predictions (which could be incorrect) which slow down execution and in hardware, this simplifies the state machine required to control the circuit.

Chapter 4

Custom Back End

The original LegUp scheduler is split into first acquiring the Directed Acyclic Graph (DAG) and then using the SDC algorithm to create a Finite State Machine (FSM). The new scheduler was implemented in the same way. First the RDFG was extracted and then it was used to schedule the operations into the FSM.

4.1 Reduced Graph Extraction Pass

In order to obtain the reduced graph of the DFGs from input software code, a custom LLVM analysis pass was written. Eventually this pass was moved into the Verilog target code instead of being a standalone pass, but the functionality remained the same. The main algorithm to extract the reduced graph is shown in Algorithm 2.

In the loop starting on line 1, all of the instructions in the BB are iterated over. The body of this loop determines the level that instruction i belongs in by iterating over all of the uses (operands) of i (the loop starting on line 3). The IR is in SSA form, which means that every operation assigns to a unique variable. Due to this and the sequential nature of software, and because the instructions are iterated over in order, all uses of i will already have been added to the working set with their levels known. The level of i is set to the maximum of the levels of all its uses. While this is being computed, the $numUsers$ value of a use is decremented and the use is removed from the working set when there are no more users. This keeps the memory

Algorithm 2 Extracting the reduced graph from a BB

```
 1: for all Instructions $i$ in BasicBlock do
 2:     $i.level \leftarrow 0$
 3:     for all Uses $u$ of $i$ do
 4:         if $u.level > i.level$ then
 5:             $i.level \leftarrow u.level$
 6:         end if
 7:         if $u.numUsers == 1$ then
 8:             if $u$ is not a memory instruction then
 9:                 Remove $u$ from working set
10:             end if
11:         else
12:             $u.numUsers \leftarrow u.numUsers - 1$
13:         end if
14:     end for
15:     if $i$ is a memory instruction then
16:         for all Instructions $i2$ in memoryInst list do
17:             if $i$ depends on $i2$ then
18:                 if $i2.level > i.level$ then
19:                     $i.level \leftarrow i2.level$
20:                 end if
21:             end if
22:         end for
23:         Add $i$ to memoryInst list
24:     end if
25:     $i.level \leftarrow i.level + 1$
26:     if $i.numUsers > 0$ then
27:         $i.numUsers \leftarrow getNumUsers(i)$
28:         Add $i$ to working set
29:     end if
30:     Add $i$ to $i.level$ of the reduced graph
31: end for
```

footprint of this algorithm smaller than if all instructions were kept in memory at all times. However, if the instruction is a memory instruction like a load or store, it is kept in the working list because there may be a memory dependency with a later instruction. In the loop starting on line 16, if i is a memory instruction, the previous memory instructions are iterated over to determine if there is a memory dependency. If i is a store to the same address as $i2$, then there is either a Write After Read (WAR) or Write After Write (WAW) dependency. If i is a load and $i2$ is a store to the same address, then there is a Read After Write (RAW) dependency. If there is a dependency, the level of i is set to that of $i2$ if it greater than previous. Finally, the level is incremented by one so that it is in the level after all its uses, and then i is added to the working set and to the correct level of the reduced graph table.

4.2 RDFG Scheduler

The LegUp back end is implemented as a Verilog LLVM target. After a few custom passes are run on the IR, the *LegupPass* is executed, which performs the allocation, scheduling, binding, and Register-Transfer Level (RTL) generation steps, ultimately producing Verilog output [27]. The `Allocation` class reads in a Tcl file with the constraints such as the target device, timing, and other LegUp-specific options. These settings are stored in the global `LegupConfig` object so other portions of the code can access this information. The `Allocation` object is passed to all subsequent steps, so global information that needs to be accessed by other stages is stored in this class.

Next, each function is iterated over for scheduling. Here, LegUp would use the SDC algorithm to schedule the operations. Instead, this is where the reduced graph (RDFG) pass is run and the `ReducedDFG` object for each function is stored in a member variable of the `Allocation` object. Then the `scheduleOperations` function of the `GenerateRTL` object is called, which was modified to call the `RDFGScheduler`'s scheduling functions instead of those from the `SDCScheduler`.

The `runOnFunction` function of the `RDFGScheduler` iterates over the BBs of the function and calls a `createPipelines` function, then calls the `EstimateSchedule` function, using the results of the reduced graph pass, and finally after all BBs have been scheduled, calls the `createFSM` function.

The `createPipelines` function iterates over all of the instructions in the BB and creates a list of the matching FU by calling the `getOpNameFromInst` function of the `LegupConfig`. Originally, the schedule created by Legup's SDC scheduler was going to be used to find the best pipelines for use in the RDFG scheduler, but it was determined that this was impractical to do algorithmically. The pipelines used in [9] were hand selected out of existing architectures based on looking at the DFG. It was decided that since a pipeline is defined as having one or more stages, the pipelines in this case would each be the available FU.

The psuedocode for `EstimateSchedule`, which performs most of the scheduling functionality, is shown in Algorithm 3. The function is named after Algorithm 1 [9], but this function produces a complete schedule in the form of mapping of instructions to epochs.

Algorithm 3 $EstimateSchedule(L, T, Ops, p)$

```
1: span := 0
2: sch[][] := {{}}
3: for all i := 1 to $L$ do
4:     epochs, uses[] := $matching(T, Ops, p, i)$
5:     sch[][] := $addToSchedule(Sch, T, Ops, p,$ epochs, $span)$
6:     span := span + max{1, epochs}
7: end for
8: sch[terminator] := span - 1
9: return sch, span
```

Several modifications were made to the original algorithm. First, the L input, originally the set of levels, is now the number of levels, since the set would simply be a list of integers. Also, the set of stages in the pipelines, S, was eliminated, since in this case the pipelines are all just a single FU. For practicality purposes, the `matching` function returns both the uses and the epochs, since keeping the maximum epochs value while computing them is faster than finding the maximum later. Also, the `addToSchedule` function handles removing nodes from the graph at the same time as scheduling them. Finally, the terminator instruction (which is found in the `addToSchedule` function) is scheduled after all other levels so that it executes in the last epoch.

