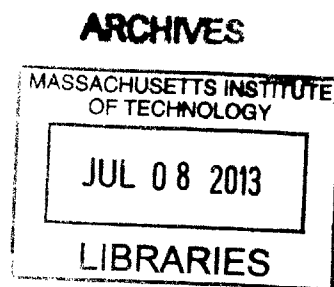


A Statically Scheduling Compiler for a Parameterized Numerical Accelerator

by

Andrew Charles Wright

B.S., University of Florida (2011)



Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author *ACW*
Department of Electrical Engineering and Computer Science
May 21, 2013

Certified by
[Signature]
Vladimir Stojanović
Associate Professor
Thesis Supervisor

Accepted by
[Signature]
Leske Kolodziejki
Chairman, Department Committee on Graduate Theses

A Statically Scheduling Compiler for a Parameterized Numerical Accelerator

by

Andrew Charles Wright

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In this work, I present a statically scheduling compiler for a numerical accelerator that parallelizes and maps algorithms to instances of a processor template. The processor template that makes up the numerical accelerator is a collection of floating point units (FPUs) connected to memories through an interconnect structure. The task of the compiler is to create schedules for the interconnect structure and the memories to perform the desired algorithm as fast as possible. The compiler does this by representing the algorithm as a data flow graph (DFG) and scheduling the graph using depth-first list scheduling. The compiler then assigns memory addresses to the intermediate values of the DFG through a graph coloring heuristic to avoid structural hazards. The final result of the compiler is a set of instructions that can be loaded onto an instance of the processor template to create the desired numerical accelerator.

This work also covers how algorithms are inputted into the compiler. One of the methods of algorithm input uses C++ function templates that express the numerical algorithm on a template data type. That type can be replaced with the `graphMaker` class to create a DFG, or it can be replaced with `float`, `double`, or `int` to check the correctness and the performance of the algorithm with different precisions. This enables algorithm designers to create a single version of the algorithm for both simulation and compilation.

Multiple algorithms were compiled with this custom compiler to show how schedules change with the size of the processor, to show how the distribution of units within processors can be specialized for algorithms, and to show how algorithms can be optimized by the compiler to rival hand optimization of algorithms.

Thesis Supervisor: Vladimir Stojanović
Title: Associate Professor

Acknowledgments

I would first like to thank my thesis advisor, Vladimir Stojanović, for his mentorship and guidance in these first two years at MIT. I am also grateful for his insight and direction throughout the project, and his availability to discuss the details of the project.

I would vary much like to thank my partner in this project, Ranko Sredojević, for the support through countless hours of meetings. His work on the processor side made my results from this compiler into something real instead of just stats and plots. When I first started at MIT, he caught me up to speed on the project in a matter of weeks, and thanks to him, I was able to contribute to the project early on.

I would also like to thank everyone else that was a part of the Integrated Systems Group at MIT during the years I was there including Omid Abari, Mike Georgas, Sumit Dutta, Fabian Lim, Amr Suleiman, Chen Sun, Stevan Urosevic, Yan Li, Zhipeng Li, Jonathan Leu, Ben Moss, Yu-Hsin Chen, and Hossein Fariborzi. I really enjoyed talking to them in lab, eating lunch with them, and going to ISSCC together. Also, the tales of tape-outs from the photonics project made me appreciate the ML605 we were using for verification.

I am very grateful of all my friends and housemasters at Ashdown House. Playing board games with my neighbors and friends provided a much needed distraction at times. The housemasters Ann, Terry, Katie, and Yuri all supported my friends and me through our time on the house's executive committee. They were all willing to hear me talk about my troubles and successes in research and grad school.

Finally, I would like to thank the people in my personal life that provided the most encouragement and were always telling me that I could make it through the hard classes and the busy periods in research. Thanks to my girlfriend Leslie Maldonado for all the support, distractions, and laughs. Thanks to my parents and the rest of my family for everything they have done for me growing up, in and out of school. The daily Lucas updates from Bob and Cynthia were always a pleasant surprise when I was in lab. The visits from my dad when he was on business trips in Boston were

always fun. Last but not least, I would like to thank my mom for the extra motivation to graduate in the spring by buying non-refundable airfare for graduation months in advance.

Contents

1	Introduction	17
2	System Overview	19
2.1	Processor Template	19
2.2	Schedules	22
2.3	Data Flow Graph	23
2.4	Algorithm Performance Limits	23
2.5	Ensuring Fast Performance of Algorithms on Fixed Processors	25
2.6	Summary	26
3	Compiler Flow	29
3.1	Motivation for Compiler	29
3.2	Processor Parameters	30
3.3	Algorithm	31
3.4	Data Flow Graph	31
3.4.1	Depth Priority	33
3.5	Scheduling	35
3.6	Memory Assignment	37
3.7	Instruction Generation	39
3.8	Compiler Optimizations	39
3.8.1	Collapsing Nodes	39
3.8.2	Expanding Super-Nodes	40
3.8.3	Constant Folding	41

3.8.4	Inverse Optimization	44
3.8.5	Operation Duplication	45
3.8.6	Compiler Optimization Settings	46
3.9	Summary	50
4	Compiler Input Language	51
4.1	Text Based Input	51
4.2	C++ Template Functions	52
4.2.1	graphMaker Class	53
4.2.2	Numerical Data Types	55
4.2.3	Helper Class Templates	55
4.3	Front-End Optimizations	56
4.3.1	Constant Folding	56
4.3.2	Constant Checking	57
4.3.3	constTracker	58
4.4	Summary	60
5	Compiler performance	61
5.1	Algorithms	62
5.1.1	LU Decomposition	62
5.1.2	LDL^T Decomposition	63
5.1.3	Cholesky Decomposition	64
5.1.4	Discrete Fourier Transform	65
5.1.5	Fast Fourier Transform	65
5.2	Compiler Driven Algorithm Exploration	66
5.3	Three Sizes of Processors	67
5.3.1	Scheduling	67
5.3.2	Memory Assignment	69
5.4	Redistributing Units	73
5.5	Optimizations	76
5.6	Summary	78

6 Conclusion	81
A Matrix class	83
A.1 Motivation	83
A.2 Overview of Matrix Class	83
B Input Algorithms	87
C graphMaker.hpp	95

List of Figures

2-1	System design flow	20
2-2	Processor block diagram	21
2-3	Processor block diagram	22
2-4	A DFG with a critical path shown in red.	24
2-5	Results of scheduler improvements on generated schedules	26
2-6	Results of DFG improvements on bounds	27
3-1	Compiler flow showing the four main stages and the interactions between the compiler and the processor configuration.	30
3-2	DFG example	32
3-3	More general DFG showing explicit memory access	34
3-4	DFG with collapsed nodes	40
3-5	Super-node expansion: optimal and suboptimal	43
3-6	Super-node representations of $x = (2b) \div (4ac)$ for constant folding optimization	44
3-7	Super-node representations of $x = (a + b) - (c - (d - a))$ for inverse operation optimization	45
3-8	Tree rebalancing for $\text{tmp}=\text{a}+\text{b}+\text{c}$; $\text{x}=\text{tmp}+\text{d}$; $\text{y}=\text{tmp}+\text{e}$; without duplicating nodes	47
3-9	Tree rebalancing for $\text{tmp}=\text{a}+\text{b}+\text{c}$; $\text{x}=\text{tmp}+\text{d}$; $\text{y}=\text{tmp}+\text{e}$; with duplicating nodes	48
3-10	Compiler optimization levels	49
5-1	Latency bound results for DFT and FFT	68

5-2	Throughput bound results for DFT and FFT	68
5-3	Scheduling results for LDL^T decomposition	69
5-4	Scheduling results for LU decomposition	70
5-5	Scheduling results for Cholesky decomposition	70
5-6	Memory assignment results for LDL^T decomposition	71
5-7	Memory assignment results for LU decomposition	72
5-8	Memory assignment results for Cholesky decomposition	73
5-9	Scheduling results for improved LDL^T processors	74
5-10	Memory assignment results for improved LDL^T processors	75
5-11	Scheduling improvement for the medium LDL^T processor	75
5-12	Optimization improvements for an unoptimized LDL^T algorithm . . .	76
5-13	Optimization improvements for a hand optimized LDL^T algorithm . .	77
5-14	Optimization improvements for LDL^T (both hand and compiler opti- mizations)	78

List of Tables

2.1	Operational units available for the processor template to use	21
4.1	Available operators and functions in simple text file input language .	52
4.2	Functions and operators overloaded for graphMaker to construct the DFG	54
4.3	Improvements made by constant folding in graphMaker with only one constant	57
4.4	Functions available for graphMaker to check constant nodes	58
5.1	Processor parameters	67
5.2	Processor parameters with units redistributed for LDL^T	74
A.1	Operators defined on matrix class	84
A.2	Row, column, and block access operations defined for the matrix class	85
A.3	Special functions	85

List of Algorithms

3.1	Operation scheduling	36
3.2	Memory assignment	38
3.3	Super-node expansion	42
5.1	LU decomposition	62
5.2	LDL^T decomposition without hand optimization	63
5.3	LDL^T decomposition with hand optimization	64
5.4	Cholesky decomposition	64
5.5	FFT	66

Chapter 1

Introduction

There is much effort put into researching ways of accelerating numerical computations for embedded systems. General purpose processors are high-performance, but they have high power consumption. Embedded processors are on the other side of the spectrum with low power, but inferior performance. When both high performance and low power are system requirements, custom numerical accelerators become attractive [5, 8, 12, 13, 14].

Custom hardware numerical accelerators are usually hand crafted hardware designs that take a significant amount of design effort. Many custom hardware numerical accelerators rely on hand-crafted designs that are a result of thorough examination of the algorithm structure [8, 12, 14]. Others rely on partitioning the algorithm between an embedded processor and a custom hardware block that accelerates a portion of the computation. These designs are usually implemented for a specific algorithm, and any changes to the implemented algorithm require extensive effort to propagate the changes to the hardware. This reduces the amount of possible design reuse.

Our approach is to pair instances of a processor template with instructions generated by a statically scheduling compiler to generate numerical accelerators from algorithms written in a user-friendly form. The processor is templated to allow for specialization of the processor to the type and size of the algorithm. We have shown in previous work that the ideal processor configuration for an algorithm changes with the problem size [15].

In this thesis, we will present the statically scheduling compiler that enables numerical accelerator generation in coordination with the processor template. Chapter 2 presents an overview of the entire system including algorithm, compiler, and processor. This chapter gives context for the compiler and introduces a way to evaluate the performance of the compiler by comparing the results to theoretical bounds on algorithm execution time. Chapter 3 covers the compilation flow and the algorithms used within the compiler. Chapter 4 introduces the two input methods for the compiler and shows how a single algorithm formulation can be used for testing and compilation. Chapter 5 presents some results from compilation runs and shows how processor size, processor configuration, and compiler optimization all affect the generated schedules.

Chapter 2

System Overview

Our approach to making numerical accelerators involves selecting an algorithm and hardware in parallel. In order to be able to do this well, the design space for the algorithm and the design space for the hardware need to be reduced. In this work we are assuming algorithms can be represented as a set of algebraic expressions and assignments. The hardware is limited to processors that are generated by a specified processor template.

Our flow of making numerical accelerators can be seen in Figure 2-1. Algorithm development starts off independently with some optimizations, but it then meets hardware design inside the compiler. The compiler generates a schedule for mapping the algorithm to the specific processor. If the schedule does not meet system design constraints, or further optimization is desired, then new processor parameters can be chosen, and compilation can be run again. This process continues until a final processor is chosen and the schedule is loaded onto the processor to create an accelerator.

2.1 Processor Template

The processor template is designed to be able to exploit parallelism in the target algorithm, but the amount of parallelism is not known until the algorithm has been chosen, so the specifics of an ideal processor cannot be decided upon until the intended

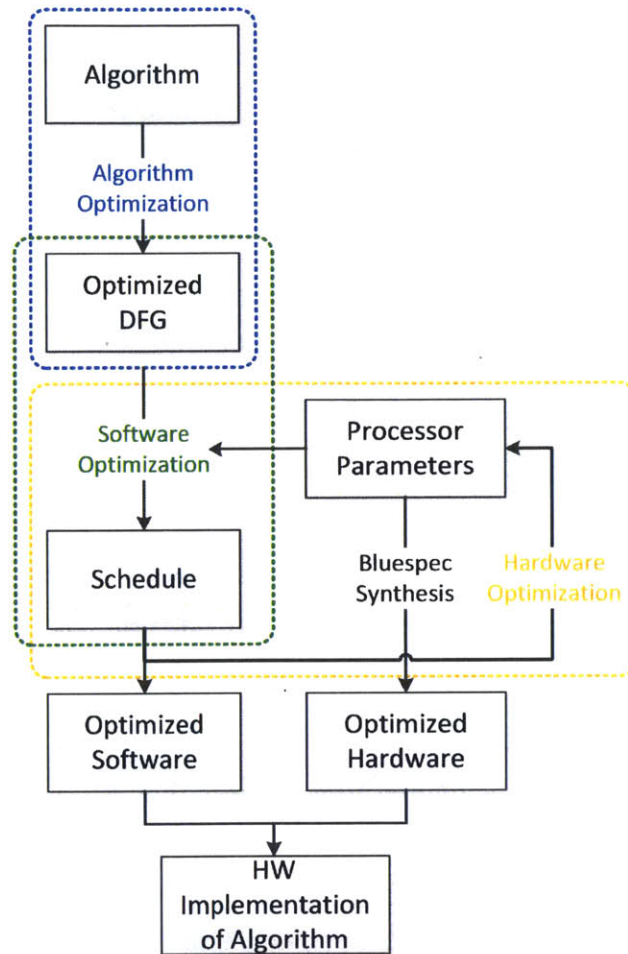


Figure 2-1: System design flow

load is known. The processor is kept generic by extensive parameterization of the hardware template through modification of the amount and types of units and the pipeline depth of units. The processor is made up of multiple operational units connected to an array of memories through muxes. An overview of an example processor generated by the processor template can be seen in Figure 2-2.

The processor is able to exploit parallelism easily due to the parallel array of fully pipelined operational units. Due to the memory interconnect structure, these operational units can all be fed data in the same cycle to obtain full utilization of these units. Each operational unit has a configurable pipeline depth so they can be designed for a specific target frequency of the processor.

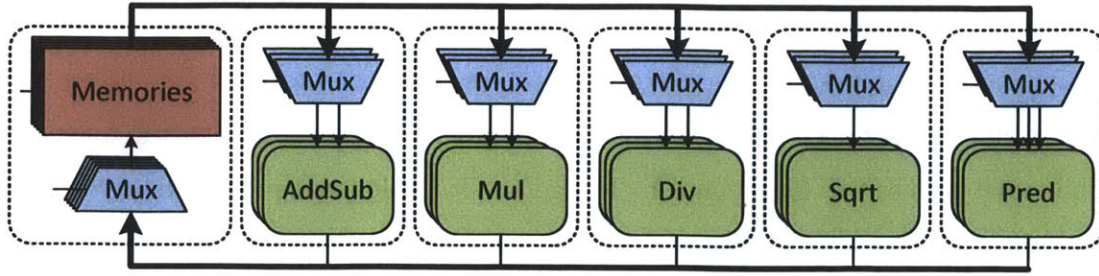


Figure 2-2: Processor block diagram

Currently all of the implemented operational units are floating point units (FPUs), but due to the parameterized nature of the processor, these units could be replaced by integer operational units and operate on integers instead. The implemented units can be seen in Table 2.1 along with the number of inputs for the unit and the range of pipeline depths [1]. The *Pred* operational unit has 3 inputs for the predication mode. One input is used as a boolean value and selects which of the other two inputs is passed to the output to be written to memory. All other operations in the *Pred* unit only use two of the inputs.

Unit Name	Inputs	Pipeline Depths	Operations
AddSub	2	1-11	addition and subtraction
Mul	2	1-6	multiplication
Div	2	3-28	division
Sqrt	1	1-28	square root
Pred	3	0-2	min, max, comparison, predication

Table 2.1: Operational units available for the processor template to use

All of the input, output, and intermediate values used in the processors are stored in one of the memories in the processor. The memories are dual ported memories so they are able to read one value and write one value in a given clock cycle. The number of memories is a configurable parameter in the processor template to allow the designer to pick the number of memories depending on the number of operational units in use, the data access patterns, and total amount of memory needed.

Each memory can provide data to each input of each operational unit through multiple muxes that act as a crossbar. In the other direction, data from each operational unit can be passed to each memory for writing through more muxes that act as

another crossbar. The size of these crossbars depends on the number of operational units and the number of memories in the processor. As the processor gets larger, the interconnect delay becomes significant, and in order to sustain high clock frequencies, the crossbars need to be pipelined. To support variable sized processors at variable clock frequencies, the crossbars have a configurable pipeline depth in the processor template.

2.2 Schedules

Each operational unit and each memory has a local controller that drives control signals for the unit and the select lines for the mux or muxes feeding the unit. These controllers have their own instruction memory where the schedules for the current algorithm are stored. A controller for an operational unit can be seen in Figure 2-3

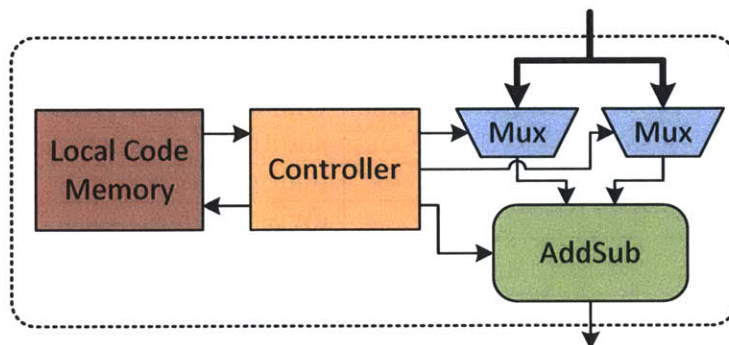


Figure 2-3: Processor block diagram

Instructions in these schedules are performed sequentially without branches or loops. This eliminates required communication between the operational units, and it significantly simplifies the control logic and the scheduling problem.

