Template-based Hardware-Software Codesign for High-performance Embedded Numerical Accelerators

by

Ranko Radovin Sredojević

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 30, 2013

Certified by .....
Vladimir Stojanovic
Associate Professor
Thesis Supervisor

Certified by .....
Alexandre Megretski
Professor of Electrical Engineering
Thesis Supervisor

Certified by .....
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Template-based Hardware-Software Codesign for High-performance Embedded Numerical Accelerators

by

Ranko Radovin Sredojević

Submitted to the Department of Electrical Engineering and Computer Science
on September 2, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Sophisticated algorithms for control, state estimation and equalization have tremendous potential to improve performance and create new capabilities in embedded and mobile systems.

Traditional implementation approaches are not well suited for porting these algorithmic solutions into practical implementations within embedded system constraints. Most of the technical challenges arise from design approach that manipulates only one level in the design stack, thus being forced to conform to constraints imposed by other levels without question. In tightly constrained environments, like embedded and mobile systems, such approaches have a hard time efficiently delivering and delivering efficiency.

In this work we offer a solution that cuts through all the design stack layers. We build flexible structures at the hardware, software and algorithm level, and approach the solution through design space exploration. To do this efficiently we use a template-based hardware-software development flow.

The main incentive for template use is, as in software development, to relax the generality vs. efficiency/performance type tradeoffs that appear in solutions striving to achieve run-time flexibility. As a form of static polymorphism, templates typically incur very little performance overhead once the design is instantiated, thus offering the possibility to defer many design decisions until later stages when more is known about the overall system design.

However, simply including templates into design flow is not sufficient to result in benefits greater than some level of code reuse. In our work we propose using templates as *flexible interfaces* between various levels in the design stack. As such, template parameters become the common language that designers at different levels of design hierarchy can use to succinctly express their assumptions and ideas. Thus, it is of great benefit if template parameters map directly and intuitively into models at every level.

To showcase the approach we implement a numerical accelerator for embedded Model Predictive Control (MPC) algorithm. While most of this work and design flow are quite general, their full power is realized in search for good solutions to a specific problem. This is best understood in direct comparison with recent works on embedded and high-speed MPC implementations. The controllers we generate outperform published works by a handsome margin in both speed and power consumption, while taking very little time to generate.

Thesis Supervisor: Vladimir Stojanovic

Title: Associate Professor

Thesis Supervisor: Alexandre Megretski
Title: Professor of Electrical Engineering

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

To my Parents,

for your gift of Life.

To my Teachers,

for illuminating my Way.

To my Friends,

for the Time you have given me.

To my Muses,

for Passions and Love.

Acknowledgments

The time I have spent at MIT was an amazing journey full of interesting people and experiences. It transformed me both as a person and as an engineer.

First and foremost, I would like to thank my advisor, Prof. Vladimir Stojanović. I could not have been luckier to have him as my mentor. He was my "negative feedback" when I needed to focus on the problem at hand and my "positive feedback" when I was in the homestretch. He let me explore many areas of engineering research, and the fact that I finally found my true engineering domain is due to him.

I would not have been able to undertake this journey without the support and encouragement of my undergraduate mentors. Above all, I wish to thank Prof. Aleksandra Pavasović, who inspired me to pursue a graduate degree and believe in myself. She saw more in me than even I did at the time, and I am infinitely grateful for her support and advice. I am grateful to Prof. Jelena Popović and the late Prof. Slavoljub Marjanović, who always had words of encouragement for me.

I am lucky to have many great friends, on both sides of the Atlantic and the Equator - too many to single out without forgetting someone. Thank you for all the support in good times, and advice in bad times. You bring out the best in me, and help me control my worst. I always look forward to spending more time with all of you.

Finally, I cannot give enough thanks to my mother Stanica, my father Radovin and my sister Vesna. Without you none of this would have been possible. No matter how far apart we might be, thinking of you makes me happy and helps me endure.