The pseudocode for the `matching` function is shown in Algorithm 4. The opera-

Algorithm 4 *matching*$(T, Ops, p, level)$

```
1:  epochs := 0
2:  for all $op$ in $T$ do
3:      funame := $p[op.opcode]$
4:      uses[$op$] := $Ops[level][op]$
5:      if uses[$op$]$.size() > 0$ then
6:          latency := getOperationLatency(funame)
7:          if $op$ is a PHI node then
8:              totalTime := 1 + latency
9:          else
10:             totalTime := uses[$op$]$.size()$ + latency
11:         end if
12:     else
13:         totalTime := 0
14:     end if
15:     epochs := max{epochs, totalTime}
16: end for
17: return epochs, uses
```

tions in the operation set T are iterated over and matched to a FU in p. Then the set of instructions in the reduced graph are placed in this level's uses set. If there are one or more instructions, the latency is computed by quering the `LegupConfig` object. This latency is the number of extra cycles an operation may take, for instance a multiply takes two cycles to complete so the latency is one. Since all the PHI nodes should execute in the first cycle of a BB, the total time is just the latency plus one. However, for other operations that may execute one after another, the total time is the number of times the operation is used in the level, plus the extra latency. The epochs variable is a running maximum of this totalTime for each operation. This makes sure that each level is allotted enough epochs so that multi-cycle operations complete before the next level begins.

The psuedocode for the `addToSchedule` function is shown in Algorithm 5. This function iterates from the current value of span, which is the next epoch that can be scheduled into, to span+epochs, which is the last epoch this level should schedule in. For each of these epochs, the operations in the uses set are iterated over. If there are

Algorithm 5 $addToSchedule(Sch, T, Ops, p, epochs, span)$

```
1: for i from span to epochs + span do
2:     for all op in uses do
3:         if op.size() > 0 then
4:             instr := op.back()
5:             if instr is a terminator then
6:                 terminator := instr
7:                 Remove instr from op list
8:             else if instr is a PHI node then
9:                 for all  dophi in op list
10:                    sch[phi] := 0
11:                end for
12:                Empty op list
13:            else
14:                sch[instr] := i
15:                Remove instr from op list
16:            end if
17:        end if
18:    end for
19: end for
20: return sch
```

instructions of this op type, the last one is tested to see if it is a terminator instrucion, a PHI node, or any other instruction. If it is a terminator, it is removed from the list and stored to be mapped later because it needs to be scheduled in the last epoch regardless of how early it is ready. If it is a PHI node, all instructions in this use are mapped to epoch 0 and removed from the use list, because they all need to execute in the first cycle of the BB. Otherwise, the one instruction is mapped to the current epoch and removed from the list. This way, the next instruction of this use will be scheduled in the next epoch until they are all scheduled.

The LegUp scheduler outputs the schedule for a function in the form of an FSM which maps instructions to execution states. For the RTL generation and Verilog writer to work, the RDFG scheduler must also produce an FSM of the same format. To create this FSM, first an empty state is created for all BBs. Then the span of each BB's schedule is used to add that many empty states to the BB with the correct transitions between them. Then, the instructions in the BB are iterated over and inserted into the state that matches the epoch they were mapped to. The end state of each instruction is set properly to account for latency. Later, transition variables are added to the terminating states of each BB so that control flow will execute properly.

Chapter 5

Results

To test the efficacy of the RDFG scheduler, five applications were compiled from a software model to a hardware module. These applications were three examples included in the LegUp 4.0 distribution: a Matrix Multiplication (MM), Finite Impulse Response (FIR) filter, Fast Fourier Transform (FFT), and two additional examples: Number Theoretic Transform (NTT) multiplier [28], and Cholesky decomposition [29]. For these applications, three metrics will be discussed and compared against the baseline, LegUp case: the execution time for varying input sizes as a metric of the schedule's efficiency, the time taken to find the schedule, and the memory usage during this process.

The MM application implemented the definition shown in (5.1).

$$\text{If } A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \text{ and } B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix},$$

$$C = AB \text{ such that } c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \text{ for } i = 1, \cdots, n \text{ and } j = i, \cdots, n \quad (5.1)$$

Psuedocode for the MM application is shown in Algorithm 6. The MULTIPLY function was selected as the hardware accelerated function. The width and height of the input,

A and B, and output, C, matrices is denoted as n, as in the definition. A and B were filled with non-zero values. All matrices were stored in global memory so that they could be accessed by both software and hardware.

Algorithm 6 Matrix Multiply

```
1: $A[n][n]$ := non-zero data
2: $B[n][n]$ := non-zero data
3: $C[n][n]$
4: function MULTIPLY($i$, $j$)
5:     $sum := 0$
6:     for $k$ from 0 to $n$ do
7:         $sum := sum + A[i][k] * B[k][j]$
8:     end for
9:     $C[i][j] := sum$
10: end function
11: function MAIN
12:     for $i$ from 0 to $n$ do
13:         for $j$ from 0 to $n$ do
14:             MULTIPLY($i$, $j$)
15:         end for
16:     end for
17: end function
```

The FIR application implemented the definition shown in (5.2).

$$y[n] = b_0x[n] + b_1x[n-1] + \cdots + b_Nx[n-N] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{5.2}$$

The output signal is $y[n]$, the input signal is $x[n]$, and b_i is the coefficient vector. Pseudocode for the FIR filter application is shown in Algorithm 7. The FIR function was selected as the hardware accelerated function. ITERS was the number of outputs, $y[n]$ computed. N was the order or number of past samples to use in the FIR calculation. xn is the value of the new sample for the current iteration, $x[n]$. $xprev$ was the array of past samples. For testing, the MAIN function initialized the past samples to zeros and the coefficients were zero through fifteen and then all zeroes for any remaining coefficients. For each iteration, the new input sample was incremented by one.

Algorithm 7 FIR Filter

```
function FIR($xn$, $b[]$, $xprev[]$)
    $sum := 0$
    for $j$ from $N$-1 to 0 do
        $xprev[j] := xprev[j-1]$
    end for
    $xprev[0] := xn$
    if $xprev[N$-1$] == 0$ then
        return 0
    else
        $sum := 0$
        for $j$ in 0 to $N$ do
            $sum := sum + xprev[N - j - 1]*b[j]$
        end for
        return $sum$
    end if
end function
function MAIN
    $xprev[N]$ := zeros
    $b[N] := [0, 1, \cdots, 15, 0, \cdots, 0]$
    $y$[ITERS]
    for $i$ from 0 to ITERS do
        $y[i-1] :=$ FIR($i$, $b$, $xprev$)
    end for
end function
```

The FFT application implemented the definition shown in (5.3).

$$X_k = \sum_{n=0}^{N-1} x_n \exp\left(\frac{-2\pi j}{N} kn\right),\ k = 0, \cdots, N-1 \tag{5.3}$$

The input size is N, the input values are x and the output values are X. In the software C code, this was implemented in two main steps: time decimation and the butterfly operation. Time decimation is a reordering of the input data based on even or odd indices. The butterfly operation breaks the computation down into smaller pieces. The C code also optimizes the sine computation using a lookup table.