The compiler presented in this work is designed to generate these schedules. The compiler's job of scheduling an algorithm for a specific processor involves assigning times and operational units to each elementary operation (add, subtract, multiply, etc.) in the algorithm. These assignments need to be free of hardware conflicts and data conflicts. A hardware conflict appears in its simplest form when two operations

are assigned the same hardware at the same time. A more complicated hardware conflict occurs when two operation schedule assignments involve reading a value from the same memory. Since the memory only has one read port, only one of the operations will be able to get a valid value.

Data conflicts occur in a schedule when an operation is scheduled for a time that is earlier than the times the operands become available in memory. This can happen when operation A depends on the results of operation B. If operation B is scheduled to finish on cycle 10, but A is scheduled to start on cycle 5, the result from B will not be ready yet and operation A will run with invalid data.

2.3 Data Flow Graph

To make the compiler's job of scheduling easier, the algorithms are converted to an intermediate representation called a data flow graph (DFG) to show hardware requirements and data dependencies between operations. The DFG is a directed acyclic graph (DAG) with nodes representing elementary operations and data and directed edges that represent the flow of data between operations. DFGs are commonly used in high-level synthesis [6], and are very similar to DAGs of basic blocks in standard compilers [4].

2.4 Algorithm Performance Limits

There are two main performance bounds that limit the performance when running an algorithm on the processor template: the latency bound and the throughput bound. These bounds are the result of the combination of the DFG's structure with the limitations of the processor configuration.

The DFG has two main structural components that contribute to bounds: its size and its depth. The size is determined by the number of nodes that map to each type of operational unit. The depth is determined by the length of the longest path of data dependencies from any source to any sink when accounting for the computation time

required for each operation along the way. This longest path is also called the critical path. When there are multiple longest paths within a DFG, they are all considered critical paths. Figure 2-4 shows a critical path of a DFG in red.

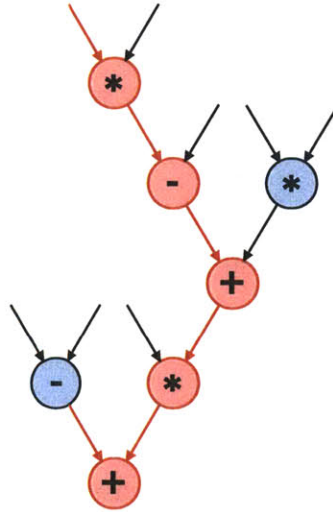


Figure 2-4: A DFG with a critical path shown in red.

The processor is limited by two structural factors: the number of units of each type and the latency of the different modules in the processor. If there are only N processors of a given type, then no more than N operations that need that processor can be scheduled in a single clock cycle. The latencies within the processor determine the amount of computation time required to calculate a result from the time the operands are read in memory.

The latency bound is the result of considering the depth of the DFG and the processor's latency for each operation. If the processor's latencies are added up for each node in the critical path of the DFG, then a lower bound for execution time is obtained since each operation in the critical path needs to wait for the previous operation to finish before it can start.

The throughput bound is the result of considering the size of the DFG and the limited number of units of each type in the processor. The simplest way to calculate a throughput bound is to divide the number of nodes in the DFG by the number of operational units in the processor. This will produce the minimum number of clock

cycles required to issue that many instructions in the given processor, so it will take at least that long to finish running all the instructions. This bound is valid, but it is not as tight as it could be since it does not take into account the different types of nodes in the DFG and the different types of operational units in the processor.

By restricting this bound to consider that nodes can only be executed by certain operational unit types, an operational unit specific throughput bound can be calculated. Consider an algorithm that has 100 additions and 100 multiplications. When this algorithm is run on a processor with two *AddSubs*, five *Muls*, and three *Divs* the previous calculation of the throughput bound would result in 20 cycles because there are 200 operations performed on a processor with 10 operational units. When only considering the 100 additions that have to be done on the two *AddSub* units, a different throughput bound can be calculated that is specific to *AddSubs*. This *AddSub* throughput bound is 50 cycles, which is higher than the other calculation of the throughput bound.

When each operational unit type is considered separately, a more accurate throughput bound is calculated by taking the maximum of all the operational unit specific throughput bounds. Throughout this work, the throughput bound is calculated using the maximum of each operational unit specific throughput bound.

2.5 Ensuring Fast Performance of Algorithms on Fixed Processors

When considering the execution time of an algorithm on a specific processor instance, there are two main factors that contribute to the performance: the scheduler's efficiency and the structure of the DFG. The scheduler assigns times for the execution of each operation in the DFG. An ideal scheduler would assign times that result in the fastest schedule possible for the DFG. The throughput and latency bounds are ways to approximate the fastest schedule possible, so an ideal processor would attempt to get the performance as close to those bounds as possible. Figure 2-5 shows what an

improvement in scheduler would look like for an algorithm's execution time. This corresponds to software optimization in Figure 2-1.

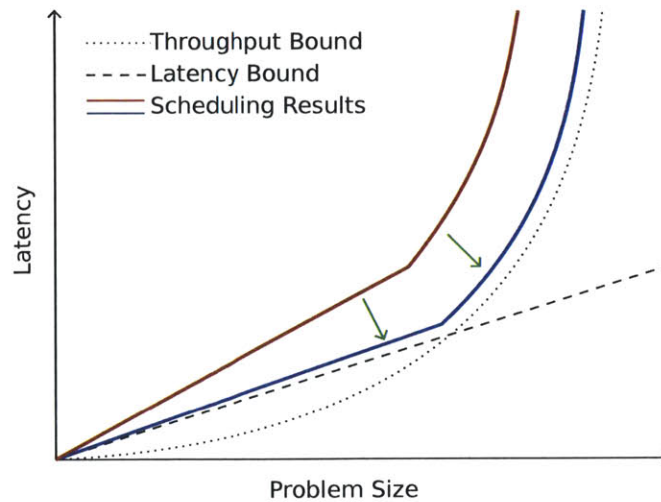


Figure 2-5: Results of scheduler improvements on generated schedules

When the processor is fixed, only changes in the DFG structure affect the throughput and latency bounds. If the scheduler is already meeting theoretical bounds, the only room for improvement is in changing the structure of the DFG so the bounds for execution time decrease. Figure 2-6 shows improvements in both latency and throughput. The scheduler's performance is assumed constant for these plots, but since the bounds are lowered, the scheduler is producing shorter schedules. This corresponds to algorithm optimization in Figure 2-1.

2.6 Summary

In this chapter we introduced the system of creating numerical accelerators consisting of the processor template and the statically scheduling compiler. The processor template is a collection of FPU cores, memories, and controllers all connected together with an interconnect structure. The controllers were shown to introduce the control signals that determine the functionality of the processor. The compiler introduced in the next section generates code that sets these control signals at each clock cycle in the processor.

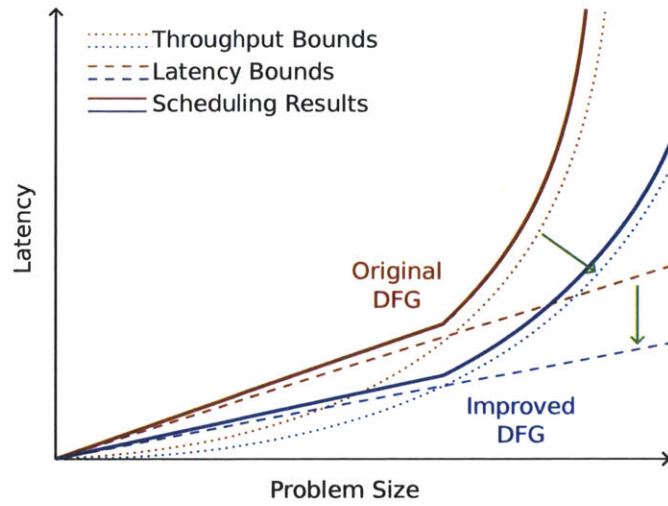


Figure 2-6: Results of DFG improvements on bounds

This chapter also included an introduction to the algorithm side of the generated numerical accelerators. DFGs were introduced to explain the computational model used for algorithms within the compiler and to introduce the two performance bounds. The two performance bounds, latency and throughput, set limits to the compiler's performance, but they also motivate the compiler optimizations introduced in the next chapter.

Chapter 3

Compiler Flow

The design flow in Figure 2-1 requires a statically scheduling compiler to produce the instructions for the controllers in Figure 2-3 and choose the optimal processor template configuration. The compiler takes two inputs: processor parameters to describe the target hardware and an algorithm to describe the target software. This compiler is divided into four parts: graph generation, scheduling, memory assignment, and code generation. The flow of the compiler can be seen in Figure 3-1.

3.1 Motivation for Compiler

A compiler is needed in this system to take algorithms and processor parameters from the user and produce lists of instructions for each unit within the specified processor. Since the compiler is the bridge between the algorithm and the processor, it is an ideal place to explore the effects of processor parameters on algorithm execution time. The compiler needs to be able to look at a DFG and determine quickly how fast it will run on a given processor so it can look at many processor configurations in a short amount of time. Also, the compiler needs to be able to output information about the DFG that could be useful to the designer when choosing processor configurations. Information like the size of the DFG, the number of each type of operation, and the length of the critical path is very useful for designers. With this functionality, the compiler becomes more of a tool than a necessary block in a system.

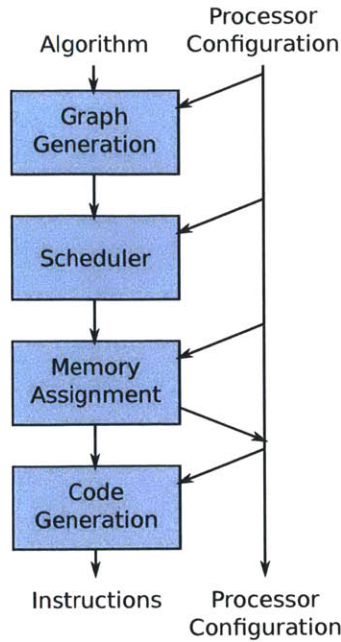


Figure 3-1: Compiler flow showing the four main stages and the interactions between the compiler and the processor configuration.

3.2 Processor Parameters

The parameters of the target processor need to be specified in order for the compilation results to be compatible with the target processor. The processor parameters include all of the parameters required to instantiate a processor from the processor template. These parameters include the number and type of operational units, the number and size of memories, the latencies of the operational units, and the latencies of the crossbars.

This information about the processor is used throughout the compilation process. The graph generation stage uses the latencies from the processor parameters to calculate depths in the graph for the scheduler to use. The scheduler uses the latencies to ensure cycle by cycle accuracy of the schedule. It also uses the number of units to make sure all available hardware is being utilized and no additional hardware is being assumed. The memory assignment module needs to know how many memories to use for variable location assignment. If the memory assignment cannot fit the variables into the number of memories specified by the processor parameters, then the memory

assignment module modifies the amount of memories in the parameters to make the design fit, and a warning message is sent to the user of the compiler. At the end of the compilation process, the processor parameters are used for code generation. Code generation creates the instruction data that will be loaded into the processor, so it needs to know all the details about the processor.

The interactions between the processor parameters and the compilation process are shown in Figure 3-1.

3.3 Algorithm

The algorithm is inputted into the compiler in one of two ways. The algorithm is either written up in a simple text file with syntax that directly maps to the operations in the processor, or it is written up using a C++ function template where higher level functions can be used as long as they are composed of lower level operations that can be mapped to the processor. The two algorithm input methods are described in more detail in Chapter 4.

3.4 Data Flow Graph

As the algorithm is inputted into the compiler, a DFG is generated. This DFG represents the structure of the algorithm through nodes that represent operations and data storage (hardware usage) and directed edges that represent data flow. The sources of the DFG represent constants and input variables in the algorithm. The sinks of the DFG represent results of the algorithm, but not all desired results are sinks, some are intermediate nodes. An example DFG can be seen in Figure 3-2.

Since the directed edges in the DFGs represent the order in which operations are done in the algorithm, there are no loops allowed in these DFGs. If there were a loop in a DFG it would imply that an operation needs to be computed in order to compute itself, which is not possible. A DFG describing a looping iterative method such as Newton's method for finding roots of functions would have the subgraph representing

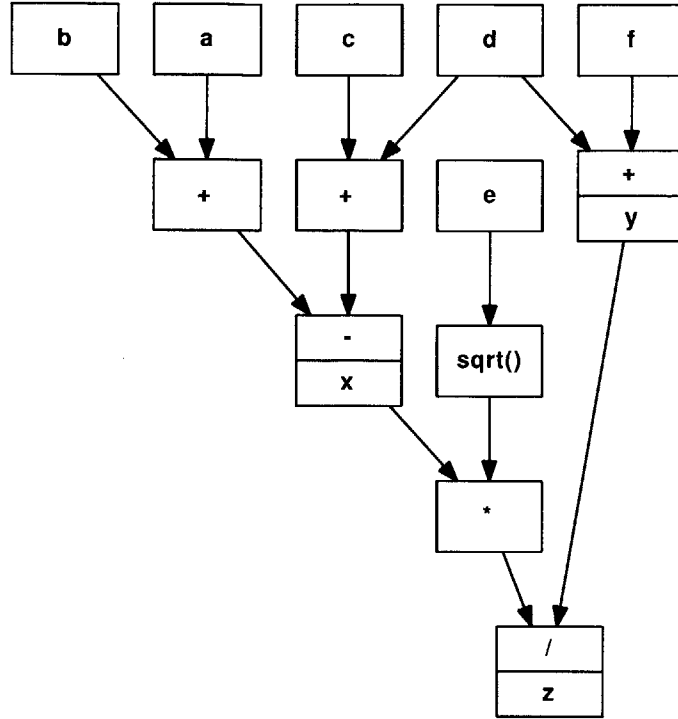


Figure 3-2: DFG example

an iteration repeated multiple times and placed in a sequence in the DFG.

Even though there are no loops in DFGs, their structure can still vary greatly. The example DFG in Figure 3-2 has many inputs and one output, but an operation like a complex Fourier transform has $2N$ inputs and $2N$ outputs making its top as wide as its bottom. Operations such as vector-vector addition produce a forest of many small trees since the individual operations do not depend on each other. The DFG for computing a power of a number using successive squaring would have one input and one output, but there would be a chain of multiplies between the two.

During scheduling, the nodes of the DFG are assigned a time to execute and hardware to execute the node. After all of the nodes in Figure 3-2 are assigned times to execute, there is still a question of what memory is going to be written to and read from for each operation because that information is not shown in the graph.

Memory accesses can be added to the DFG to make it more general as shown in Figure 3-3. This DFG has nodes for floating point operations, nodes for memory writes, and nodes for memory reads. The memory reads and memory writes are

constrained such that they have to be from the same memory block. This is a more detailed view of what the processor will be doing to compute an algorithm, but it adds complexity and additional constraints that complicate the scheduling process.

To simplify scheduling, we use DFGs that only show nodes for numerical operations and do not treat memories as a constrained resource at scheduling time. The reasons this assumption can be made and the cost of the assumption are discussed in more depth in section 3.6.

3.4.1 Depth Priority

At this stage, the DFG contains information about all the required operations, but it needs information about which nodes are more important to schedule if efficient schedules are going to be generated. A priority for each node can be obtained by looking at required execution time after each node is scheduled.

At each node in the DFG that is not a sink, there is at least one path from that node leading to a sink of the graph. Each node along that path will require a calculation to be performed that depends, either directly or indirectly, on the given node. Due to the dependencies, these nodes will have to be scheduled after the initial node has completed execution. These nodes will also have to be scheduled after each previous node on the path has completed execution as well. This path gives a lower bound for the amount of time required to finish executing the algorithm after the initial node has been scheduled for execution. This bound is obtained by adding up the execution time for the initial node and every node on that path. If there are multiple paths from the initial node to the sinks, then each path can be examined to calculate a better lower bound. If the scheduled time is known for the initial operation, then a lower bound for the completion of the entire algorithm can be computed by adding the scheduled time to the lower bound.

The lower bound for completion time after scheduling can be used for prioritizing the scheduling process. If there are multiple nodes that can be scheduled in the same time slot, scheduling the node with the largest amount of computation required before completing the algorithm is preferred. Scheduling that node later will increase the

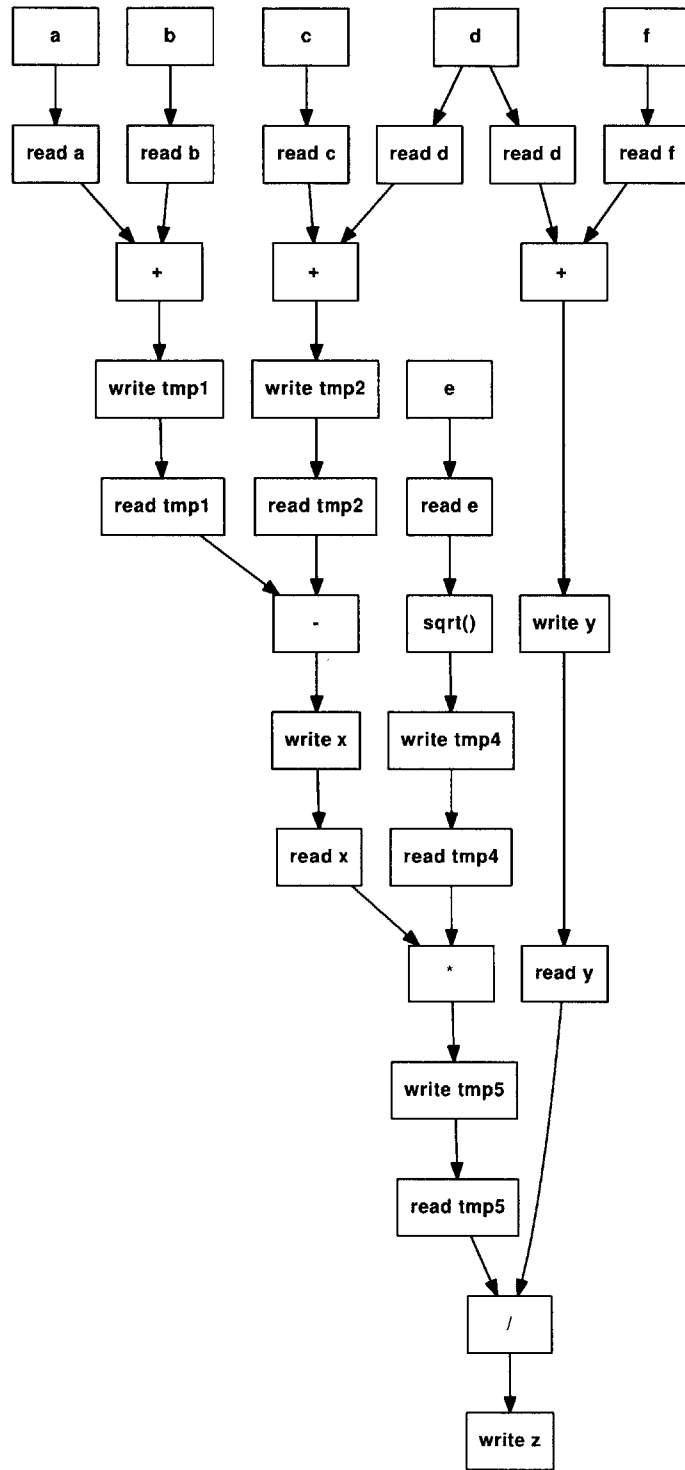


Figure 3-3: More general DFG showing explicit memory access

lower bound for completion time of the entire algorithm.