Contents

List of Figures

List of Tables

List of Acronyms

QP Quadratic Programming

MPC Model Predictive Control

UAV Unmanned Aerial Vehicle

RF Radio Frequency

WSN Wireless Sensor Network

μC Microcontroller

DFG Data Flow Graph

DSL Domain Specific Language

ASIC Application Specific Integrated Circuit

FPGA Field Programmable Gate Array

KKT Karush-Kuhn-Tucker

IP Interior Point

LTI Linear Time Invariant

DSP Digital Signal Processing

SS State Space

SSB System Side Bus

FSB Front Side Bus

BSV Bluespec System Verilog

HDL Hardware Definition Language

HLS High Level Synthesis

NOP No Operation Instruction

USB Universal Serial Bus

UART Universal Asynchronous Receiver/Transmitter

ADC Analog to Digital Converter

DAC Digital to Analog Converter

RAM Random Access Memory

PLL Phase-Locked Loop

GCC The GNU Compiler Collection

ICC Intel C Compiler

IO Input/Output

Chapter 1

Introduction

For it is unworthy of excellent men to lose hours like slaves
in the labor of calculation which could safely be relegated
to anyone else if machines were used.
—Gottfried Wilhelm von Leibniz

We should forget about small efficiencies, say about 97% of
the time: premature optimization is the root of all evil.
—Donald Knuth

Success of a technology is only partially determined by the promise of the underlying scientific principles. For widespread adoption a solution must also be time and cost effective. This usually requires a wide range of applications, fast development and reliable maintenance cycle. Until recently, advanced signal processing and control were considered to be out of the reach of embedded and mobile platforms due to their modest computing capabilities.

In the last decade, implementations of sophisticated signal conditioning algorithms for use in embedded systems have been under active development. However, most of the solutions reported to date are based on ad-hoc considerations and specific treatment of a problem instance. Few that consider building infrastructure for efficient implementation are usually heavily dependent on existing hardware/software infrastructure. This is not enough to raise advanced Digital Signal Processing (DSP) to the level of technology in these spaces. While the results are quite promising and the possible improvements in performance appealing, the unstructured design flow is hard to predict, schedule and use reliably.

Showing that advanced signal processing and control are viable and reliable solutions

for improving performance of embedded and mobile systems involves addressing most of the risk factors. Ad-hoc approach to implementation should be replaced by a general framework for accelerator construction to remove uncertainties of unstructured design flow. At the same time, it should be demonstrated that the resulting approach yields accelerators with competitive performance, thus confirming that the new set of applications is within reach. This can be done by considering and tightly connecting all layers of embedded design hierarchy - from hardware design through software implementation up to algorithm exploration. In such a flow, if executed well, a small team of designers can own all the design decisions and make sure that system components fit together and perform under desired conditions.

1.1 Embedded Digital Signal Processing and Control

Sophisticated digital signal processing and control have tremendous potential to improve performance of many systems in robotics, mobile and general embedded markets [1–5]. However, the applicability of these sophisticated techniques was traditionally considered to be quite limited; their deployment was considered only in systems where powerful workstations can be used for implementation and sampling rates are very slow. The reason for this is difficulty in achieving efficient implementations of these demanding computational loads under the resource constraints of embedded environments [6, 7]. This generally disqualifies them in embedded systems with standard computation resources, leaving them with simple signal processing like digital filtering and algorithms with low-cost recursive update.

In this work we are interested in re-evaluating these conclusions. We focus on embedded systems, usually constrained in area, weight and power consumption (e.g. mobile and battery powered devices). Many such designs would benefit greatly if more sophisticated algorithmic solutions could be used within the resource constraints. The applications we consider require known, and fixed, input-to-output latency (e.g. discrete time communication and feedback control systems). In other words, we consider systems where worst case latency is of interest for the correct operation. Typically, in such designs, queueing, flexible pipelines and other techniques for average processing rate increase are not of much benefit.

For example, many prototypes of Unmanned Aerial Vehicle (UAV) and general robotics are reported with off-board control and state estimation using an indoor visual tracking

system and a dedicated control server [5, 8, 9]. While they are showing very promising results, such an approach poses significant challenges in scaling past the proof-of-concept stage. Off-board state estimation and control imply significant round-trip delays during Radio Frequency (RF) communication with the controller setup [9]. Consequently, the distance between the unmanned robot and the control setup must be kept small, limiting the possible applications. Long sensor to control delays also results in poorer performance under disturbance and limit some of these techniques to indoor use or to proof-of-concept systems. Achieving high performance on-board control under the speed, area, weight and power constraints of these systems would enable construction of more agile systems capable of properly functioning in a wider range of situations [5]. This is one of the reasons research in controller implementations suitable for on-board operation [4] is receiving more attention lately.