Psuedocode for the NTT application is shown in Algorithm 8. The order of the input (a and b) and output (c) polynomials is n. The implementation is based on the

Algorithm 8 NTT Based Modular Polynomial Multiplication [28]

```
Require: Polynomials $a(x)$ and $b(x)$ of maximum degree $n$ with coefficients $a_i, b_i \in$
    $\mathbb{Z}_p$ for $i = 0, 1, \ldots, n-1$
Ensure: $c(x) = a(x) \cdot b(x) \mod (x^n + 1)$
 1: Pre-calculate look-up tables for consecutive powers of $\theta$, $\omega$, $\theta^{-1}$, and $\omega^{-1}$
 2: weight_coeff : for $i \leftarrow 0$ to $n$ do
 3:     $a_i \leftarrow a_i \cdot \theta^i \mod p$
 4:     $b_i \leftarrow b_i \cdot \theta^i \mod p$
 5: end for
 6: FFT($a$, $\omega$)
 7: FFT($b$, $\omega$)
 8: mult_coeff : for $i \leftarrow 0$ to $n$ do
 9:     $c_i \leftarrow a_i \cdot b_i \mod p$
10: end for
11: IFFT($c$, $\omega^{-1}$)
12: unweight_coeff : for $i \leftarrow 0$ to $n$ do
13:     $c_i \leftarrow c_i \cdot \theta^{-i} \mod p$
14: end for
```

Schönhage-Strassen algorithm. The FFT function implements the Gentleman-Sande algorithm and the IFFT function implements the Cooley-Tukey algorithm. The algorithm was tested with a maximum polynomial degree of 32,768 with 32-bit coefficients.

The Cholesky decomposition application implemented the finding of a lower triangular matrix, L, which satisfies (5.4).

$$A = LL^T \tag{5.4}$$

The Cholesky factor, L, if the input matrix, A, is symmetric and positive definite. The software implementation was generic for any valid $n \times n$ A matrix.

5.1 Schedule Length

The five test applications were organized in the same way. All of the algorithms were implemented as functions which were called by a "testbench" main function with stimulus. These applications were compiled using the hybrid flow offered by LegUp, with the function implementing the algorithm chosen as the accelerator.

To evaluate the quality of the schedules created by both the RDFG and SDC schedulers, both versions of LegUp were used to generate HDL for each of the five test algorithms. This way, everything would be identical except for the schedule of the accelerated function. Both versions were then simulated and the number of clock cycles used to complete the application were recorded for varying input sizes. The input sizes were increased until it became impractical to run the test.

The MM schedule was identical when generated by both the RDFG and SDC schedulers, so the number of cycles was exactly the same in all cases.

On the other hand, the schedule for FIR converged to an approximately 5.1% difference in the number of cycles for sufficiently large sizes, as shown in Table 5.1 and Figure 5.1. Upon examination, it was observed that —as expected— the differences in schedule corresponded to the inner loops of the code, resulting in a proportionally growing number of additional clock cycles.

Table 5.1: FIR Execution Cycles Results

Taps	RDFG Cycles	SDC Cycles	Difference (%)
16	15,747	15,141	4.00
32	51,749	49,586	4.36
64	182,165	173,713	4.87
128	695,116	661,732	5.05
256	2,728,123	2,596,048	5.09
512	10,824,857	10,299,039	5.11
1,024	43,229,169	41,130,281	5.10

The CFG of the FIR function is shown in Figure 5.2. The only BB that was scheduled differently between the SDC and RDFG scheduler was `BB_preheader`. The schedules for this BB are shown in Figure 5.3. Each instruction is labeled with the same letter in each schedule. The difference in the schedules is with the scheduling of the instruction labeled ‘J’, a load which depends on the output from the instruction

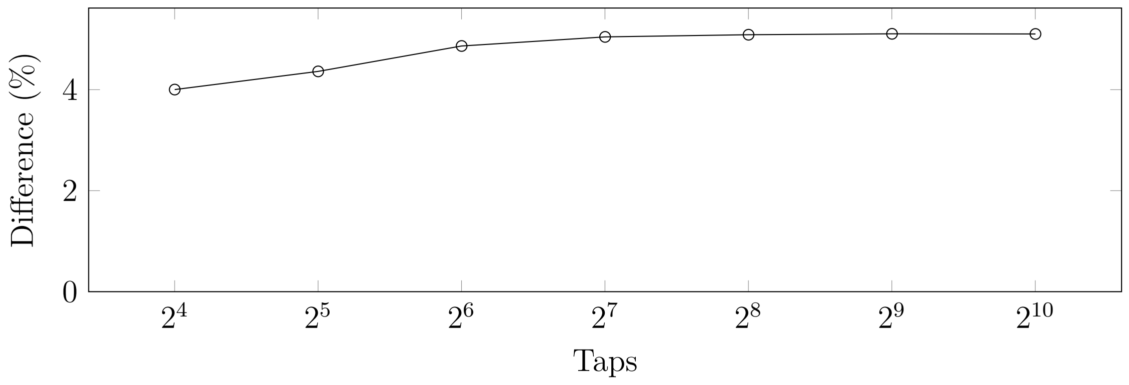


Figure 5.1: Percent increase in execution cycles for various input sizes in the FIR example

labeled ‘F’. In Figure 5.3b, the levels of the graph are outlined with bold lines. To see where the levels come from, the DFG for this BB is shown in Figure 5.4. The instructions labeled ‘F’, ‘G’, and ‘H’ are in the same level, so ‘J’ must be scheduled after all of these instructions have completed, since the RDFG only has enough information to guarantee dependencies are satisfied if all instructions in a level are allowed to finish before the next is allowed to begin. The problem with this is that ‘J’ only needs the output from ‘F’, but ‘G’, another load, takes two extra cycles to finish. This means that, in this case, the RDFG schedule is two cycles longer.

Since the schedules of all of the BBs are the same aside from these two extra cycles in the BB of inner most loop, the percent increase in execution cycles converges to approximately 5.1%. As the input size increases, time spent in the inner most loops becomes much greater than time spent elsewhere. This leads to the percent increase converging on a constant value for large enough input sizes.

The execution time results for the FFT example are shown in Table 5.2 and Figure 5.5.