The node prioritizer starts at each sink and calculates lower bounds for each node. After the node prioritizer is done, the node with the highest lower bound for additional execution time gives the latency bound. This lower bound is tight when there are sufficiently many FPU units. The path that causes this lower bound on total execution time is called the critical path.

This priority is very similar to depth in a tree, except the difference in priority between two nodes depends on computation time, not the number of edges between them as is the case with depth.

In the example program shown in Figure 3-2, after the operation $a + b$ is finished, there is still a subtraction, a multiplication, and a division along the path from $a + b$ to the sink z . The priority for the operation $a + b$ is the time it takes to do an addition, a subtraction, a multiplication, and a division. The priority for $d + f$ is only the time required to perform an addition and a division, so $a + b$ has a higher priority than $f + g$.

3.5 Scheduling

Scheduling is done using a list scheduling algorithm sequentially in time starting with the first clock cycle. The compiler looks at all of the operations that depend only on variables that will be valid in memory at the current clock cycle. It then chooses the operations with the highest priorities and assigns them to FPUs for the current clock cycle. The results are then marked to be ready at a time in the future (when the specified operation is completed and the results are written back). The compiler then looks at the next clock cycle, and the process continues. Since the priority function is closely related to depth, this process is very similar to depth first scheduling. The full scheduling algorithm can be seen in Algorithm 3.1

Algorithm 3.1 Operation scheduling

```
node list  $L \leftarrow [ ]$ 
for all nodes  $n$  do
    Calculate  $\text{depth}(n)$ 
    if node  $n$  is a source then
        Insert  $n$  into  $L$  with descending depth
    end if
end for
 $t \leftarrow 0$ 
while node list  $L$  not empty do
    for all operational units  $u$  do
         $\text{sched}_n \leftarrow \text{NULL}$ 
        for all nodes  $n$  in  $L$  do
            if  $n$  can be scheduled on unit  $u$  and  $n$ 's operands are ready at time  $t$ 
then
                 $\text{sched}_n \leftarrow n$ 
                Break
            end if
        end for
        if  $\text{sched}_n \neq \text{NULL}$  then
            Schedule node  $\text{sched}_n$  on unit  $u$  at time  $t$ 
            Insert dependents of  $\text{sched}_n$  with scheduled operands into  $L$  by depth
        end if
    end for
     $t \leftarrow t + 1$ 
end while
```

3.6 Memory Assignment

While scheduling produces the times for each operation to execute, the memory assignment in the next step produces the read and write addresses for each operation. Each node in the DFG needs to be assigned a memory and an address within that memory so there are no conflicts within the processor. Since the processor memories have one read port and one write port, this means that the memories can only be written to by one FPU at a time, and only one variable can be read from a memory at a time (even though multiple FPUs may be reading the same variable in the same clock cycle).

To make sure the memory ports are not overused in a single cycle, the compiler generates a graph showing the dependencies between all of the variables. The graph has an edge between two variables if they are both read in the same cycle or if they are both written in the same cycle. If all the variables connected by edges are always in different memories, then there will never be a resource conflict between instructions. The task of assigning each node in the graph a different memory such that no two edges connect nodes with the same memory is the same as finding an M coloring of the graph where M is the number of memories. Once a valid coloring is found using a heuristic, the memory assignments are shuffled while satisfying the constraints to even out the number of variables in each memory.

After the memory assignment, each variable needs to have an address within the memory assigned to it. If the program does not have too many intermediate results, unique addresses can be assigned to each variable in a memory. If space needs to be saved, the variables are tracked in the schedule to see when they become valid, and how long they remain in memory. The compiler will then share addresses between variables that do not need to be stored in memory at the same time. The full algorithm can be seen in Algorithm 3.2.

Algorithm 3.2 Memory assignment

```
nmems  $\leftarrow$  0
L is list of all nodes
for all nodes n do
    n.deps  $\leftarrow$  0
end for
for all pairs of nodes n, m do
    if n and m cannot use the same memory then
        n.deps  $\leftarrow$  n.deps + 1
        m.deps  $\leftarrow$  n.deps + 1
    end if
end for
Sort L by decreasing n.deps
for all nodes n in L do
    i  $\leftarrow$  1
    while mem[i] has memory conflict with n do
        i  $\leftarrow$  i + 1
    end while
    n.memory  $\leftarrow$  i
    nmems  $\leftarrow$  max(nmems, i)
end for
Sort L by increasing n.deps
for all nodes n in L do
    Assign n to the memory with the least nodes assigned to it
end for
```

3.7 Instruction Generation

The last step of the compiler is to take all the scheduling information and memory assignments and write them into a file that can be loaded into the processor's instruction memories. The processor has independent controllers for each crossbar and each memory, so the compiler runs through the schedule figuring out the settings for the crossbars and the address lines at each clock cycle, and it creates an instruction file for each unit. This information is all known at compile time because the programs do not have data dependent branches. Once the compiler has calculated all the control signals for each controller, there is an instruction file for every unit on the processor ready to be loaded.

3.8 Compiler Optimizations

The compiler has many optional optimizations built into it. These optimizations are applied after the DFG has been generated, but before scheduling starts. All of these optimizations rely on arithmetic laws of real numbers. Floating point arithmetic does not follow all the arithmetic laws of real numbers because of rounding errors, but just like floating point representations of real numbers, they are good approximations.

There are various optimization that can be applied to the DFG to either reduce the critical path or to reduce the number of operations. Both of these changes reduce the associated bound for algorithm performance, potentially improving the generated schedule for the DFG.

3.8.1 Collapsing Nodes

When the DFGs are generated from the input algorithm, the graph represents a specific way of combining inputs to get results, but since some of the operations used in the processor are commutative and associative, there are many different ways of representing the combination of inputs to get the same result. To reduce the dependency on representation, subtrees of commutative and associative operations

are collapsed into a single node called a super-node. This action is done during optimization primarily for rebalancing trees and shortening the critical path, but it is also useful to have this collapsed representation of commutative and associative subtrees when performing the other operations.

Super-nodes can be created for subtrees made up of $+$ and $-$ operations, subtrees of \times and \div , subtrees of \min , and subtrees of \max . Since $-$ and \div are not commutative or associative, the second operand in each of these cases is treated as if it is the inverse of the operand so the operations can be treated as $+$ and \times . The super-nodes keep track of each of the inverted inputs so when the node is expanded into individual operations, the subtree still produces the same result.

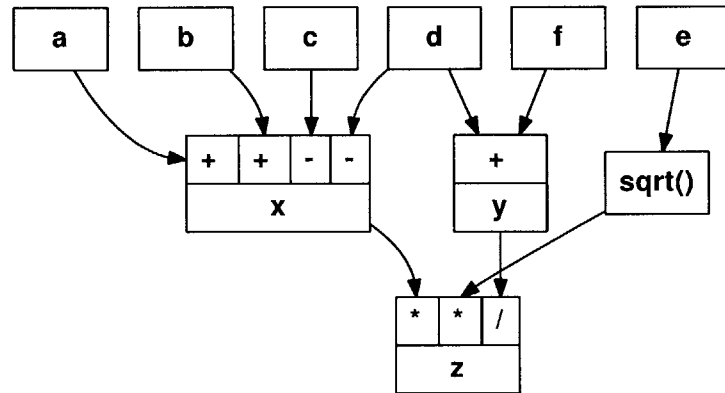


Figure 3-4: DFG with collapsed nodes

3.8.2 Expanding Super-Nodes

The scheduling process requires each node in the DFG to be assigned a depth. The depth is calculated using how long it takes to perform operations that occur along a path in the dependency graph. If a path in the DFG passes through a super-node, then it is unknown how many operations are on that path because super-nodes represent the combination of multiple nodes, and there are multiple ways to arrange them. Depending on how the super-node is expanded, the super-node can represent few or many operations along the path. Therefore DFGs with super-nodes cannot have an accurate depth calculation and cannot be scheduled without expanding super-nodes.

When expanding super-nodes, the goal is to expand the nodes in such a way that the critical path remains as short as possible. If the DFG is a single super-node of additions, then when expanding that node, the ideal configuration would be a balanced binary tree of additions because that has the shortest critical path of configurations.

It is not always ideal to have super-nodes expanded into balanced trees. Sometimes it is ideal for a super-node to be expanded into an unbalanced tree because one of the operands depends on many operations, and that path is more critical than the other paths entering the super-node. Figure 3-5 shows a pair of super-nodes expanded optimally and expanded into balanced trees.

The algorithm for expanding super-nodes is similar to ASAP scheduling with infinite resources [6]. The algorithm starts at the sources of the DFG and builds its way to the sinks. Along the way, when the algorithm gets to a super-node from two of its operands, a new operand node is created by the combination of the two inputs and it takes the inputs' place in the super-node. The full algorithm can be seen in Algorithm 3.3.

3.8.3 Constant Folding

Some nodes in the DFG represent constant values, and these known values can be used to reduce the number of operations in the DFG through constant folding [4]. If there are nodes in the DFG that depend only on constants, then the node can be evaluated and replaced with a constant before scheduling. Additionally, if there are nodes that are being operated on by the identity element of the operation, those can be simplified too.

This optimization can also be performed on super-nodes to reduce the number of constants a super-node is dependent on. If two inputs in a super-node are constants, they can be replaced with the constant equal to the combination of the two constants. For example, the equation

$$x = \frac{2b}{4ac} \tag{3.1}$$

Algorithm 3.3 Super-node expansion

```
 $L \leftarrow [ ]$   
for all sources  $n$  do  
  Set  $\text{depth}(n)$  to 0  
  Insert  $n$  into  $L$   
end for  
while  $L$  is not empty do  
  Pop node  $n$  from front of  $L$   
  Set  $\text{depth}(n)$  to be  $n.\text{latency} + \max \text{depth}(n.\text{operands})$   
  if  $n$  is a super-node then  
    Create node  $m$  from two operands of  $n$  with min depth  
    Set  $\text{depth}(m)$  to be  $m.\text{latency} + \max \text{depth}(m.\text{operands})$   
    Replace corresponding inputs of  $n$  with  $m$   
    if  $n$  has two operands with assigned depth then  
      Insert  $n$  in  $L$  by the second lowest operand depth  
    end if  
    for all nodes  $n'$  in  $L$  do  
      if  $n'$  can use  $m$  as an operand then  
        Replace corresponding inputs of  $n'$  with  $m$   
        if  $n'$  has two operands with assigned depth then  
          Insert  $n'$  in  $L$  by the second lowest operand depth  
        end if  
      end if  
    end for  
  else  
    Set  $\text{depth}(n)$  to be  $n.\text{latency} + \max \text{depth}(n.\text{operands})$   
    for all dependents  $m$  of  $n$  do  
      if  $m$  is a super-node then  
        if  $m$  has two operands with assigned depth then  
          Insert  $m$  in  $L$  by the second lowest operand depth  
        end if  
      else  
        if  $m$  has all operands with assigned depth then  
          Insert  $m$  in  $L$  by max operand depth  
        end if  
      end if  
    end for  
  end if  
end while
```

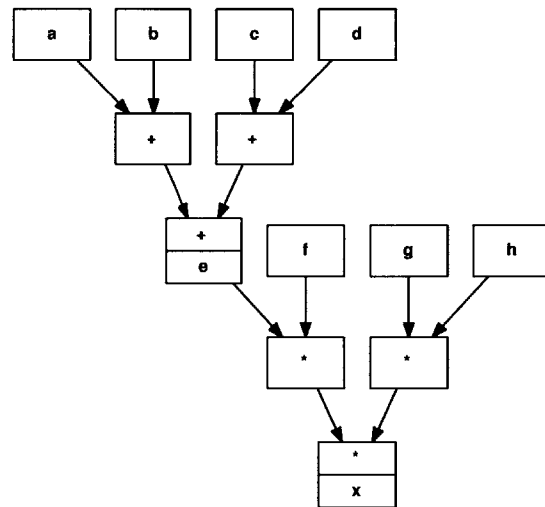
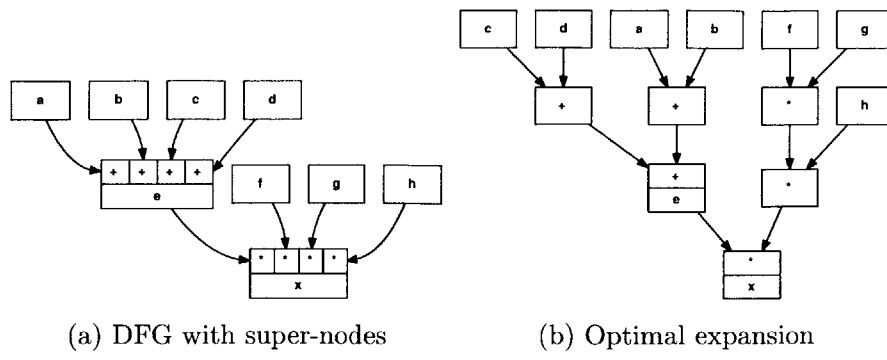
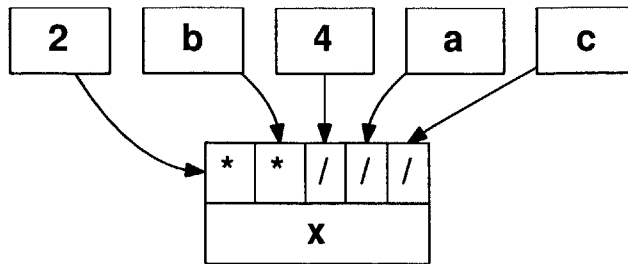


Figure 3-5: Super-node expansion: optimal and suboptimal

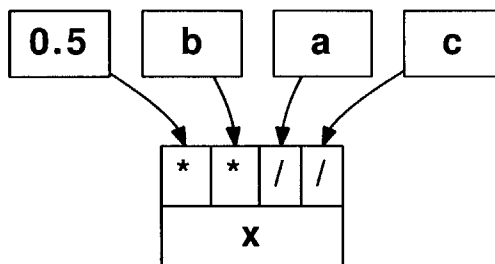
can be expressed as a super node as shown in Figure 3-6a. This super-node has a $\times 2$ and a $\div 4$ so the two of the can be replaced with a $\times 0.5$ resulting in the super node shown in Figure 3-6b. This new super-node represents the optimized equation

$$x = \frac{0.5b}{ac}. \tag{3.2}$$

Additionally, some select operations that depend on only one constant can be optimized as well using algebraic properties of 0 and 1 [4]. Since 0 is the identity element of addition, the expressions $a + 0$ and $a - 0$ can both be reduced to a . Similarly, since 1 is the identity element for multiplication, the expressions $b \times 1$ and



(a) Before optimization



(b) After optimization

Figure 3-6: Super-node representations of $x = (2b) \div (4ac)$ for constant folding optimization

$b \div 1$ can be reduced to b . Also, when 0 is multiplied by anything, the result is zero, so the expressions $c \times 0$ can be reduced to 0.

3.8.4 Inverse Optimization

Another algebraic optimization available in the compiler is inverse operation optimization. Inverse operation optimization is when an operation is able to be simplified because a value and its inverse appear in the same expression. The optimization is performed by removing the value and its inverse, and replacing them with the identity elementary for the operation and performing further constant folding. The simplest for of this is replacing $a - a$ and $a + (-a)$ with 0. For multiplication, this optimization replaces $a \cdot (a)^{-1}$ and $a \div a$ with 1.

This optimization can be performed on super-nodes to find less trivial optimizations. If two inputs in a super-node have the same data but opposite operation,

they can be replaced with the identity element for the operation. For example, the equation

$$x = (a + b) - (c - (d - a)) \quad (3.3)$$

can be expressed as a super node as shown in Figure 3-7a. This super-node has a $+a$ and a $-a$ so the two of them can be removed from the super-node and replaced with a $+0$. An obvious optimization allows for the removal of $+0$ to produce the super node in Figure 3-7b. This new super-node represents the equation

$$x = b - c + d. \quad (3.4)$$

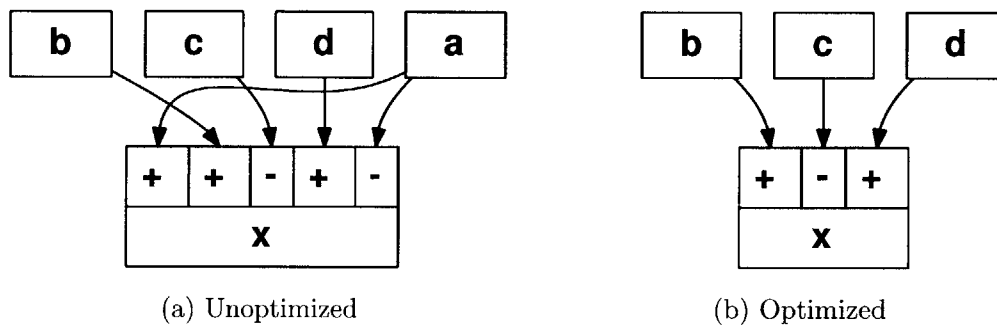


Figure 3-7: Super-node representations of $x = (a + b) - (c - (d - a))$ for inverse operation optimization

3.8.5 Operation Duplication

The source of the schedule improvements from the previous optimizations are clear from their actions. Constant folding and inverse operation optimization both reduce the number of operations in a DFG, potentially lowering the throughput bound. If those removed operations are on a critical path, then the latency bound could decrease also.