Similar tradeoffs can be observed in Wireless Sensor Network (WSN) settings. Analysis of power tradeoffs shows that it is preferable to avoid RF communication from a WSN node because of the disproportionately high power cost of RF communication [10] relative to the cost of signal acquisition and computation. It is always preferable to do as much computation as possible on the node before passing data further into the network [10]. Furthermore, overall functionality and performance of WSN is very dependant on communication delays [11] and powerful node controllers could trade off communication for computation. In that sense having high-performance numerical accelerators with low power and area footprint would be quite beneficial for WSN applications as well.

Embedded processor resources and instruction sets evolved targeting linear digital filtering and data acquisition operations. As such, they cannot efficiently run more sophisticated classes of control and signal processing algorithms, which are based on optimization solvers [6]. Furthermore, it is hard to optimize processor resources for low latency without knowing the program. Thus throughput performance is usually considered as a surrogate latency, even though it is well known that they do not coincide [12].

On the other hand, general purpose desktop and server processors usually consume orders of magnitude more power than what is available in mobile, robotic and embedded systems in general. Unsurprisingly, achievable average performance can be quite good if a general purpose processor can be used [6]. However, ensuring real-time algorithms always execute properly on machines with out-of-order execution, complicated cache hierarchy and

other speculative attempts at throughput optimization is quite difficult and sometimes even impossible [12–14]. A statically scheduled system with straightforward execution timing would be more appropriate for all applications mentioned, especially time-critical ones since fast computing is not real-time computing [12].

Poor latency of embedded processors and high power of high-performance processors, coupled with advances in numerical and signal processing algorithms [6, 15] give rise to research in custom offload engines [6, 7, 15–26]. Usually, these solutions rely on extreme customization, hand-crafted code and optimizations based on thorough examination of the algorithm structure, reducing the generality, impeding design reuse and increasing the engineering cost and time to market. Few works recognize the need for establishing proper infrastructure and not relying on guesswork and ad-hoc methods in the design process [6,15,21,24], but not all of them are suited for embedded and low power use simply because of the tools and processor power they need to operate efficiently [6].

1.1.1 Traditional Development Flows

In general, two major paths to custom accelerator construction can be seen in literature: *algorithm-first* or *platform-first*.

The algorithm-first route assumes that the top-level algorithmic choice can be made before the system implementation begins. A sketch flow-chart of this design approach is shown in Figure 1-1. The algorithm is analyzed and partitioned into smaller functional units, usually defining the major modules for system implementation. The inter-module synchronization and data handoff protocols are also defined at this point, thus fixing the algorithm execution timing to a large degree. As each module is implemented the first performance measures start emerging, but only estimates and bounds to the system performance can be given until the system is fully integrated. Changing any of the system-level design decisions (e.g. algorithm, algorithm partitioning) requires going through the full design cycle again, at the expense of human designer time. An example design following this methodology can be found in [17].

Usually, algorithm-first designs produce hardware-based solutions. In other words, most of the functionality is achieved by mapping it directly into hardware.

Often, system and algorithm designers have little information about implementation tradeoffs and feasible design space before the implementation starts. Algorithms can have

many equivalent formulations and the choice might not be obvious, making the laborious and long implementation a risky endeavor and possibly a trial-and-error route to an acceptable solution.

Ideally, the algorithm-first route should offer great flexibility in defining and fine-tuning the system. This intuition comes from the belief that full knowledge of the algorithm, coupled with full control of every aspect of the implementation, can help designers make better decisions. Unfortunately, the serialized development cycle, Figure 1-1, makes feedback in design decision making very expensive in terms of time and design effort. Furthermore, in every iteration we are working only with a particular algorithm instance and implementing one accelerator instance. Observing general tradeoffs is quite challenging in this setting, and carrying tradeoff knowledge across redesign spins is not easy because every implementation attempt tends to be very specific.

Despite its initial promise, this approach usually yields quite rigid design descriptions, seldom capable of any algorithmic tradeoffs. Furthermore, without the abstract system model to unify common ideas between different implementations the design is easily lost in the low-level implementation clutter.

The platform-first route starts with hardware component design or choice as shown in Figure 1-2. Following is a process similar to algorithm-first approach, but it happens entirely on the software side. In this way, a mostly software-based solution is created. Examples of such designs are [6, 7, 26].