Because the FFT is three nested loops, BB have varying levels of effect on the total number of execution cycles. Several of the BB are scheduled differently. The differences in the outer loops have a greater effect until the 2048 input size. From the 4096 input size, the total execution cycles drops because the effect of the outer-loop

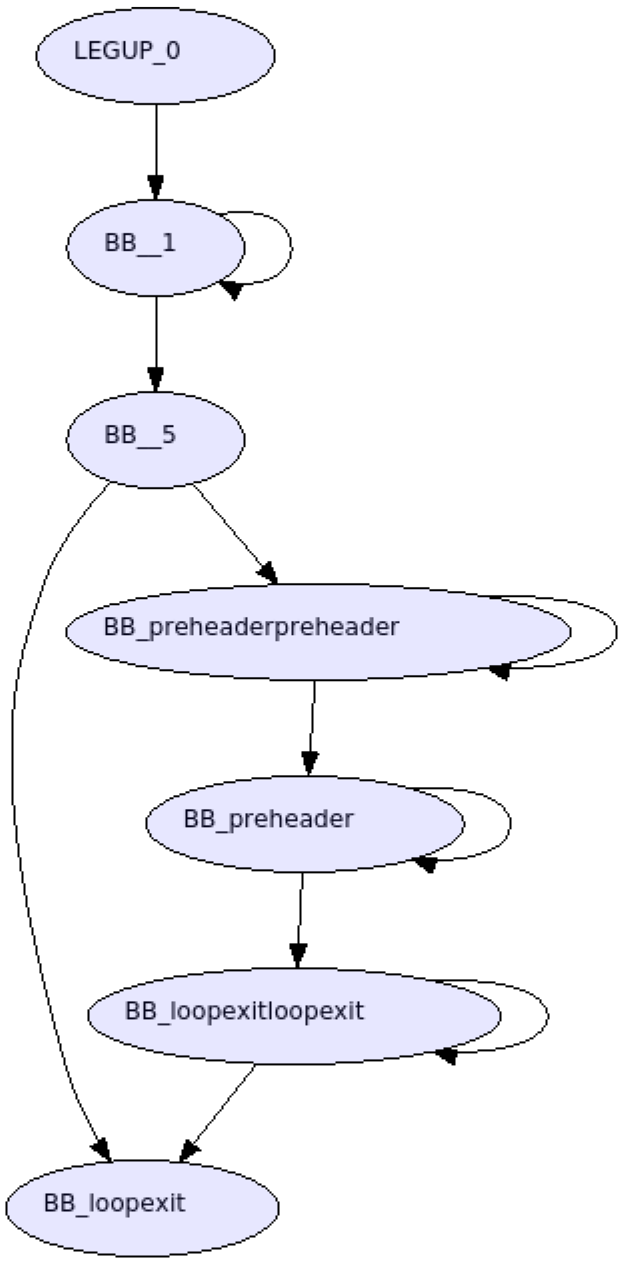
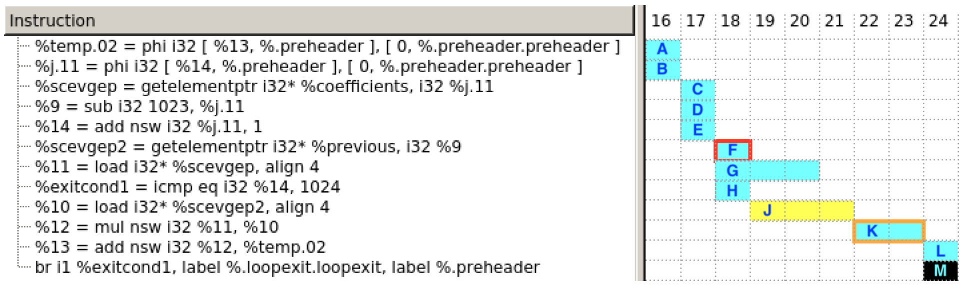
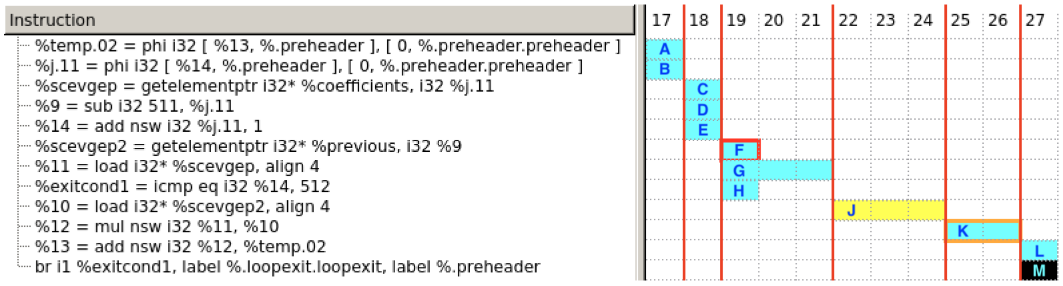


Figure 5.2: CFG for the FIR example



```
Instruction
%temp.02 = phi i32 [ %13, %.preheader ], [ 0, %.preheader.preheader ]
%j.11 = phi i32 [ %14, %.preheader ], [ 0, %.preheader.preheader ]
%scevgep = getelementptr i32* %coefficients, i32 %j.11
%9 = sub i32 1023, %j.11
%14 = add nsw i32 %j.11, 1
%scevgep2 = getelementptr i32* %previous, i32 %9
%11 = load i32* %scevgep, align 4
%exitcond1 = icmp eq i32 %14, 1024
%10 = load i32* %scevgep2, align 4
%12 = mul nsw i32 %11, %10
%13 = add nsw i32 %12, %temp.02
br i1 %exitcond1, label %.loopexit.loopexit, label %.preheader
```

(a) SDC Scheduler



```
Instruction
%temp.02 = phi i32 [ %13, %.preheader ], [ 0, %.preheader.preheader ]
%j.11 = phi i32 [ %14, %.preheader ], [ 0, %.preheader.preheader ]
%scevgep = getelementptr i32* %coefficients, i32 %j.11
%9 = sub i32 511, %j.11
%14 = add nsw i32 %j.11, 1
%scevgep2 = getelementptr i32* %previous, i32 %9
%11 = load i32* %scevgep, align 4
%exitcond1 = icmp eq i32 %14, 512
%10 = load i32* %scevgep2, align 4
%12 = mul nsw i32 %11, %10
%13 = add nsw i32 %12, %temp.02
br i1 %exitcond1, label %.loopexit.loopexit, label %.preheader
```

(b) RDFG Scheduler

Figure 5.3: The schedules produced for the main loop body (`BB_preheader`) of the FIR example

BBs is hidden by the cycles spent in the innermost loops.