Even though it is not intuitive, sometimes it is advantageous to increase the number of operations in order to shorten the critical path and reduce the latency bound. This is the foundation for the operation duplication optimization; duplicating an intermediate result so trees can be rebalanced easier to shorten the critical path.

Consider the following algorithm:

```
tmp = a + b + c;  
x = tmp + d;  
y = tmp + e;
```

The DFG for this algorithm can be seen in Figure 3-8a. If all subtrees of the DFG made of associative operations are collapsed into super-nodes, the DFG is the one shown in Figure 3-8b. This algorithm cannot be collapsed into a single super-node because x and y both depend on tmp . Therefore, when the super-node is expanded, the resulting DFG as seen in Figure 3-8c is the same as the initial DFG.

If the super-node for tmp is duplicated into a second node $tmp2$, then x could depend on tmp and y could depend on $tmp2$ like in Figure 3-9a. At this stage, the DFG can be fully collapsed into two super-nodes, one for x and one for y . These super-nodes can be expanded more efficiently than the super-node in Figure 3-8b. When expanded, the super-nodes in Figure 3-9b become the DFG seen in Figure 3-9c.

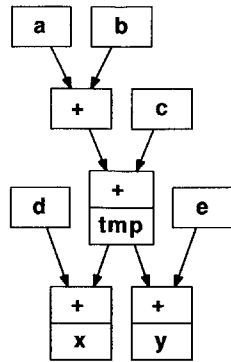
The original DFG contains 4 additions, and the critical path is a chain of 3 additions. The new DFG contains one more addition, but the critical path is shorter by one addition.

Often times, this optimization method is too aggressive, and it increases the number of operations by so much that the throughput bound becomes the active bound for scheduling. In these cases it is best to only do the other optimizations.

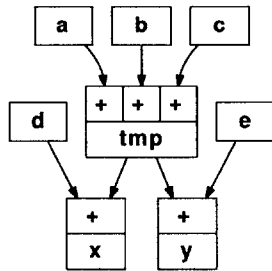
3.8.6 Compiler Optimization Settings

The custom compiler implements these settings and applies them depending on the optimization level which ranges from -00 to -03 similar to GCC [9]. -00 contains no optimizations, and the DFG is scheduled as-is. Each level above -00 adds optimizations to the compilation flow between DFG generation and scheduling.

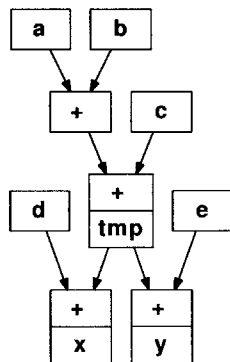
-01 keeps the structure of the DFG, but it performs constant folding and inverse operation optimizations. Constant folding and inverse operation optimizations are repeated one after the other until no gains are made in both. This repetition allows



(a) Initial DFG



(b) DFG with super-nodes

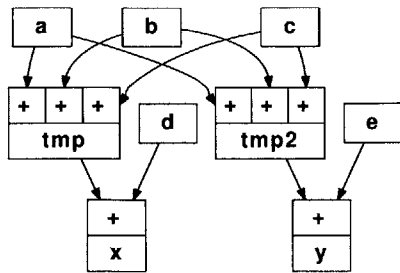


(c) Rebalanced DFG

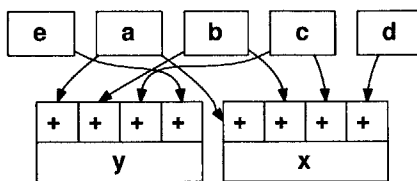
Figure 3-8: Tree rebalancing for $tmp=a+b+c$; $x=tmp+d$; $y=tmp+e$; without duplicating nodes

for expressions like

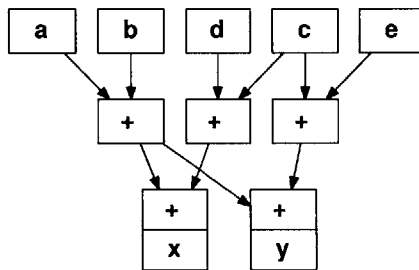
$$x = ((a \div (a + 0)) + 3) - b \tag{3.5}$$



(a) DFG with duplicated super-node



(b) Fully collapsed DFG



(c) Rebalanced DFG

Figure 3-9: Tree rebalancing for $\text{tmp}=\text{a}+\text{b}+\text{c}$; $\text{x}=\text{tmp}+\text{d}$; $\text{y}=\text{tmp}+\text{e}$; with duplicating nodes

to be optimized. Just one pass of the two optimizations results in

$$x = (1 + 3) - b. \tag{3.6}$$

A second pass is needed to fully optimize it to

$$x = 4 - b. \tag{3.7}$$

-O2 changes the structure of the DFG by collapsing it into super-nodes and then expanding it into balanced trees. While the DFG is collapsed into super-nodes, the -O1 optimizations are run. When the DFG is collapsed, these optimizations are more effective because the compiler can look across an entire associative subtree for optimizations.

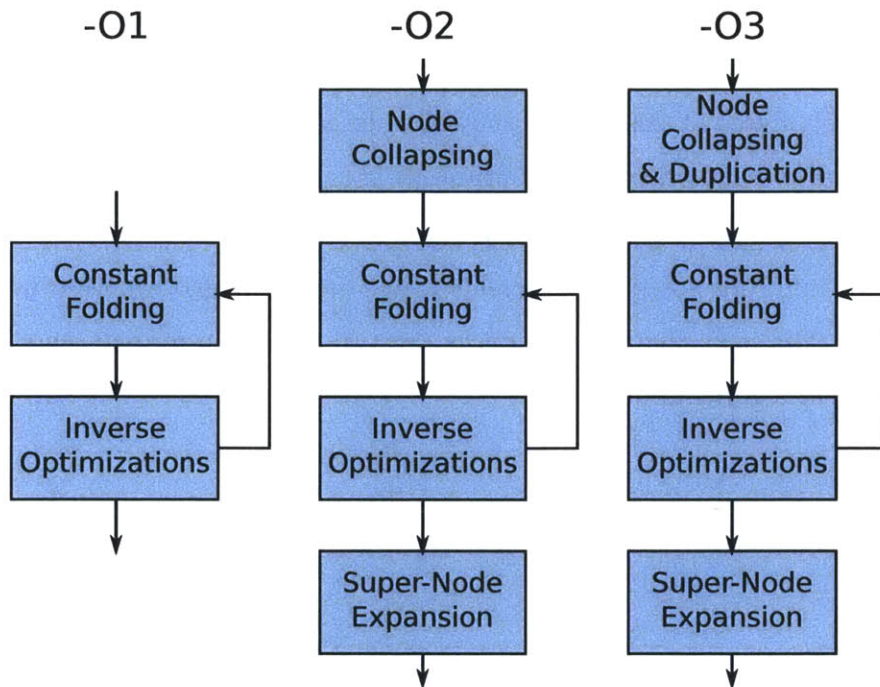


Figure 3-10: Compiler optimization levels

-O3 does the same as -O2, except when the compiler is collapsing the graph into super-nodes, it performs operation duplication to be able to do further collapsing. Section 3.8.5 shows an example of how this duplication works, and how it can be beneficial. Since -O3 increases the number of nodes in the DFG, it does not always produce a better schedule, but there are many cases when -O3 has gains in scheduling performance that surpass all other levels of optimization.

The three optimization levels are summarized in Figure 3-10.

3.9 Summary

In this chapter we presented the stages of the compiler including an overview of each optimization included in the compiler. The scheduler was introduced to show how DFGs are mapped to the target processor configuration. The memory assignment algorithm was covered to show how variables get their memory locations after scheduling.

Each optimization performed by the compiler was presented to show how the compiler can modify the DFG to get better scheduling results. The different optimization levels stated in this chapter showed when each optimization is enabled giving a comparison to standard compiler optimization levels.

The first stage of the compiler, graph generation, was briefly covered. The DFGs generated during this stage were presented, but the input algorithm format was not introduced. The next chapter introduces the two algorithm formats and how they enable efficient algorithm design.

Chapter 4

Compiler Input Language

Normal compilers for traditional programming languages take in a combination of text files and object files to produce a final program. These text files are written in the programming language's syntax.

Our compiler is targeting numerical algorithms so our compiler input methods need to be ways of expressing numerical algorithms. We have two ways of expressing algorithms to our compiler: a text file and a C++ function template.

4.1 Text Based Input

The first input method for the compiler is a plain text file with simple syntax. The file is parsed into the compiler and generated into a DFG by a lexer and parser generated by Flex and Bison [3, 7].

The file contains two parts: an optional header and a list of assignments separated by semicolons. The optional header includes parameters for the target processor. Any parameters omitted are assumed to be default values provided by the compiler. Listing 4.1 shows an example header for the medium processor in Table 5.2.

The body of the input file is a list of assignments separated by semicolons. For each function the processor's operational units can perform, there is a function or an operator in the language to express it in the text file. A full list of operators and functions can be seen in Table 4.1. An example for calculating the distance between

```

1 addsubs 6 latency 5;
2 muls 5 latency 3;
3 divs 1 latency 15;
4 xbar 1 1;

```

Listing 4.1: Example header for medium processor

Assignment	=
Binary operators	+, -, *, /
Relational operators	<, <=, >, >=
Equality operators	==, !=
Conditional function	cond(a,b,c)
Arithmetic functions	sqrt(a), min(a,b), max(a,b)

Table 4.1: Available operators and functions in simple text file input language

```

1 d_x = A_x - B_x;
2 d_y = A_y - B_y;
3 d_z = A_z - B_z;
4 distance = sqrt( d_x * d_x + d_y * d_y + d_z * d_z );

```

Listing 4.2: Computing distance between two points in 3D space

two points in 3D space in this language is shown in Listing 4.2.

Currently the target processor does not support loops or branches and neither does this input language, but they could be useful if added to the input language as a sort of preprocessor. Currently, if you wanted a function to compute the distance between two points in N dimensional space for $N = 2$ to 10 would require a separate file for each N used. With a preprocessor loop, the N could be used as the range for a loop and different code could be generated depending on N . Adding this behavior to the language would enable parameterized algorithms, but it would also add a lot of complexity. We instead decided to leverage the existing power of function templates in C++ to design more complicated algorithms.

4.2 C++ Template Functions

The other input method uses C++ template functions to input the algorithm to the compiler. When using this input method, the compiler no longer has the traditional

compiler flow. In order to compile a C++ template function called `my_algorithm` with the custom compiler, a main function needs to be written in C++. This main function needs to call `my_algorithm` with the type `graphMaker`, and then it needs to call functions in the compiler library. The DFG is generated by the call to `my_algorithm`, and the DFG is stored in a static variable in the class `graphMaker`. When the compiler's functions are called, it knows to look for the DFG in the class `graphMaker`.

To perform the compilation, first main needs to be compiled into an executable and linked with the custom compiler. Once the executable has been generated, it can be run to compile `my_algorithm` for the processor template. Listing 4.3 shows mock-up of a main function used to compile `my_algorithm` for the processor template.

```
1 #include "compiler.hpp"
2 #include "my_algorithm.hpp"
3
4 int main() {
5     graphMaker input[10];
6     graphMaker output[10];
7     my_algorithm<graphMaker>(input, output);
8     do_compile();
9     return 0;
10 }
```

Listing 4.3: Example main function for compiling `my_algorithm`

The function `my_algorithm` could have been written specifically for `graphMaker`, and the compilation results would still be the same, but by making it a template function, the same algorithm that was compiled for the processor template can be tested with numbers on a standard computer. Consider the code in Listing 4.4; by just changing the data type, the same write-up of the algorithm can be used to test the algorithm implementation for correctness.

4.2.1 graphMaker Class

The `graphMaker` class is the class used to generate DFGs from template C++ functions. Each `graphMaker` object is a container for a DFG node that can be combined with other `graphMaker` objects with the specified overloaded operators and functions

```

1 #include "my_algorithm.hpp"
2
3 int main() {
4     float input[10] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
5         10.0};
6     float output[10];
7     my_algorithm<float>( input , output );
8     for( int i = 0 ; i < 10 ; i++ ) {
9         std::cout << "output[" << i << "]" = " << output[i] << std::endl;
10    }
11    return 0;
12 }

```

Listing 4.4: Example main function for testing `my_algorithm`

to produce a node in the DFG that represents that function. In C++, the operators `+`, `-`, `*`, and `/` are normally used on numeric data types to do math associated with the operator. When running on `graphMaker` data, the operators `+`, `-`, `*`, and `/` are overloaded to add nodes to the DFG to represent those operations and return a `graphMaker` object containing the new node. All of the operations in Table 4.2 have been overloaded to work on `graphMaker` data to generate nodes of a DFG for each operation and return a `graphMaker` object containing the new node.

unary operators	<code>+</code> , <code>-</code>
binary operators	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>
assignment operators	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code>
relational operators	<code>lt(a,b)</code> , <code>lteq(a,b)</code> , <code>gt(a,b)</code> , <code>gteq(a,b)</code>
equality operators	<code>eq(a,b)</code> , <code>neq(a,b)</code>
conditional function	<code>cond(a,b,c)</code>
arithmetic functions	<code>sqrt(a)</code> , <code>min(a,b)</code> , <code>max(a,b)</code>

Table 4.2: Functions and operators overloaded for `graphMaker` to construct the DFG

Each individual `graphMaker` object points to a node in the DFG, but the compiler needs the entire graph to be able to process it. As each node is created, it is also added to a static member of the `graphMaker` class that contains the entire DFG. When the graph is done generating and the compiler is called, the custom compiler looks at the static member of `graphMaker` which contains the DFG to get the algorithm to compile.

4.2.2 Numerical Data Types

The same function that generates DFGs for the compiler to process can also be used to run the algorithm in a C++ program with floating point numbers. By replacing the template data type with `float`, `double`, or any other numerical data type, the same function that is used to generate DFGs can be used to compute the results of the target algorithm for provided inputs. This allows for algorithm designers to test the algorithm with the same code used to implement the algorithm in hardware.

Further algorithm tests can be performed by changing the data types in the algorithm. To get an approximation of the error from running an algorithm in single precision floating point, run the algorithm twice with the same inputs, once cast as `float` and once cast as `double`, and compare the results. The difference in the results will be an approximation of the error in the single precision floating point implementation of the algorithm.

By using custom data types, more aspects of the algorithm can be tested. To test non-standard floating point representations, a custom data type could be written as a class in C++ to emulate the custom floating point precision. As long as all of the operations in Table 4.2 are defined for the custom class, then the C++ template function can be used to test the performance of the algorithm with a custom floating point representation.

4.2.3 Helper Class Templates

The previous section shows how different data types can be used as long as they have definitions for the operations in Table 4.2. In a similar manner, any template class can be used within functions targeting the template processor as long as that template class only uses the operations in Table 4.2 on objects of the template data type.

To easily write linear algebra algorithms for the compiler, we created a matrix class template `matrix<T>` that only uses the functions shown in Table 4.2 on its template type `T`. Since the matrix class only uses those operations, it can be used in

any template C++ function that is targeting the compiler. This matrix class can be used to efficiently build up larger algorithms with `matrix<graphMaker>` objects, and the algorithms can still be tested with `matrix<float>` objects. If the algorithm was written correctly, switching out these objects remains as easy as before.

More details on the matrix class template can be found in Appendix A.

4.3 Front-End Optimizations

When the compiler is building a DFG using `graphMaker` objects, the compiler can tell if a new `graphMaker` represents a constant or a variable by how the node is created. If the node is created through a cast from an `int`, `float`, or `double`, the node is constant. If the node is created through assignment from a `graphMaker` object, then it is a constant only if the other object is a constant. That leaves nodes created through functions on `graphMaker` objects. Without front-end optimizations enabled, nodes created through functions on `graphMaker` objects are never constants. By enabling front-end optimizations, primarily constant folding, these nodes can be constants.

4.3.1 Constant Folding

The main front-end optimization is constant folding. If a function that creates a `graphMaker` object depends only on other `graphMaker` objects that contain constant value nodes, then the function can be evaluated for those constant values and the result of the function will be a `graphMaker` object that contains a constant value node equal to the result. There are also some cases where knowing a single input to a function can be used to optimize the output. These cases utilize special properties of the arithmetic operations such as identity elements.

One of the cases, optimizing `lt(x,0)` into just `x`, relies on the architectural representation of true and false values in the processor. In this architecture, the sign bit of floating point numbers is used to express boolean values. That makes negative numbers true and positive numbers false. Therefore `lt(x,0)` will return a negative

Before	After
$x+0, 0+x$	x
$x-0$	x
$x*0, 0*x$	0
$x*1, 1*x$	x
$0/x$	0
$x/1$	x
$lt(x,0)$	x

Table 4.3: Improvements made by constant folding in graphMaker with only one constant

number if and only if x is negative. If only the boolean value of $lt(x,0)$ matters and not the actual floating point value, then $lt(x,0)$ can be replaced directly with x . Table 4.3 shows all the improvements made by constant folding when only one of the operands is a constant.

4.3.2 Constant Checking

Once constant folding is enabled, there are potentially many compile time constants that the algorithm designer did not explicitly set. It makes sense to allow the designer to check for constant values to see if their algorithm could be improved by knowing constant values.

Need for the ability to check compile-time constants can be seen in Gaussian elimination. Consider the implementation of Gaussian elimination in Listing 4.5. This algorithm will not work if $A(i,i)$ ends up being zero because there is a division by $A(i,i)$. Sometimes $A(i,i)$ depends on many different inputs, but other times, especially in sparse matrices, these intermediate values may be known to be zero at compile-time through constant folding. If $A(i,i)$ is known to be zero, then the entire i th row can be swapped with a lower row, and the process can continue.