The platform is optimized according to results of profiling of expected programming loads. The software infrastructure (e.g. OS, libraries, etc.) is constructed next. Finally, the user defined application is implemented. Inevitably, decisions made with lack of full algorithm specification constrain the algorithm design space, resulting in loss of flexibility for the algorithm designer. Furthermore, some standard processor and compiler optimization (e.g. out of order execution, pipeline flushing, instruction reordering) strategies make verifying real-time constraints very hard [13, 14].

The platform-first route enables high level of effort reuse because the whole processor/library stack can be thought of as reusable component. However, the reusable components are created without much regard for the final application. This often brings more constraints than necessary and limits the achievable performance. The extra weight of solutions achieved through this approach comes from the necessity to design a platform that

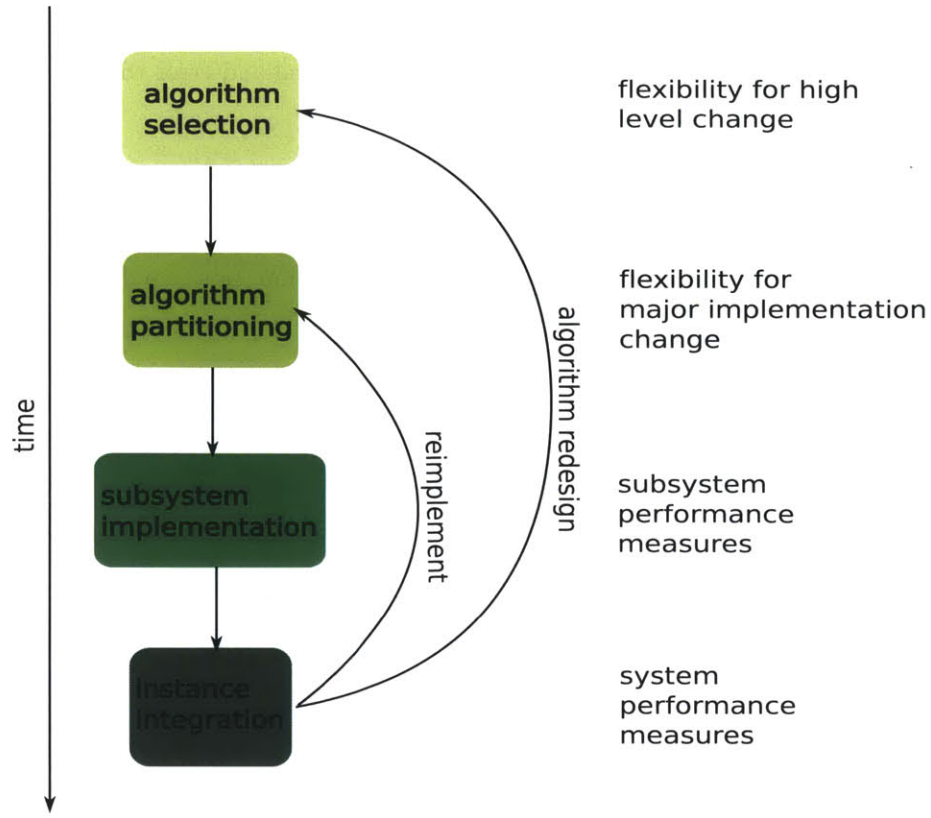


Figure 1-1: Typical algorithm-first design cycle

can work in many different scenarios, most of which are unknown or insufficiently described at the platform design time. In a sense, it resembles dynamic (run-time) polymorphism in programming. A generic support infrastructure that can express any solution in a given set is built and then programmed to perform one specific task. However, the overhead cost of unused components can be quite high.

A mixed approach is sometimes utilized [16]. In such framework part of the solution is implemented in platform-first and part in algorithm-first fashion. Usually, control and top level sequencing is implemented platform-first by using an embedded Microcontroller (μC). Critical computation sections are then moved to a custom computation engine. This approach is mostly an implementation convenience. In terms of flexibility and code reuse it is the intersection, not the union, of previous two flows; it suffers from poor effort reuse on the custom side, just like most of algorithm-first designs; on the platform-first part of the design, the reuse potential is quite limited as changes in the sequencing or data handoff easily propagate into the custom part of the design. It gives the worst of both worlds in