The execution time results for the NTT example is presented in Table 5.3 and Figure 5.6. The structure of this code has three separate triple-nested loops. In the first two, the BBs of the inner most loops were scheduled with a total of 26 cycles by SDC and with 28 cycles by RDFG and the innermost loop of the last triple-nested loop was scheduled with 27 cycles by SDC and 28 cycles by RDFG. A few BBs at

Table 5.2: FFT Execution Cycles Results

Points	RDFG Cycles	SDC Cycles	Difference (%)
64	17,496	16,885	3.62
128	37,773	36,421	3.71
256	79,471	76,483	3.91
512	169,318	162,876	3.96
1024	393,296	377,159	4.28
2048	815,031	779,564	4.55
4096	4,220,049	4,128,284	2.22
8192	9,371,660	9,161,584	2.29

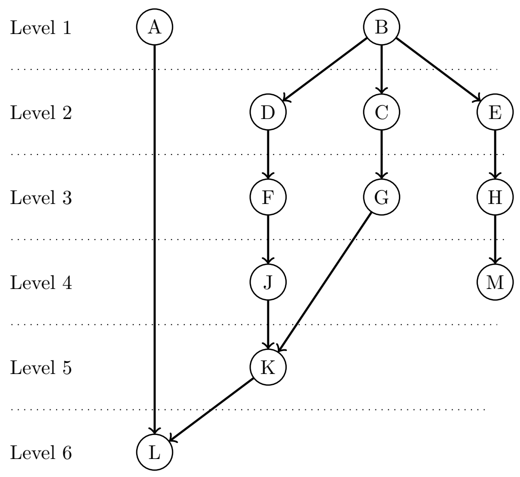


Figure 5.4: DFG of `BB_preheader` in the FIR example

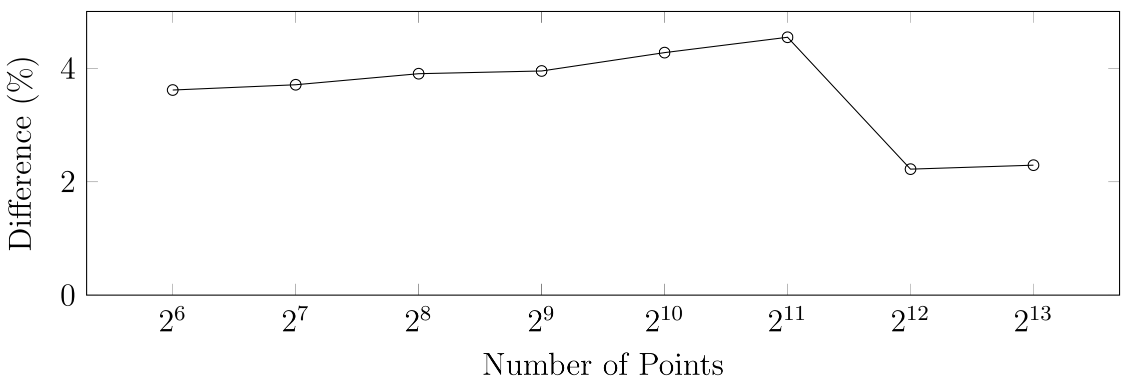


Figure 5.5: Percent increase in execution cycles for various input sizes in the FFT example

Table 5.3: NTT Execution Cycles Results

Input Polynomials Degree×Bit width	RDFG Cycles	SDC Cycles	Difference (%)
128×32	78,159	75,433	3.61
256×32	162,114	155,923	3.97
512×32	344,188	330,289	4.21
1024×32	736,209	705,247	4.39
2048×32	1,796,093	1,728,572	3.91
4096×32	6,675,640	6,601,184	1.13
8192×32	14,393,095	14,231,037	1.14
16384×32	30,910,366	30,558,420	1.15
32768×32	66,076,673	65,327,517	1.15

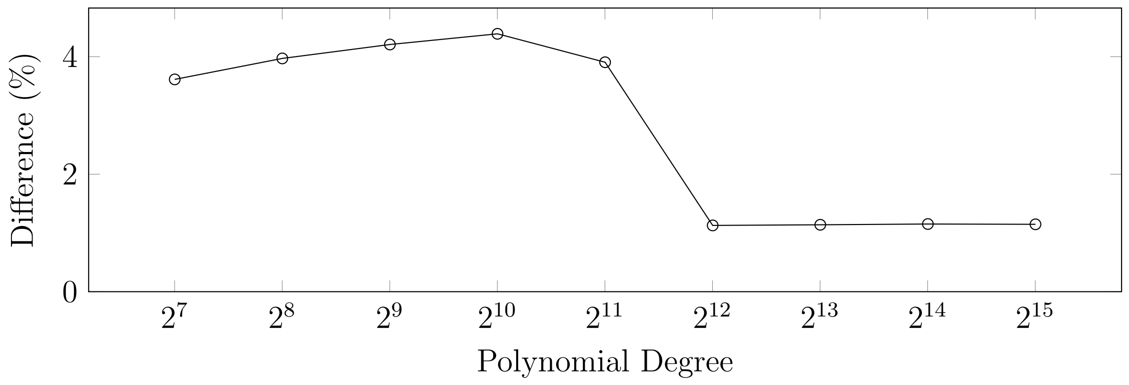


Figure 5.6: Percent increase in execution cycles for various input sizes in the NTT example

the top level of the program were scheduled with one cycle more when scheduled by RDFG. The increase in the outer levels is shown in the initial increase of the overall execution cycles, but since the innermost loops have only one or two extra cycles, the overall percent increase drops and converges to approximately 1%.

The execution time results for the Cholesky decomposition example are presented in Table 5.4 and Figure 5.7. In this test, the worst case percent increase of execution cycles was 5.8%. This drops after a matrix of size 10×10 and continues dropping. The example became impractical to run for sizes larger than 100×100 so the convergence point was not found, but at the last point only a percent increase of 2% was observed.