If $A(i,i)$ is zero because of compile time constants, then it will always be zero, and the implementation of Gaussian elimination in Listing 4.5 will never work. This motivates the need for designers to be able to check for compile-time values so they can modify the algorithm depending on these values. To enable modifying algorithms

```

1 template <class T>
2 matrix<T> row_eschelon( const matrix<T>& A_in ) {
3     int M = A_in.get_m(); // number of rows
4     matrix<T> A = A_in;
5     for( int i = 0; i < M; i++ ) {
6         A.row(i) = A.row(i) / A(i, i);
7         for( int j = i+1; j < M; j++ ) {
8             A.row(j) = A.row(j) - A(j, i) * A.row(i);
9         }
10    }
11    return A;
12 }

```

Listing 4.5: Templated Gaussian elimination algorithm using the matrix class

Numerical functions	<code>isZero(a)</code> , <code>isOne(a)</code>
Boolean functions	<code>isTrue(a)</code> , <code>isFalse(a)</code>

Table 4.4: Functions available for `graphMaker` to check constant nodes

based on constants, a few functions were added that take in `graphMaker` nodes and return boolean values depending on the compile-time constant of inputted node. These functions can be seen in Table 4.4. In all of the functions, if `a` is not a compile time constant, then the function returns false. Otherwise the function returns true if the constant value matches what the function is checking for.

Continuing with the Gaussian elimination example, the function `isZero` can be used inside the template function to detect zeros known at compile-time. If the known structure of the input matrix would cause Gaussian elimination to divide by zero without pivoting, then the algorithm in Listing 4.6 is able to pivot the matrix at compile-time to avoid that division.

4.3.3 `constTracker`

By adding the constant checking functions to the C++ template algorithms, we introduced a function that is not already defined on numeric data types such as `float`. Simply defining `isZero` and the others for `float` is not an option because `isZero` needs to return false if the input is not a compile-time constant, and that information is not encoded into `float` objects. To fix this problem, we introduced

```

1  template <class T>
2  matrix<T> row_echelon( const matrix<T>& A_in ) {
3      int M = A_in.get_m();
4      matrix<T> tmp, A = A_in;
5      int i, j;
6      for( i = 0; i < M; i++ ) {
7          j = i;
8          while( isZero(A(j,i)) ) {
9              j++;
10             if( j >= M ) throw matrix_singular();
11         }
12         if( i != j ) {
13             // swap rows i and j since A(i,i) is zero
14             tmp = A.row(i);
15             A.row(i) = A.row(j);
16             A.row(j) = tmp;
17         }
18         A.row(i) = A.row(i) / A(i,i);
19         for( j = i+1; j < M; j++ ) {
20             A.row(j) = A.row(j) - A(j,i) * A.row(i);
21         }
22     }
23     return A;
24 }

```

Listing 4.6: Templated Gaussian elimination algorithm with compile-time pivoting

the class `constTracker<T>` as a container for any data type to track if it is a compile time constant or not. The types `constTracker<float>` and `constTracker<double>` act like `float` and `double` for all of the functions in the top part of Table 4.2, but they contain extra information that tracks if the value could have been known at compile-time or not.

This tracking is done through a combination of static state variables within the `constTracker<T>` class and boolean values stored locally in each object. The static state variable says if casts from type `T` to type `constTracker<T>` should be constants or non-constants at the current time in the program. The local boolean value tracks for each value if it is a compile-time constant or not.

When the input data is being populated, casts from `T` to `constTracker<T>` should be treated as non-constants, and only certain elements in the input data should be treated as constants. Inside the algorithm, when something is cast from `T`, it should be treated as a constant because at that point, all non-constant inputs should already

be defined.

The results of functions on `constTracker<T>` are denoted as constants or non-constants depending on the inputs. For example if both inputs to `operator+` are constants, then the resulting `constTracker<T>` will also be constant. If one of the inputs to `operator*` is a constant zero, then the result will be a constant zero regardless of the other value. `constTracker<T>` has support for all of the constant folding optimizations in Table 4.3 and constant checking functions in Table 4.4 so the results from `constTracker<float>` will still match the results of the algorithm run with `graphMaker`.

4.4 Summary

This chapter introduced the two methods of algorithm input to the compiler: plain text and C++ template functions. Plain text provides a simple interface to the compiler. C++ template functions provide a more advanced input method.

As shown in this chapter, C++ template functions can be used to write advanced algorithms that depend on variable sized inputs. Also the same C++ template functions that generates a DFG can be used with different template data types to run the algorithm with floating point numbers. The next chapter relies on C++ template functions with variable sized inputs to explore the compiler performance across various problem sizes of different algorithms.

Chapter 5

Compiler performance

The performance of a numerical accelerator is measured in terms of latency, power consumption, and area. The compiler can contribute to the performance in these areas, but the compiler's performance can also be measured separately from the generated numerical accelerator's performance. The compiler is designed to efficiently map algorithms to the hardware by generating schedules that are as short as possible and memory assignments that require as few memories as possible, so it makes sense to measure the compiler in those areas.

In this chapter we will introduce a collection of algorithms that have been compiled with the custom compiler presented in Chapter 3. First the compiler will be used to measure some tradeoffs between two algorithms that perform the same function. Then some of these algorithms will be used to test the compiler's scheduling and memory assignment. The compiler's list scheduling approach is measured against theoretical bounds for execution times from the DFGs. The compiler's memory assignment algorithm is measured against the minimum number of memories required for a processor to be able to run at full throughput (one memory for each FPU input). Finally some compiler optimizations will be introduced to the algorithm and compared to hand optimizations for an algorithm.

5.1 Algorithms

Each algorithm in the section was written for the compiler using C++ function templates. Each algorithm works on variable size inputs, so in many of the tests performed later in the chapter, the tests are swept across input sizes.

5.1.1 LU Decomposition

LU decomposition is a matrix decomposition algorithm for square matrices A that produces a lower triangular matrix L and an upper triangular matrix U such that $A = LU$. The decomposition is not unique unless some of the diagonals are fixed to be one. For this section, we are assuming L to be a unit triangular matrix meaning all the entries along the diagonal are one. The algorithm we tested for computing LU is shown in Algorithm 5.1

Algorithm 5.1 LU decomposition

```
 $L \leftarrow I$   
 $U \leftarrow A$   
for  $i = 0 \rightarrow N - 1$  do  
  for  $j = i + 1 \rightarrow N - 1$  do  
     $L_{ji} \leftarrow U_{ji}/U_{ii}$   
  end for  
  for  $j = i + 1 \rightarrow N - 1$  do  
     $U_{ji} \leftarrow 0$   
  end for  
  for  $j = i + 1 \rightarrow N - 1$  do  
    for  $k = i + 1 \rightarrow N - 1$  do  
       $U_{jk} = U_{jk} - L_{ji} * U_{ik}$   
    end for  
  end for  
end for
```

There is a variant of LU decomposition called LDU decomposition that forces L and U to both be unit triangular, but it introduces a diagonal matrix D to the decomposition. When the input matrix is symmetric for LDU matrix decomposition, $U = L^T$, so the decomposition can be written LDL^T .

5.1.2 LDL^T Decomposition

LDL^T decomposition is a matrix decomposition algorithm for symmetric matrices. This algorithm decomposes the matrix A into unit lower triangular matrix L and diagonal matrix D such that $A = LDL^T$. The values of L and D are commonly written as [16]

$$D_{ii} = A_{ii} - \sum_{k=0}^{i-1} L_{jk}^2 D_{kk} \quad (5.1)$$

$$L_{ij} = \frac{1}{D_{jj}} \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} D_{kk} L_{jk} \right) \quad (5.2)$$

Those two formulas can be used to create an algorithm to compute L and D as shown in Algorithm 5.2. Examining the computations done in the algorithm reveals an inefficiency: L_{ji} is computed using a division by D_{ii} , but L_{ji} is later multiplied by D_{ii} to calculate other values. These multiplications by D_{ii} undo previous divisions, and they should not be necessary assuming you have enough space to store the intermediate values obtained when calculating L_{ji} .

Removing these multiplications by hand optimization results in the algorithm shown in Algorithm 5.3. Both algorithms were implemented to test compiler optimizations against hand optimizations.

Algorithm 5.2 LDL^T decomposition without hand optimization

```
for  $i = 0 \rightarrow N - 1$  do  
   $D_{ii} \leftarrow A_{ii} - \sum_{k=0}^{i-1} L_{jk}^2 D_{kk}$   
  for  $j = i + 1 \rightarrow N - 1$  do  
     $L_{ji} \leftarrow \frac{1}{D_{ii}} \left( A_{ji} - \sum_{k=0}^{i-1} L_{jk} D_{kk} L_{ik} \right)$   
  end for  
end for
```

Algorithm 5.3 LDL^T decomposition with hand optimization

```
for  $i = 0 \rightarrow N - 1$  do  
   $D_{ii} \leftarrow A_{ii} - \sum_{k=0}^{i-1} L_{jk}P_{jk}$   
  for  $j = i + 1 \rightarrow N - 1$  do  
     $P_{ij} \leftarrow A_{ij} - \sum_{k=0}^{j-1} P_{ij}L_{jk}$   $\triangleright P_{ij}$  is the product of  $L_{ij}$  and  $D_{jj}$   
     $L_{ij} \leftarrow P_{ij}/D_{jj}$   
  end for  
end for
```

5.1.3 Cholesky Decomposition

Cholesky decomposition is a matrix decomposition algorithm for symmetric positive definite (SPD) matrices. The algorithm decomposes A into LL^T where L is a lower triangular matrix. This algorithm is often used for solving linear systems $Ax = b$ where the matrix A is SPD. The equations for the matrix L are shown below [11]

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2} \quad (5.3)$$

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik}L_{jk} \right) \quad (5.4)$$

The sequential algorithm to calculate the matrix L is shown in Algorithm 5.4. This algorithm requires four core types from the processor: *Addsub*, *Mul*, *Div*, and *Sqrt*.

Algorithm 5.4 Cholesky decomposition

```
for  $i = 0 \rightarrow N - 1$  do  
   $L_{ii} \leftarrow \sqrt{A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2}$   
  for  $j = i + 1 \rightarrow N - 1$  do  
     $L_{ji} \leftarrow \frac{1}{L_{ii}} \left( A_{ij} - \sum_{k=0}^{i-1} L_{ik}L_{jk} \right)$   
  end for  
end for
```

5.1.4 Discrete Fourier Transform

The Fourier transform is a transformation from continuous signals in the time domain to continuous signals in the frequency domain. When the input signal is discrete, the discrete Fourier transform (DFT) can be used instead to produce its discrete frequency components. The formula for the DFT of a complex N element vector x is shown below.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \quad (5.5)$$

Each entry in X is the dot product of x with a vector of coefficients, so the transform can be written as a complex matrix vector multiplication $X = Wx$ where W is

$$W = \begin{bmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & \dots & w^{N-1} \\ w^0 & w^2 & w^4 & \dots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{N-1} & w^{2(N-1)} & \dots & w^{(N-1)(N-1)} \end{bmatrix} \quad (5.6)$$

and w is the N^{th} root of unity given by the formula

$$w = e^{-i2\pi/N}. \quad (5.7)$$

The DFT is implemented as complex matrix-vector multiplication where the matrix is constant and the vector is the input.

5.1.5 Fast Fourier Transform

The fast Fourier transform (FFT) is an implementation of the DFT that partitions the problem into smaller problems and then recombines them using certain properties of the DFT [10].

The algorithm for implementing FFT is shown in Algorithm 5.5.

Algorithm 5.5 FFT

```
for all  $i = 0 \rightarrow N - 1$  do  
     $x_{br}(i) \leftarrow x(\text{bitreverse}(i))$   
end for  
for all  $step = 1 \rightarrow N - 1$  do  
     $jump \leftarrow step \ll 1$   
     $delta \leftarrow -\pi/step$   
     $sine \leftarrow \sin(delta \cdot 0.5)$   
     $multiplier \leftarrow -2.0 \cdot sine \cdot sine + i \sin(delta)$   
     $factor \leftarrow 1$   
    for all  $group = 0 \rightarrow step - 1$  do  
        for  $pair = group \rightarrow N - 1$  with step  $jump$  do  
             $match \leftarrow pair + step$   
             $product \leftarrow factor * x_{br}(match)$   
             $x_{br}(match) = x_{br}(pair) - product$   
             $x_{br}(pair) = x_{br}(pair) + product$   
        end for  
         $factor = multiplier \cdot factor + factor$   
    end for  
end for  
return  $x_{br}$ 
```

5.2 Compiler Driven Algorithm Exploration

The compiler is typically used to generate schedules for processor configurations, but due to the analysis it does along the way, it can also be used for general algorithm exploration. By using an arbitrary processor configuration, algorithms can be compared relative to each other to determine the differences in number of operations and critical path.

Consider the DFT and FFT algorithms presented in the previous section. The FFT algorithm is named "Fast" because of the reduced execution time on CPUs versus other DFT implementations. This reduced execution time comes from FFT requiring fewer operations. Since our processors have parallel units, the number of operations does not explain execution time by itself; the critical path also needs to be considered.

By using a simple processor, we can compare the latency bounds, and we can compare the throughput bounds. Figure 5-1 shows the comparison of the latency

bounds for both the DFT and the FFT; Figure 5-2 compares the throughput bounds. These figures show that the reduction in the number of operations in the FFT comes at a cost in critical path. If the current combination of problem size and processor configuration is latency limited, then the DFT will beat FFT, but if the system is throughput limited, then the FFT will remain faster.

5.3 Three Sizes of Processors

We will begin the exploration of the compiler's performance when scheduling algorithms for processor configurations with three example processors used to represent different sized processors and the tradeoffs involved in picking the number of operational units in the processor. The three processors assume fixed latencies for the different operational units, and as the processors grow, one extra stage of crossbar latency is added to the processor.

The three processor configurations can be seen in Table 5.1.

Table 5.1: Processor parameters

		Small	Medium	Large
Number of Units		4	12	20
Addsubs	Number	1	3	5
	Latency	5	5	5
Muls	Number	1	3	5
	Latency	3	3	3
Divs	Number	1	3	5
	Latency	15	15	15
Sqrts	Number	1	3	5
	Latency	15	15	15
Crossbar latencies		0	1	2

5.3.1 Scheduling

The following figures show how the compiler performs when scheduling three different matrix decomposition algorithms for three processors. Figure 5-3 shows the scheduling results for different sized LDL^T decompositions. The figure shows that depending

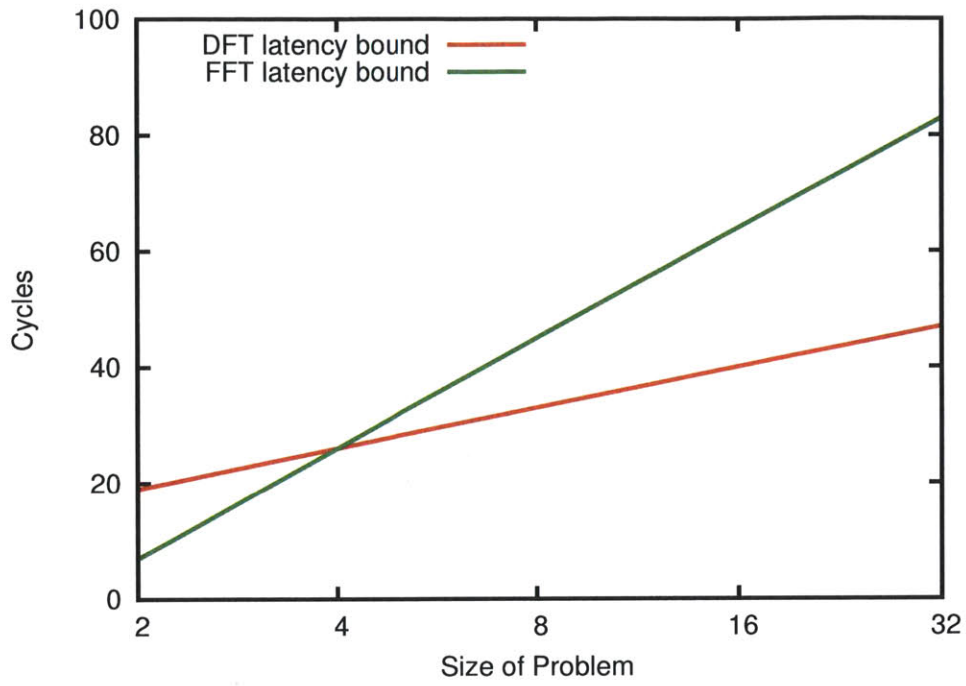


Figure 5-1: Latency bound results for DFT and FFT

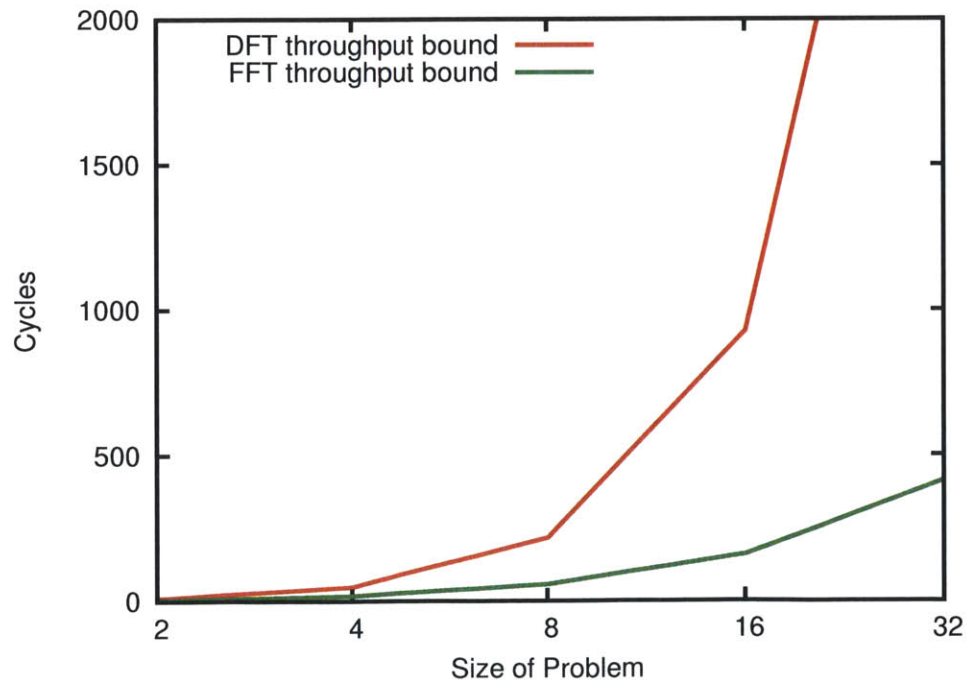


Figure 5-2: Throughput bound results for DFT and FFT

on the size of the LDL^T decomposition being run, the ideal processor of the three processors changes. The actual scheduling curves are close to the bounds showing that the compiler is scheduling well.