Table 5.4: Cholesky Execution Cycles Results

Matrix Size	RDFG Cycles	SDC Cycles	Difference (%)
3×3	11,087	10,591	4.68
5×5	17,086	16,213	5.39
10×10	39,916	37,706	5.86
20×20	128,868	122,808	4.94
50×50	1,051,616	1,023,749	2.72
100×100	6,773,112	6,674,957	1.47

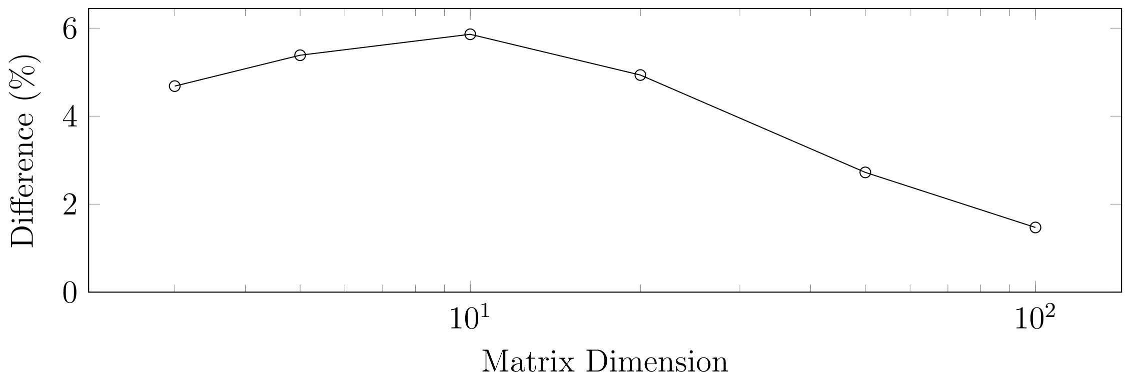


Figure 5.7: Percent increase in execution cycles for various input sizes in the Cholesky example

5.2 Time Efficiency

In order to measure the time difference of the proposed scheduling algorithm, RDFG, against the original LegUp scheduler, SDC, both schedulers were wrapped in timing code which ran each scheduler for 10,000 iterations. The tests were run on an Ubuntu 14.04 VM with 4GB of memory on an AMD A10-5800K APU. The results for the tests are shown in Table 5.5. The number of BBs in each test is listed to show the relative complexity of each application. The *Speedup* achieved by using the RDFG as opposed

Table 5.5: Timing results

Test	BBs	RDFG Time (ms)	SDC Time (ms)	Speedup
MM	3	0.12	0.67	5.69
FIR	7	0.42	4.91	11.79
FFT	13	0.96	12.79	13.29
NTT	42	2.41	31.18	12.93
Cholesky	70	5.31	186.70	35.16

to the SDC ranges from 5x to 35x faster to resolve the schedule, with larger speedup for more complex applications. For the SDC algorithm, if V is the number of vertices in the CDFG (the operations), then the number of scheduling variables n is $O(|V|)$ and the number of constraints m is $O(|V|^2)$. The SDC scheduling problem has a complexity of $O(n^2(m + n \log n) \log n)$ [30]. Because the RDFG scheduling algorithm processes the reduced graph by iterating over each of the L levels of the graph and within each level iterated over the T types of operations needed, the complexity is $O(|L| \times |T|)$ [9]. It is expected that the time benefits of the RDFG scheduler will be greater for larger, more complex applications.

5.3 Memory Efficiency

To measure the memory performance of the RDFG scheduling algorithm, the memory allocation functions in both the modified and unmodified copies of LegUp were overloaded to record stack traces and sizes for every allocation and deallocation as shown in Listing 5.1.

```
void* operator new(std::size_t sz) {
    // Allocate the memory
    void * requestedMemory = std::malloc(sz);
    // Write the allocs to file
    std::ofstream& memoryProfile = resultFile();
    memoryProfile << "Allocation, size = " << sz << " at "
        << static_cast<void*>(requestedMemory) << std::endl;
    // Stack trace of allocation
    dumpStackTrace(memoryProfile);
    memoryProfile << "-----------" << std::endl;
    // Return the alloc'd memory
    return requestedMemory;
}

void operator delete(void * p) {
    // Write the deallocs to file
    std::ofstream& memoryProfile = resultFile();
    memoryProfile << "Deallocation at "
        << static_cast<void*>(p) << std::endl;
    // Stack trace of deallocation
    dumpStackTrace(memoryProfile);
    memoryProfile << "-----------" << std::endl;
    // Deallocate the memory
    free(p);
}
```

Listing 5.1: Memory profiling instrumentation

This instrumentation generated huge unreadable files, so these files were parsed to strip out anything except the scheduling functions. After each allocation or deallocation, the running total of currently allocated memory was printed to a new file. These data points were saved to be plotted. Graphs showing the total memory usage are shown in Figures 5.8 - 5.12. The x-axis is the index of the allocation/deallocation. The y-axis is the total allocated memory at that allocation/deallocation. The RDFG plots appear to end earlier than the SDC graphs, since the RDFG algorithm performs fewer allocations. This also aligns with the fact that the RDFG algorithm takes less time.

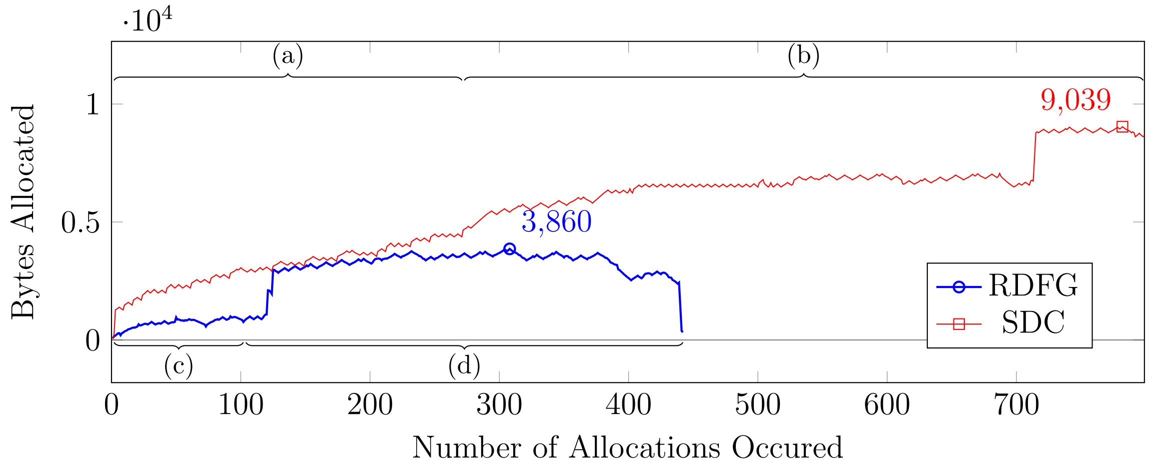


Figure 5.8: Matrix multiply memory usage (peak memory usage labeled)

Because the schedules in the MM example were identical, this graph shows the best comparison of the memory usage. On Figure 5.8, the graph extraction and actual scheduling phases are presented. It can be seen that the RDFG extraction (c) takes significantly less memory than the DAG extraction (a), since only portions of the graph are stored temporarily for each epoch, and then discarded. The DAG however must store the entire graph at once, and it cannot be deallocated for at least the duration of scheduling, in order to track every possible dependency in the graph. The RDFG scheduler (d) also deallocates portions of the RDFG as the schedule is created, while the SDC scheduler (b) retains all information throughout the duration of scheduling.

For the FIR test, the SDC scheduler had a peak memory usage of 27,487 bytes. The RDFG scheduler used 5,253 bytes, 19.1% of the SDC usage, as shown in Figure 5.9.

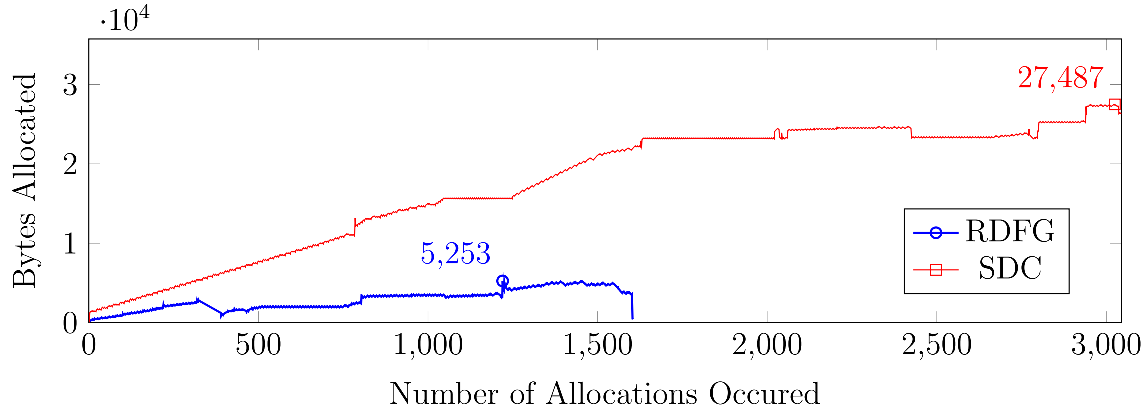


Figure 5.9: FIR memory usage (peak memory usage labeled)

The SDC scheduler on the FFT test used a maximum of 60,685 bytes. The RDFG scheduler used 10,645 bytes, 17.5% of the SDC usage, as shown in Figure 5.10.

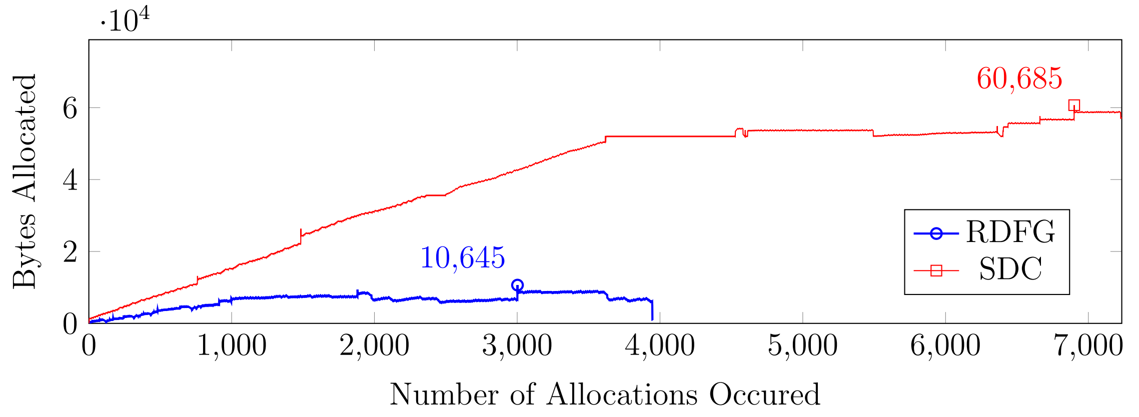


Figure 5.10: FFT memory usage (peak memory usage labeled)

The NTT test needed a peak of 111,645 bytes to scheduler with the SDC algorithm. The RDFG scheduler used 19,097 bytes, 17.1% of the SDC usage, as shown in Figure 5.11.

The most complex example, the Cholesky decomposition, showed the best memory savings. The SDC scheduler used a peak of 222,269 bytes. The RDFG scheduler used at most 35,107 bytes, 15.8% of the SDC usage, as shown in Figure 5.12.

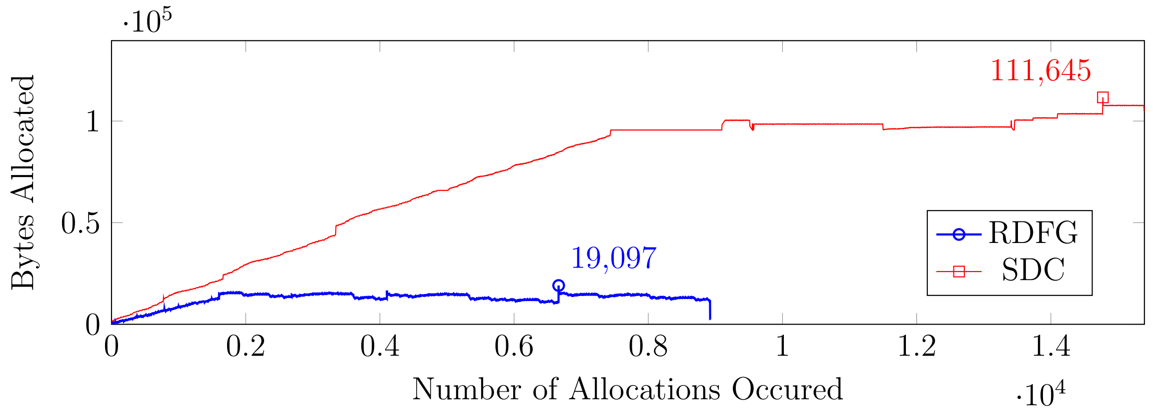


Figure 5.11: NTT memory usage (peak memory usage labeled)