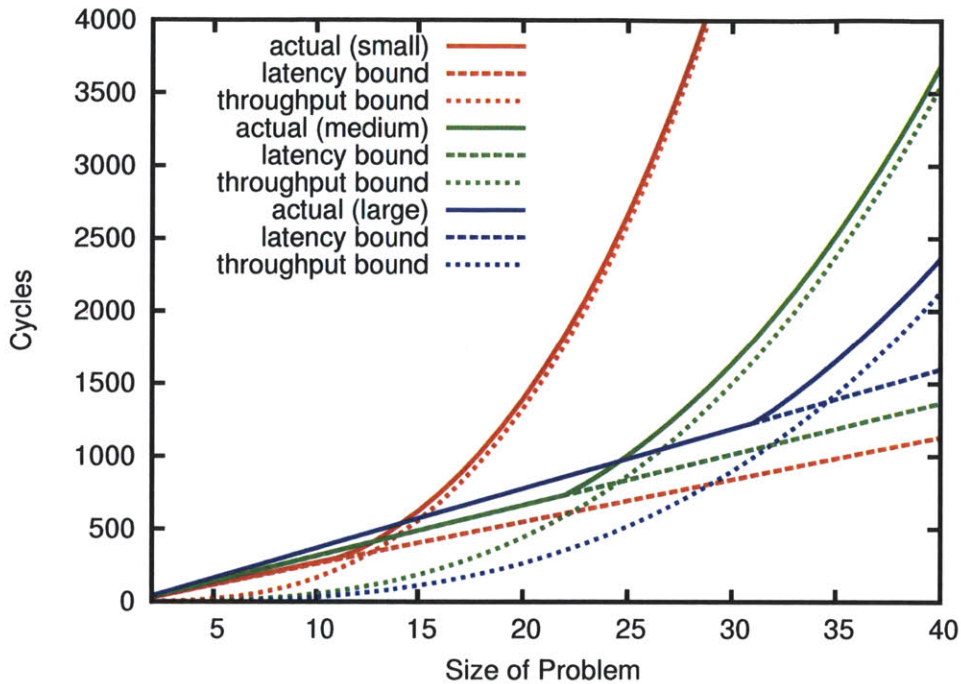


Figure 5-3: Scheduling results for LDL^T decomposition

Figure 5-4 shows the scheduling results for different sized LU decompositions. The same trends seen in Figure 5-3 also appear here.

Figure 5-5 shows the scheduling results for different sized Cholesky decompositions.

5.3.2 Memory Assignment

After scheduling, the compiler assigns memories to each intermediate result of the computation. This assignment does not assume a set number of memories. Instead it provides the designer information about how many memories it needs. A processor with N operational unit inputs requires N memories to be able to send a different value to each input. The memory assignment algorithm is expected to use at least N memories during assignment, but ideally it would use as few as possible.

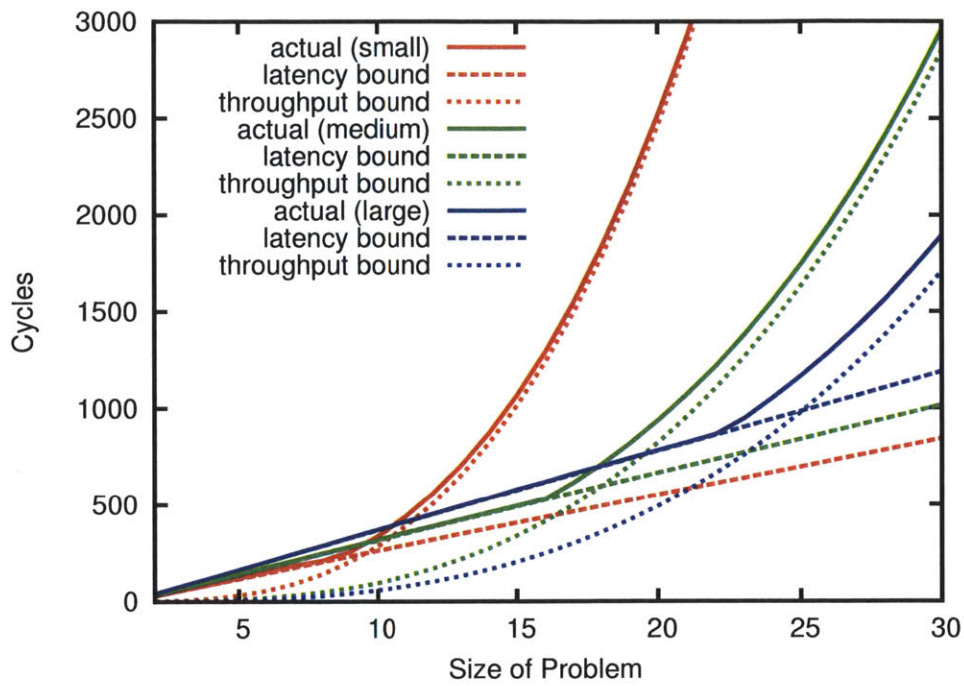


Figure 5-4: Scheduling results for LU decomposition

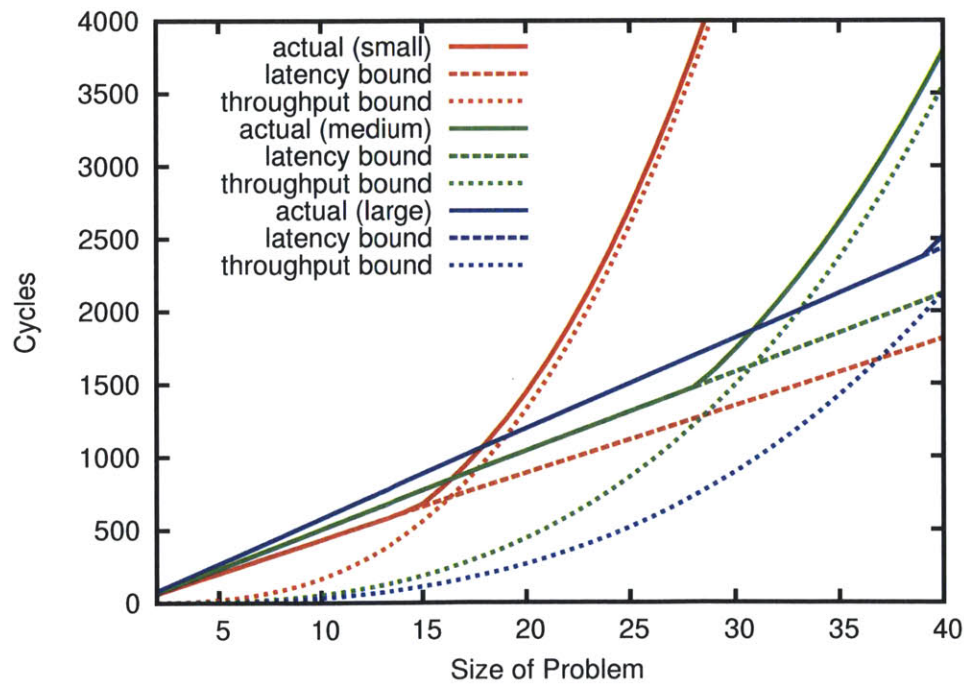


Figure 5-5: Scheduling results for Cholesky decomposition

Figure 5-6 shows the results of memory assignment for LDL^T decomposition. The expected minimum number of memories for these processors count the inputs on *AddSubs*, *Muls*, and *Divs*, but not *Sqrt*, because the *Sqrt* units are not used. While the number of memories required increases with the size of the problem, it does not increase far beyond the expected minimum number of memories for this target algorithm.

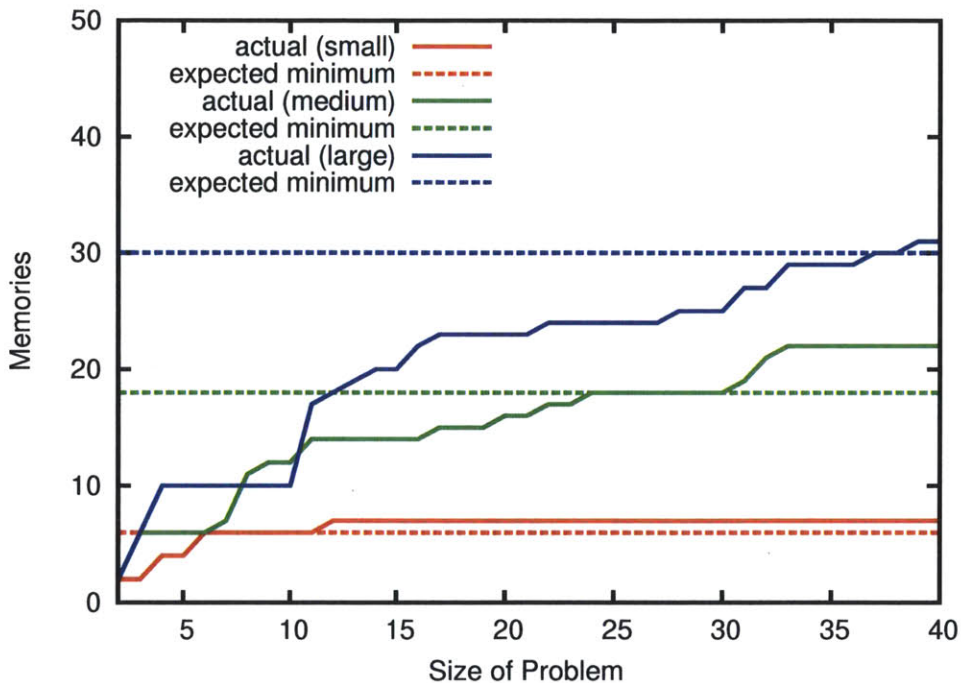


Figure 5-6: Memory assignment results for LDL^T decomposition

Figure 5-7 shows the results of memory assignment for LU decomposition. Again, the expected minimum number of memories for these processors does not count the *Sqrt* inputs. The medium and large processors do not even meet the expected minimum number of memories. There are two reasons a processor does not meet the minimum number of memories: either there is data reuse between operations that are scheduled during the same cycle, or not all of the units are used during the same cycle. For LU decomposition there are fewer divide instructions than add, subtract, and multiply, but these processors have the same number of each type of unit. Therefore the utilization for the dividers will be lower than the utilization for the

other units, and therefore the dividers do not need as many inputs as the other units per cycle which reduces the need for more memories.

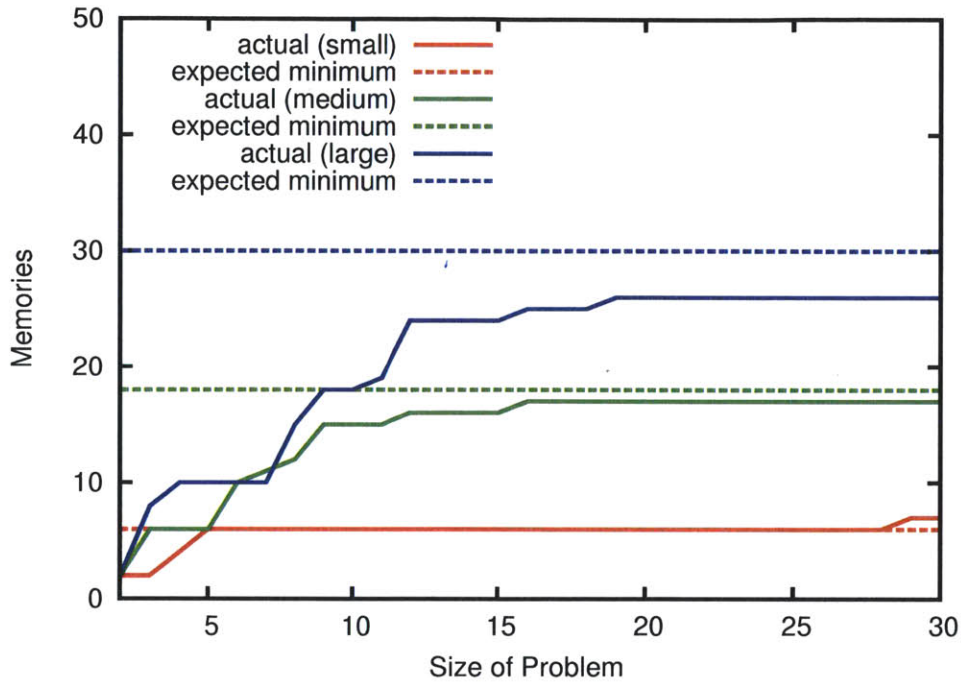


Figure 5-7: Memory assignment results for *LU* decomposition

Figure 5-8 shows the results of memory assignment for Cholesky decomposition. Unlike the other two examples shown, this example requires a linearly increasing number of memories as the decomposition gets larger. This suggests that, unless the size of the decomposition is no more than the number of operational unit inputs, then the memory assignment algorithm will be inefficient in this case.

This behavior is not always a result of the memory assignment algorithm. Sometimes it is the result of the structure of the DFG and limiting values to only one memory. For example, the outer product operator xx^T has a fully connected memory dependency graph regardless of schedule since each pair of inputs get multiplied at some point. A fully connected memory dependency graph requires one memory for each variable, so the outer product operator always requires N memories where N is the size of x .

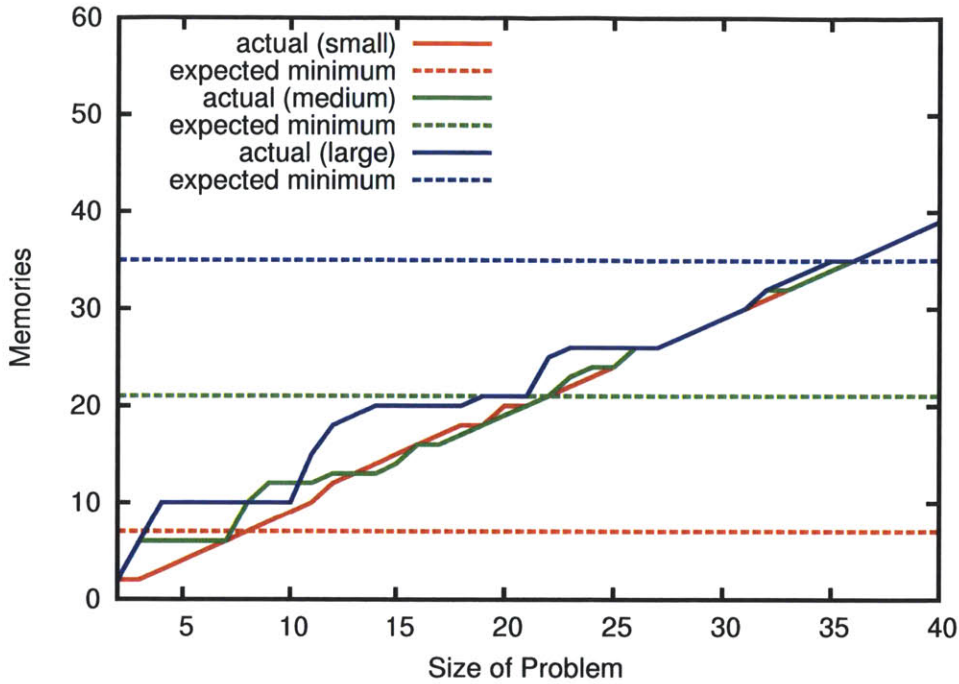


Figure 5-8: Memory assignment results for Cholesky decomposition

5.4 Redistributing Units

The previous section examined three processors for a variety of applications, but since the compiler is targeting a parameterized processor template, the processor can be designed with parameters specifically targeting the desired application. Throughout this section we will be exploring the optimal distribution of units for each size of processor for the LDL^T algorithm.

The LDL^T decomposition algorithm requires additions, subtractions, multiplications, and divisions, but it does not require any square roots, so the target processor should not have square root units. The processors in Table 5.1 all have square root units, so when these processors are optimized for LDL^T , those units will be replaced with units of different types.

With these new processors, the scheduling results are still close to the bounds as shown in Figure 5-9. Also, the memory assignments are still close to the expected minimum as seen in Figure 5-10.

Table 5.2: Processor parameters with units redistributed for LDL^T

		Small	Medium	Large
Number of Units		4	12	20
Addsubs	Number	2	6	10
	Latency	5	5	5
Muls	Number	1	5	9
	Latency	3	3	3
Divs	Number	1	1	1
	Latency	15	15	15
Sqrts	Number	0	0	0
	Latency	15	15	15
Crossbar latencies		0	1	2

The improvements from specializing the types of operational units can be seen by comparing the new schedules with the old schedules for specific processor size. For example, the gains in the medium sized processor are shown in Figure 5-11.

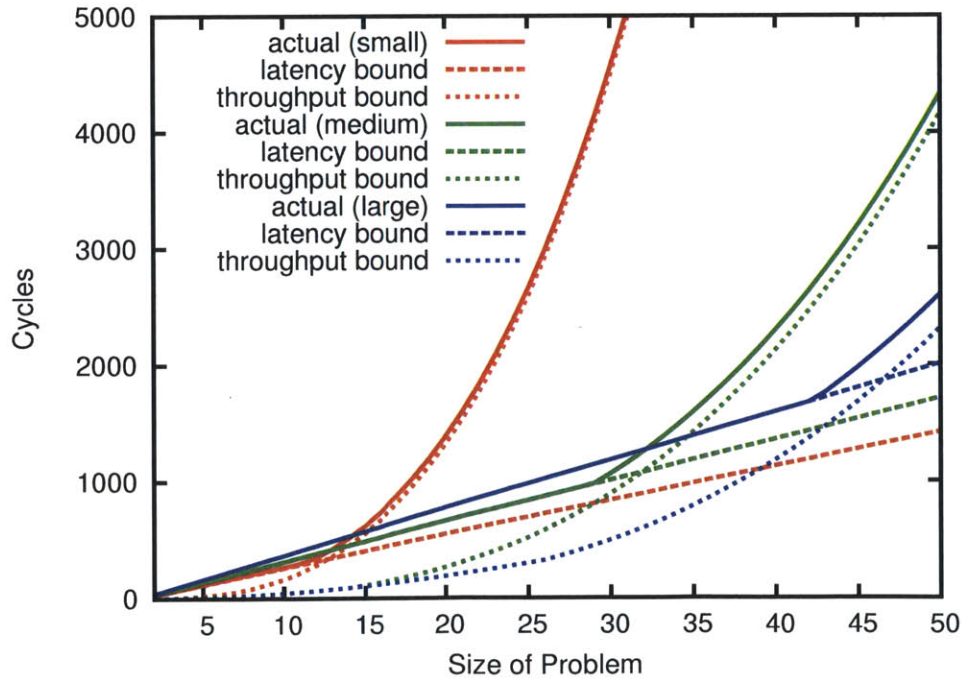


Figure 5-9: Scheduling results for improved LDL^T processors

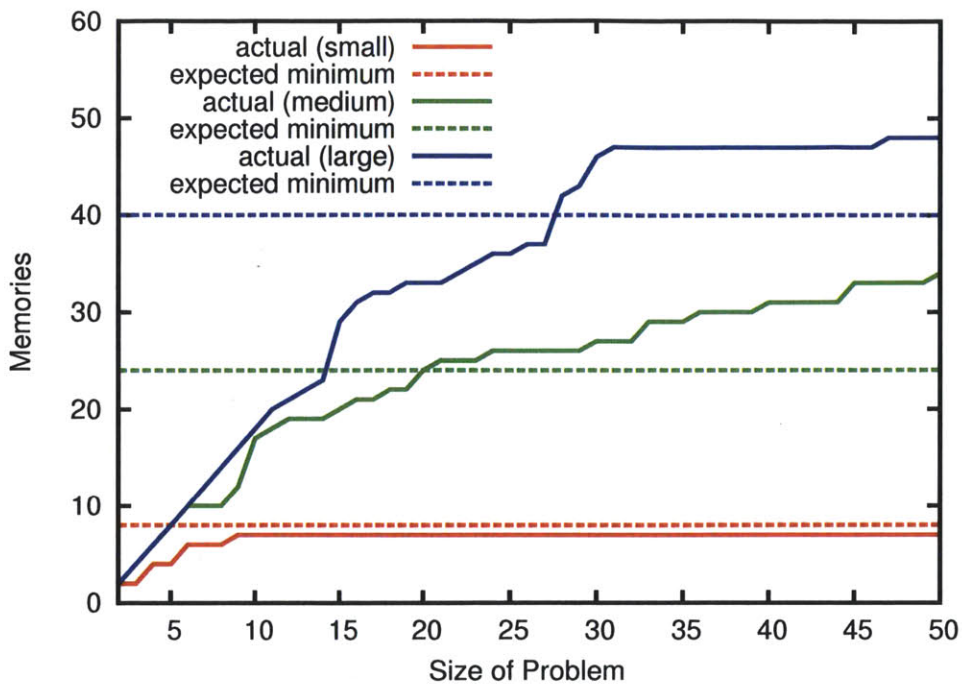


Figure 5-10: Memory assignment results for improved LDL^T processors

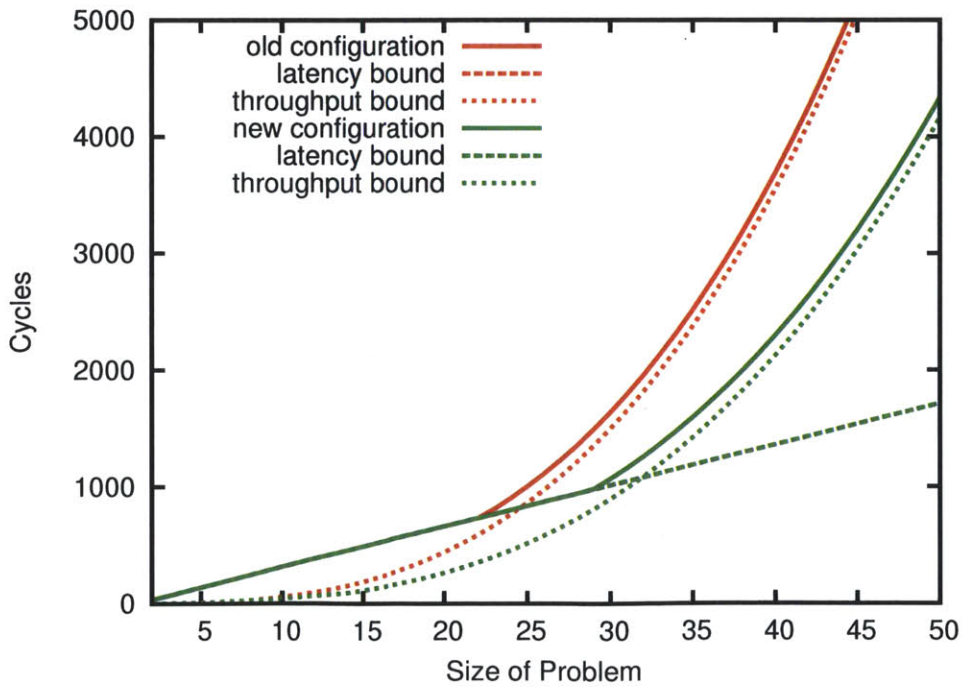


Figure 5-11: Scheduling improvement for the medium LDL^T processor

5.5 Optimizations

The compiler optimizations can also be tested for LDL^T decomposition. There are two versions of the LDL^T algorithm written for the processor. One is simply a translation from equations that express each element in terms of other elements in A , L , and D . This simple version can be seen in Algorithm 5.2.

There is also a hand optimized version that removed number of unneeded operations. This hand optimized version can be seen in Algorithm 5.3.

Figure 5-12 shows that there are no gains when going from no optimization (-00) to optimization level -01, but going to optimization level -02 improves the critical path by about 15%. Optimization level -03 includes some aggressive optimizations that increase the number of nodes in the DFG. The increased number of nodes results in a higher throughput bound, but at the same time, the optimizations reduce the critical path which improves the performance over -02 from $N = 2$ to $N = 15$.

Figure 5-13 shows that there are still some gains from using the compiler's opti-

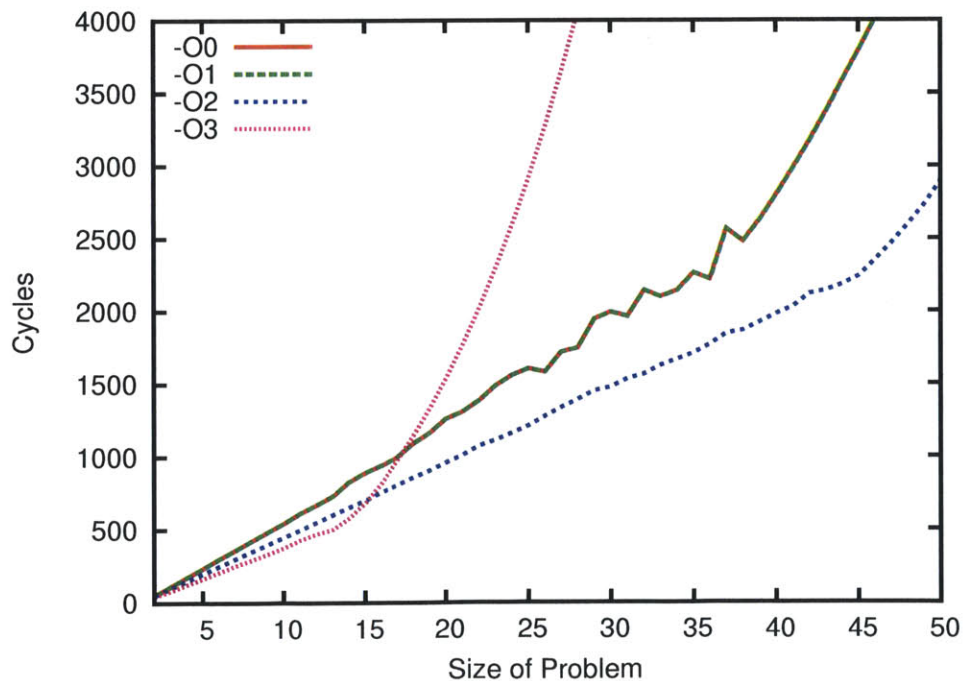


Figure 5-12: Optimization improvements for an unoptimized LDL^T algorithm

mization on top of the existing hand optimizations in the algorithm. Like the other version of the algorithm, there are no gains from using -01. When -02 is used, the latency bound drops causing the duration of the schedules to drop with it. -03 produces more operations and increases the throughput bound like in Figure 5-12, but in this version of the algorithm, the latency bound does not drop any further than -02. Therefore for this algorithm, it is worse to use -03 than -02 because the optimizations -03 enables do not outweigh the penalty of duplication operations to further collapse nodes.

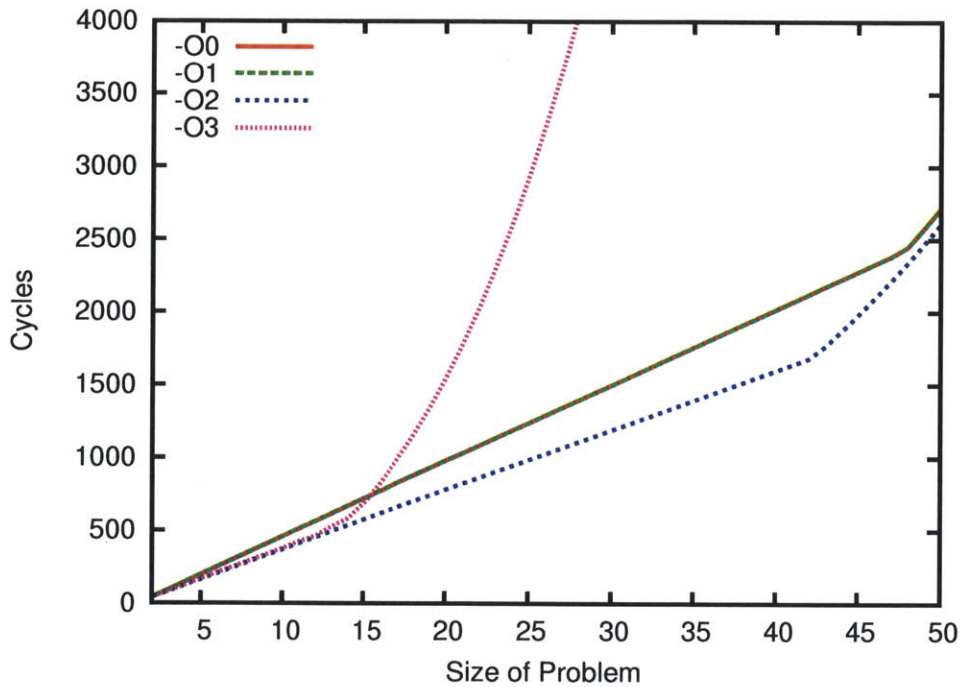


Figure 5-13: Optimization improvements for a hand optimized LDL^T algorithm

Figure 5-14 shows results from both algorithms for different levels of optimization in the same graph. One interesting thing shown in this graph is that for small problem sizes ($N < 15$), the unoptimized algorithm compiled with -03 optimizations meets the performance of the hand optimized algorithm compiled with additional compiler optimizations. At $N = 15$, the inefficiency of having the compiler optimize the code for you catches up, and the performance of the unoptimized algorithm with -03 becomes throughput limited.

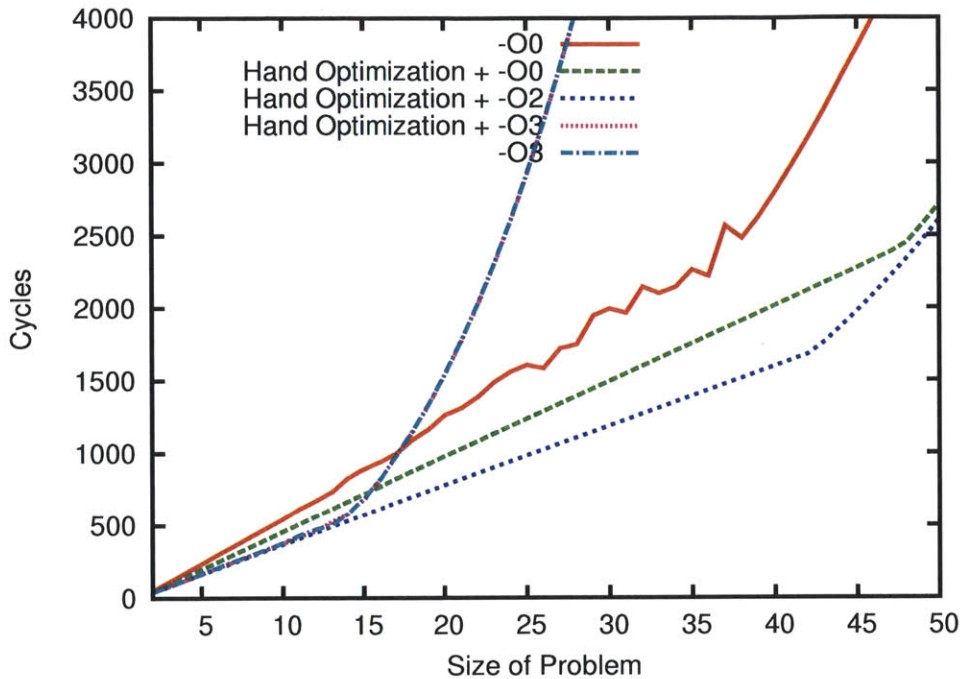


Figure 5-14: Optimization improvements for LDL^T (both hand and compiler optimizations)

5.6 Summary

In this chapter, we showed how the compiler performs in various conditions for a collection of algorithms. First the compiler was used to perform some high-level algorithm exploration comparing throughput and latency bounds for DFT and FFT algorithms. Next the compiler was used to map matrix decomposition algorithms to various processors. To improve the results for LDL^T decomposition, the processor configurations were modified to have the optimal distribution of unit types. Finally optimizations were added to a standard LDL^T algorithm and a hand optimized algorithm to show how compiler optimizations are able to compete against hand optimizations.

Throughout this chapter, the scheduling performance stays close to one of the two performance bounds for execution time. This shows that the list scheduling algorithm is efficient and is able to approach theoretical bounds for scheduling performance. With the exception of Cholesky decomposition, all of the algorithms in

this section support the claim that memory assignment can be done after scheduling without a large penalty on the number of memories required for operation. These results combine together to show the effectiveness of the statically scheduling compiler presented in this thesis.

Chapter 6

Conclusion

The trade-off between power and performance in general purpose processors and embedded processors makes custom hardware attractive to embedded system designers. The large design effort and low flexibility of typical custom hardware designs normally prevent algorithm exploration. A parameterized custom hardware design that achieves low power and high performance while keeping flexibility is very attractive, but it is questionable how much design effort would be required to specialize the design to a given algorithm.

The compiler we presented in this thesis keeps the design effort low for specializing a processor template to a desired algorithm. The compiler, paired together with the processor template, generates numerical accelerators from standard C++ function templates. In a measure to further reduce design effort, the same C++ function that generates instructions for the hardware can also be used to test the correctness and accuracy of the algorithm. In addition, comparisons between the theoretical bounds for execution times and actual scheduling results shows that often times the compiler generated latency for an algorithm is normally close to the latency or throughput bound.

Throughout this thesis, results for varying problem sizes and processors shows that looking at the number of operations in an algorithm, as is the conventional way of looking at algorithms for CPUs, is not always valid for custom processor template. CPUs have a low number of FPUs with clock cycle latencies comparable to our

processor's latencies [2]. The low number of FPUs cause CPUs to be limited by the throughput bound therefore making the performance more dependant on the number of operations. When the number of FPUs is increased, which is often the case for our processors, the throughput bound decreases and gives way to the latency bound. The latency bound is not affected by the number of operations; it is only affected by the critical path of the algorithm. To get the full story of algorithm performance on our processor template, both aspects of the algorithm's structure, number of operations and critical path, must both be considered.

The compiler also provides the flexibility required to explore the design space made up by having configurable hardware running easily modified algorithms. Chapter 5 provides a brief view of how processors can be specialized for an algorithm by fixing the number of units in a processor, then adjusting the types of units, and then applying optimizations. The problem gets harder once post-synthesis clock frequencies are considered in the algorithm's execution time. This compiler, along with its target processor template, enables research on how well an entire class of algorithms can be implemented in parameterized hardware, and how to find the optimal algorithm-hardware pair for a given goal.

Appendix A

Matrix class

A.1 Motivation

Writing matrix algorithms using arrays and element-by-element operations is cumbersome and inefficient from a designer's point of view. Fundamental operations such as matrix multiplication requires multiple levels of nested loops to perform. Consider the implementation of Gaussian elimination shown in Listing A.1. This implementation uses one dimensional arrays and it requires for loops nested 3 deep. In addition, this code is hard to read because the fundamental matrix-level operations that are occurring are split up into smaller components so it can be coded up.