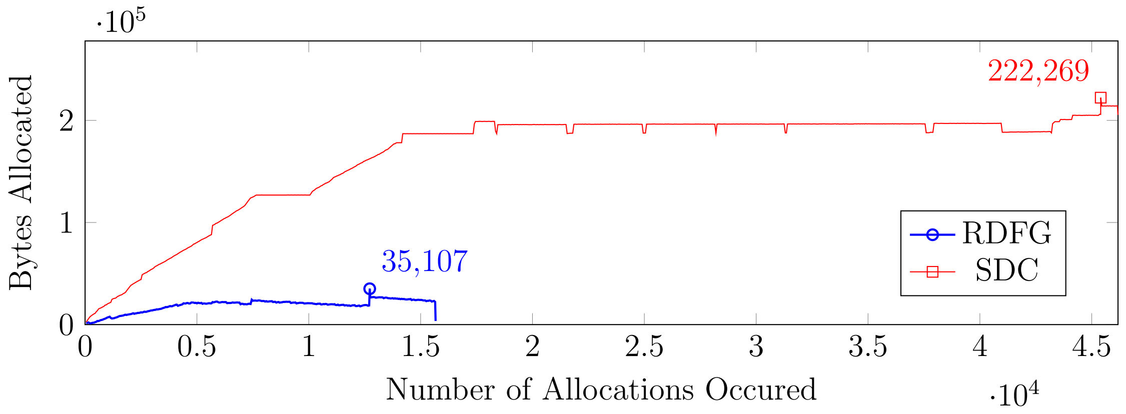


Figure 5.12: Cholesky memory usage (peak memory usage labeled)

A summary of the peak memory usage for each test with the SDC and the RDFG schedulers is shown in Table 5.6. It can be seen that in general, more complex

Table 5.6: Memory results

Test	BBs	RDFG Peak Bytes	SDC Peak Bytes	Percent (%) RDFG of SDC
MM	3	3,860	9,039	42.7
FIR	7	5,253	27,487	19.11
FFT	13	10,645	60,685	17.54
NTT	42	19,097	111,645	17.11
Cholesky	70	35,107	222,269	15.79

algorithms with more BB show more memory savings when scheduled with the RDFG scheduler instead of the SDC scheduler. Even the simplest algorithm, the MM, used less than half the memory.

Chapter 6

Conclusion

Scheduling is a key piece in creating efficient hardware systems from software applications. It is, however, time and memory consuming. In this research a new RDFG scheduling approach that can be used as part of the HLS process was presented. The reduced graph used for the proposed scheduler extracts the operations in the CDFG and only partially considers dependencies by placing these in levels that determine the order of execution. The RDFG scheduler was directly compared to the SDC scheduler used by the LegUp HLS tool. The original scheduler was replaced such that the LegUp tool used the RDFG approach and five test applications were scheduled using the original SDC scheduler and the custom RDFG scheduler. The schedules produced by the RDFG approach achieve up to 35$\times$ speedup in the scheduling process with less than 20% memory usage for sufficiently complex applications. The execution times of the synthesized hardware benchmark circuits use 0 to 6% additional clock cycles. Future work may include evaluating the performance of the scheduler using a published benchmark suite. Also, in order to minimize the extra execution cycles, a hybridized scheduler may be investigated which uses heuristic schedulers on critical, inner loop BBs, but the efficient RDFG scheduler on the majority of BBs.

Bibliography

[1] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, Jul. 2009. [Online]. Available: http://ieeexplore.ieee.org/document/5209958/

[2] "Xilinx Vitis," 2019. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[3] "Intel oneAPI Toolkits," 2019. [Online]. Available: https://software.intel.com/oneAPI

[4] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? A case study on matrix multiplication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2013, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/document/6732298/

[5] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8356004/

[6] 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," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7368920/

[7] S. Logesh, D. S. Harish Ram, and M. Bhuvaneswari, "A Survey of High-Level Synthesis Techniques for Area, Delay and Power Optimization," *International Journal of Computer Applications*, vol. 32, no. 10, pp. 1–6, 2011.

[8] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling," *ACM Computing Surveys*, vol. 45, no. 4, pp. 1–36, Aug. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2501654.2501664

[9] S. Skalicky, S. Lopez, M. Lukowiak, and C. Wood, "Mission control: A performance metric and analysis of control logic for pipelined architectures on FPGAs," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, Dec. 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7032539/

[10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[Online]. Available: http://dl.acm.org/citation.cfm?id=2514641.2514740http://dl.acm.org/citation.cfm?doid=2514641.2514740

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2007.

[12] D. Koch, F. Hannig, and D. Ziener, Eds., *FPGAs for Software Programmers.* Cham: Springer International Publishing, 2016. [Online]. Available: http://link.springer.com/10.1007/978-3-319-26408-0

[13] "LLVM Compiler Infrastructure." [Online]. Available: https://llvm.org/docs/index.html

[14] Xilinx Inc., *Vivado Design Suite User Guide*, Xilinx, 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug903-vivado-using-constraints.pdf

[15] Intel Corp., "Intel ® High Level Synthesis Compiler Best Practices Guide," 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf

[16] Mentor Graphics, "Handel-C Synthesis Methodology," 2018. [Online]. Available: https://www.mentor.com/products/fpga/handel-c/

[17] B. Inc, "BSV High-Level HDL," 2017. [Online]. Available: http://bluespec.com/54621-2/

[18] C. Pilato and F. Ferrandi, "Bambu: A Free Framework for the High Level Synthesis of Complex Applications," 2012. [Online]. Available: https://panda.dei.polimi.it/wp-content/uploads/PosterUB_DATE.pdf

[19] "GAUT - High-Level Synthesis tool," 2013. [Online]. Available: http://www.gaut.fr/

[20] "MyHDL," 2018. [Online]. Available: http://www.myhdl.org/

[21] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines.* IEEE, 2010, pp. 127–134. [Online]. Available: http://ieeexplore.ieee.org/document/5474060/

[22] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, Mar. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4133993/

[23] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[24] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, vol. 2018-Janua. IEEE, Dec. 2017, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/8279778/

[25] P. Sarkar, A. Sengupta, and M. K. Naskar, "GA driven integrated exploration of loop unrolling factor and datapath for optimal scheduling of CDFGs during high level synthesis," in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, May 2015, pp. 75–80. [Online]. Available: http://ieeexplore.ieee.org/document/7129163/

[26] S. Dutt and O. Shi, "A fast and effective lookahead and fractional search based scheduling algorithm for high-level synthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2018, pp. 31–36. [Online]. Available: http://ieeexplore.ieee.org/document/8341975/

[27] "LegUp 4.0 Documentation," pp. 1–37, 2015. [Online]. Available: http://legup.eecg.utoronto.ca/docs/4.0/

[28] K. Millar, M. Lukowiak, and S. Radziszowski, "Design of a Flexible Schonhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis to Accelerate HE in the Cloud," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2019.

[29] "Cholesky decomposition." [Online]. Available: https://rosettacode.org/wiki/Cholesky_decomposition

[30] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd annual conference on Design automation - DAC '06*. New York, New York, USA: ACM Press, 2006, p. 433. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1146909.1147025