A.2 Overview of Matrix Class

The custom `matrix` class template can be used to represent matrices of templated data types. Each operation in the custom library can be performed on any data type that has the operations in Table 4.2 defined for it. The library is made up of common operators (Table A.1), elementary matrix row-column operations (Table A.2), and special functions (Table A.3). There is also a collection of linear algebra routines written for matrices.

This matrix class allows for programming algorithms in a manner that is closer to how people think about the algorithms. The Gaussian elimination example introduced

```

1 template <class T>
2 void row_echelon(T* in, int M, int N, T* out) {
3     for( int i = 0; i < M*N; i++ ) {
4         out[i] = in[i];
5     }
6     for( int row1 = 0; row1 < M; row1++ ) {
7         for( int col = row1; col < N; col++ ) {
8             out[N*row1+col] = out[N*row1+col] / out[N*row1+row1];
9         }
10        for( int row2 = row1+1; row2 < M; row2++ ) {
11            for( int col = row1; col < N; col++ ) {
12                out[N*row2+col] = out[N*row2+col] - out[N*row2+row1] *
13                    out[N*row1+col];
14            }
15        }
16    }

```

Listing A.1: Templated gaussian elimination algorithm

unary operators (matrix)	+, -
binary operators (matrix-matrix)	+, -, *
binary operators (matrix-scalar)	*, /
assignment operators (matrix-matrix)	=, +=, -=, *=
assignment operators (matrix-scalar)	*=, /=

Table A.1: Operators defined on matrix class

above can be programmed using the `matrix` class and its row operations in a more intuitive way. The rewritten algorithm can be seen in Listing A.2.

See Appendix B for more examples of this class in action.

```

1 template <class T>
2 matrix<T> row_eschelon( const matrix<T>& A_in ) {
3     int M = A_in.get_m(); // number of rows
4     matrix<T> A = A_in;
5     for( int i = 0; i < M; i++ ) {
6         A.row(i) = A.row(i) / A(i, i);
7         for( int j = i+1; j < M; j++ ) {
8             A.row(j) = A.row(j) - A(j, i) * A.row(i);
9         }
10    }
11    return A;
12 }

```

Listing A.2: Templated Gaussian elimination algorithm using the matrix class

<code>A.row(i)</code>	Returns a matrix containing just row <code>i</code> of <code>A</code>
<code>A.col(i)</code>	Returns a matrix containing just column <code>i</code> of <code>A</code>
<code>A.block(imin, jmin, imax, jmax)</code>	Returns a matrix containing rows <code>imin</code> to <code>imax</code> and columns <code>jmin</code> to <code>jmax</code> of <code>A</code>
Note: all of these returned matrices can be used as <i>lvalues</i> in assignments to update the corresponding elements in <code>A</code>	

Table A.2: Row, column, and block access operations defined for the matrix class

<code>ones<fp>(int m, int n)</code>	Creates an $m \times n$ matrix of ones
<code>zeros<fp>(int m, int n)</code>	Creates an $m \times n$ matrix of zeros
<code>eye<fp>(int n)</code>	Creates an $n \times n$ identity matrix
<code>diag(matrix<fp> V)</code>	Creates a diagonal matrix from a vector
<code>simplify(matrix< matrix<fp> > BM)</code>	Creates a normal matrix from a block matrix
<code>simplify(matrix<fp> M)</code>	Returns the value of the only entry in the 1×1 matrix <code>M</code> . Throws an exception if <code>M</code> is not 1×1 .
<code>matrix_cast<fp_out,fp>(matrix<fp> A)</code>	Casts the matrix <code>A</code> to the type <code>matrix<fp_out></code> by casting each element from <code>fp</code> to <code>fp_out</code>

Table A.3: Special functions

Appendix B

Input Algorithms

```
1 template <class fp>
2 void lu( const matrix<fp>& A, matrix<fp>& L, matrix<fp>& U ) {
3     if( A.get_m() != A.get_n() ) {
4         throw matrix_not_square();
5     }
6     int N = A.get_m();
7     L.set_size(N,N);
8     U = A;
9     // Begin LU decomposition
10    for( int n = 0 ; n < N ; n++ ) {
11        // calculating nth column of L
12        // above diagonal
13        for( int lrow = 0 ; lrow < n ; lrow++ ) {
14            L(lrow,n) = 0;
15        }
16        // on diagonal
17        L(n,n) = 1;
18        // below diagonal
19        for( int lrow = n+1 ; lrow < N ; lrow++ ) { // lrow = n+1
20            L(lrow,n) = U(lrow,n) / U(n,n);
21        }
22        // Working on U
23        // update below nth row
24        // add 0's in the nth column
```

```

25     for( int urow = n+1 ; urow < N ; urow++ ) {
26         U(urow,n) = 0;
27     }
28     for( int urow = n+1 ; urow < N ; urow++ ) {
29         for( int ucol = n+1 ; ucol < N ; ucol++ ) {
30             U(urow, ucol) = U(urow, ucol) - L(urow,n)*U(n, ucol);
31         }
32     }
33 }
34 }
35
36 template <class fp>
37 void ldl( const matrix<fp>& A, matrix<fp>& L, matrix<fp>& D ) {
38     if( A.get_m() != A.get_n() ) {
39         throw matrix_not_square();
40     }
41     int N = A.get_m();
42     // Initialize L and D
43     L = eye<fp>(N);
44     D = zeros<fp>(N,N);
45     for( int i = 0 ; i < N ; i++ ) {
46         for( int j = 0 ; j < i ; j++ ) {
47             if( j == 0 ) {
48                 // sum_over_k = 0
49                 L(i,j) = A(i,j) / D(0,0);
50             } else {
51                 fp sum_over_k = L(i,0) * D(0,0) * L(j,0);
52                 for( int k = 1 ; k < j ; k++ ) {
53                     sum_over_k += L(i,k) * D(k,k) * L(j,k);
54                 }
55                 L(i,j) = (A(i,j) - sum_over_k) / D(j,j);
56             }
57         }
58         if( i == 0 ) {
59             // sum_over_k = 0
60             D(i,i) = A(i,i);

```



```

61     } else {
62         fp sum_over_k = L(i,0) * L(i,0) * D(0,0);
63         for( int k = 1 ; k < i ; k++ ) {
64             sum_over_k += L(i,k) * L(i,k) * D(k,k);
65         }
66         D(i,i) = A(i,i) - sum_over_k;
67     }
68 }
69 }
70
71 // Hand optimized ldl decomposition
72 template <class fp>
73 void ldl_optimized( const matrix<fp>& A, matrix<fp>& L, matrix<fp>& D )
74 {
75     if( A.get_m() != A.get_n() ) {
76         throw matrix_not_square();
77     }
78     int N = A.get_m();
79     // Initialize L and D
80     L = eye<fp>(N);
81     D = zeros<fp>(N,N);
82     // Intermediate results that are reused to reduce the length of the
83     // critical path
84     matrix<fp> Lij_times_Djj(N,N);
85     Lij_times_Djj = zeros<fp>(N,N);
86     for( int i = 0 ; i < N ; i++ ) {
87         for( int j = 0 ; j < i ; j++ ) {
88             if( j == 0 ) {
89                 Lij_times_Djj(i,j) = A(i,j);
90                 L(i,j) = Lij_times_Djj(i,j) / D(0,0);
91             } else {
92                 fp sum_over_k = Lij_times_Djj(i,0) * L(j,0); // L(j,0)
93                 // is calculated earlier than L(i,0)
94                 for( int k = 1 ; k < j ; k++ ) {
95                     sum_over_k += Lij_times_Djj(i,k) * L(j,k); // L(j,k)
96                     // is calculated earlier than L(i,k)

```

```

93         }
94         Lij_times_Djj(i, j) = A(i, j) - sum_over_k;
95         L(i, j) = Lij_times_Djj(i, j) / D(j, j);
96     }
97 }
98 if( i == 0 ) {
99     D(i, i) = A(i, i);
100 } else {
101     fp sum_over_k = L(i, 0) * Lij_times_Djj(i, 0);
102     for( int k = 1 ; k < i ; k++ ) {
103         sum_over_k += L(i, k) * Lij_times_Djj(i, k);
104     }
105     D(i, i) = A(i, i) - sum_over_k;
106 }
107 }
108 }
109
110 template <class fp>
111 void cholesky( const matrix<fp>& A, matrix<fp>& L ) {
112     int i, j, k;
113     fp sum;
114     int N = A.get_n();
115     L = zeros<fp>(N,N);
116     for( i = 0 ; i < N ; i++ ) {
117         for( j = i ; j < N ; j++ ) {
118             sum = A(i, j);
119             for( k = i-1 ; k >= 0 ; k-- ) {
120                 sum -= L(i, k) * L(j, k);
121             }
122             if( i == j ) {
123                 L(i, i) = sqrt(sum);
124             } else {
125                 L(j, i) = sum/L(i, i);
126             }
127         }
128     }

```

```

129 }
130
131 // This function creates the matrix needed by dft()
132 template <class fp>
133 matrix< complex<fp> > dft_matrix( int N ) {
134     complex<fp> *w = new complex<fp>[N];
135     matrix< complex<fp> > W(N,N);
136     // Find Nth roots of unity
137     for( int i = 0 ; i < N ; i++ ) {
138         complex<double> tmp = polar<double>( 1, -2 * PI * i / (double)N
139             ) / sqrt(N); // division by sqrt(N) makes transform unitary
140         fp fp_real = tmp.real();
141         fp fp_imag = tmp.imag();
142         w[i] = complex<fp>( fp_real , fp_imag );
143     }
144     // Fill dft matrix with Nth roots of unity
145     for( int k = 0 ; k < N ; k++ ) {
146         for( int n = 0 ; n < N ; n++ ) {
147             W(k,n) = w[ (k * n) % N ];
148         }
149     }
150     delete [] w;
151     return W;
152 }
153
154 template <class fp>
155 matrix< complex<fp> > dft( const matrix< complex<fp> >& x ) {
156     // Check Sizes
157     int N = x.get_n();
158     int M = x.get_m();
159     if( M != 1 && N != 1 ) {
160         throw matrix_not_vector();
161     }
162     if( M != 1 ) {
163         return dft_matrix<fp>( M ) * x;

```

```

164     } else {
165         return transpose(dft_matrix<fp>( N ) * transpose(x));
166     }
167 }
168
169 template <class fp>
170 matrix< complex<fp> > fft( const matrix< complex<fp> >& x ) {
171     // Inspired by Numerical Recipes in C++
172     // Check Sizes
173     int N = x.get_n();
174     int M = x.get_m();
175     if( M != 1 && N != 1 ) {
176         throw matrix_not_vector();
177     } else if( M == 1 && N != 1 ) {
178         return transpose( fft_raw( transpose(x) ) );
179     }
180
181     int i, j, logN;
182     matrix< complex<fp> > x_br(M,N); // x bit-reversed
183
184     // N (N dimension of matrix) == 1
185     // x is a vertical vector with entries x(i,0)
186     // Make N the dimension of the vector now for simplicity
187     N = M;
188
189     // Calculate log_2( N )
190     // if N == 1, logN = 0
191     // if N == 2, logN = 1
192     logN = 0;
193     for( i = 1 ; i < N ; i = i << 1 ) {
194         logN++;
195     }
196     // Now (1 << logN) == N
197
198     // Bit Reversal
199     // index = 001011 -> index = 110100

```

```

200 for( int i = 0 ; i < N ; i++ ) {
201     // calculate bit reverse of i
202     j = 0;
203     for( int bit_i = 0 ; bit_i < logN ; bit_i++ ) {
204         if( ((1 << bit_i) & i) != 0) {
205             j |= (1 << (logN - 1 - bit_i));
206         }
207     }
208     x_br( i, 0 ) = x( j , 0 );
209 }
210
211 // Performing FFT
212 // Does calculations inplace in x_br
213 double pi = 3.14159265358979323846;
214 for( int step = 1 ; step < N ; step <= 1 ) {
215     // Known compile time constants
216     int jump = step << 1;
217     double delta = -pi / double(step);
218     double sine = sin(delta * .5);
219     complex<double> multiplier(-2.0 * sine * sine , sin(delta));
220     complex<double> factor(1.0, 0.0);
221
222     for( int group = 0 ; group < step ; group++ ) {
223         for( int pair = group ; pair < N ; pair += jump ) {
224             int match = pair + step;
225
226             // factor is a compile time constant
227             // here it is converted to the template data type
228             complex<fp> fp_factor = factor;
229             complex<fp> product = fp_factor * x_br(match,0);
230
231             x_br(match,0) = x_br(pair,0) - product;
232             x_br(pair,0) += product;
233         }
234         factor = multiplier * factor + factor;
235     }

```

```
236     }  
237     return x_br;  
238 }
```

appendix/src/examples.hpp

Appendix C

graphMaker.hpp

```
1 /**
2  * \file graphMaker.hpp
3  * \brief This file holds the description of the graphMaker class
4  *
5  * \author Andy Wright
6  * \date May 22, 2012
7  */
8
9 #ifndef GRAPHMAKER_HPP
10 #define GRAPHMAKER_HPP
11
12 #include "schedNode.hpp"
13 #include "schedGraph.hpp"
14
15 /**
16  * \brief This class creates a dependencyGraph through overloaded
17  * operations including +,-,*, and /
18  *
19  * An example of creating a dependencyGraph using this class can be
20  * seen below:
21  * \code{.cpp}
22  * graphMaker a,b,c,d,tmp1,tmp2,sum;
23  * tmp1 = a * b;
24  * tmp2 = c + d;
```

```

23 *      sum = tmp1 + tmp2;
24 *  \endcode
25 */
26 class graphMaker {
27     public:
28         graphMaker();
29         graphMaker(double constant);
30         graphMaker(int constant);
31         graphMaker(string nodeName_req);
32         graphMaker(const graphMaker& x);
33         ~graphMaker();
34
35         graphMaker operator+=(const graphMaker &rhs);
36         graphMaker operator-=(const graphMaker &rhs);
37         graphMaker operator*=(const graphMaker &rhs);
38         graphMaker operator/=(const graphMaker &rhs);
39         graphMaker operator=(const graphMaker rhs);
40
41         void renameNode(string name_req);
42
43         static void newGraph();
44         static schedGraph *graph;
45
46         friend graphMaker operator+(const graphMaker &opA,
47                                     const graphMaker &opB);
48         friend graphMaker operator-(const graphMaker &opA,
49                                     const graphMaker &opB);
50         friend graphMaker operator-(const graphMaker &opB);
51         friend graphMaker operator*(const graphMaker &opA,
52                                     const graphMaker &opB);
53         friend graphMaker operator/(const graphMaker &opA,
54                                     const graphMaker &opB);
55         friend graphMaker sqrt(const graphMaker &opA);
56         friend graphMaker min( const graphMaker &opA,
57                                const graphMaker &opB);
58         friend graphMaker max( const graphMaker &opA,

```



```

59         const graphMaker &opB);
60     friend graphMaker cond( const graphMaker &opA,
61                             const graphMaker &opB,
62                             const graphMaker &opC);
63     friend graphMaker eq(   const graphMaker &opA,
64                             const graphMaker &opB);
65     friend graphMaker neq(  const graphMaker &opA,
66                             const graphMaker &opB);
67     friend graphMaker lt(   const graphMaker &opA,
68                             const graphMaker &opB);
69     friend graphMaker lteq( const graphMaker &opA,
70                             const graphMaker &opB);
71     friend graphMaker gt(   const graphMaker &opA,
72                             const graphMaker &opB);
73     friend graphMaker gteq( const graphMaker &opA,
74                             const graphMaker &opB);
75
76     friend bool isZero( const graphMaker &x );
77     friend bool isOne( const graphMaker &x );
78     friend bool isTrue( const graphMaker &x );
79     friend bool isFalse( const graphMaker &x );
80
81     private:
82         void ensureValidNode() const;
83         bool isConstant() const;
84         float getConstantVal() const;
85
86         /// The pointer to the last calcNode this class represented
87         schedNode *node;
88         schedGraph *myGraph;
89     };
90
91 #endif // GRAPHMAKER_HPP

```

appendix/src/graphMaker.hpp

Bibliography

- [1] *LogiCORE IP Floating-Point Operator v5.0*. Xilinx, Inc., 2011.
- [2] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, April 2012.
- [3] *Lexical Analysis With Flex, for Flex 2.5.37*. The Flex Project, 2012.
- [4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [5] Koldo Basterretxea and Khaled Benkrid. Embedded high-speed Model Predictive Controller on a FPGA. *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 327–335, June 2011.
- [6] Philippe Coussy and Adam Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.
- [7] Charles Donnelly and Richard Stallman. *GNU Bison - The Yacc-compatible Parser Generator*. Free Software Foundation, Boston, MA, USA, 2012.
- [8] Yong Dou, Jie Zhou, and Xiaoyang Chen. FPGA accelerating three QR decomposition algorithms in the unified pipelined framework. *Field Programmable Logic . . .*, 410073:410–416, 2009.
- [9] Richard M. Stallman et al. *Using the GNU Compiler Collection*. Free Software Foundation, Boston, MA, USA, 2012.
- [10] William H. Press et al. *Numerical recipes in C++: the art of scientific computing*. Cambridge University Press, Cambridge, UK, second edition, 2002.
- [11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
- [12] Tor A Johansen, Warren Jackson, Robert Schreiber, and Petter Tøndel. Hardware Architecture Design for Explicit Model Predictive Control. (5):1924–1929, 2006.

- [13] Kumud Nepal, Omur Ulusel, R. Iris Bahar, and Sherief Reda. Fast Multi-Objective Algorithmic Design Co-Exploration for FPGA-based Accelerators. *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 65–68, April 2012.
- [14] Yasser Shoukry, M.W. El-Kharashi, and Sherif Hammad. MPC-On-Chip: An Embedded GPC Coprocessor for Automotive Active Suspension Systems. *Embedded Systems Letters, IEEE*, 2(2):31–34, June 2010.
- [15] Ranko Sredojevic, Andrew Wright, and Vladimir Stojanovic. Hardware-Software Codesign for Embedded Numerical Accelerators. *2013 IEEE 21st International Symposium on Field-Programmable Custom Computing Machines*, April 2013.
- [16] G. W. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics, 1998.