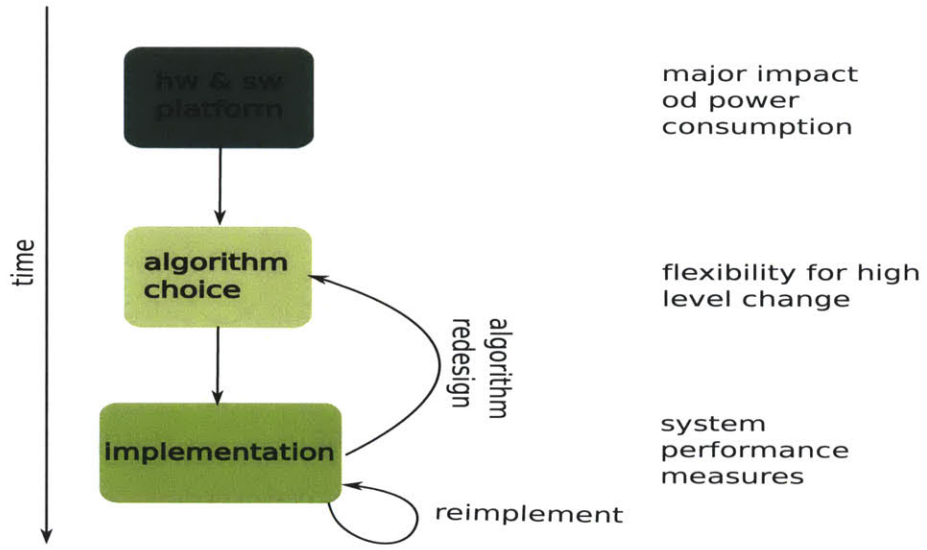


Figure 1-2: Platform-first design cycle

terms of design flexibility, trading it off for implementation convenience.

All these traditional flows suffer from the same problem: they manage design complexity by aggressively locking design decisions along the way. The algorithm-first approach starts by fixing the algorithm; the platform-first locks hardware/software infrastructure right at the beginning of the design. The result is a very quick loss of flexibility as the design progresses. Either approach is good if some aspects of the design performance can be sacrificed. In high performance embedded signal processing and control this is not feasible.

1.1.2 Template-based Hardware-Software Codesign

An ideal design flow would preserve the best of both worlds, offer the flexibility of the algorithm-first approach while achieving the high level of design reuse similar to platform-first design.

The need for improved design strategy stems from the observation that problems with both main design flows in literature arise from aggressive decision making at the early stages of the design, when the design tradeoffs are largely unknown and unpredictable. Our main motive is to follow a different route and only fix the minimal number of design decisions to make progress towards completing the system. As much of the design as possible, at each design level, should be left unspecified and presented at the design layer interface as parameters. This allows parallel development at every design layer. Furthermore, such design

approach simplifies and focuses reimplementation loops. It replaces the highly serialized development flows of Figures 1-1 and 1-2 for the parallelized and localized design flow in Figure 1-3.

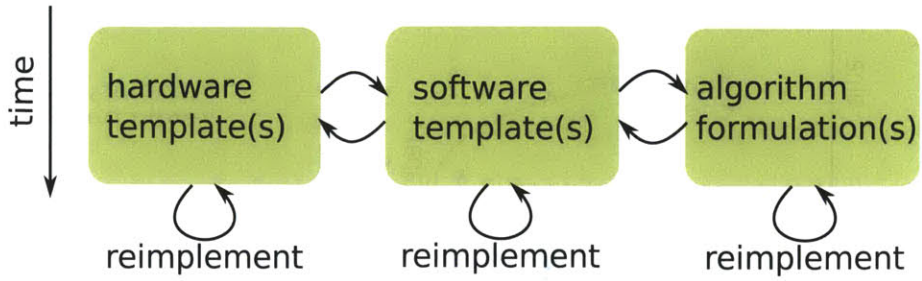


Figure 1-3: Template-based design cycle

By keeping this simple idea in mind we are working towards a class design. The goal is a description of an area of the design space of interest, not a particular point in that area. This lets us prepare the low-level implementation details beforehand, in a generalized form. Relying on generalized descriptions of whole design space regions, we should be able to achieve the reuse levels similar to those in platform-first approach. The difference is in the fact that our design is not a run-time flexible system, but a compile-time flexible system description keeping the performance penalty acceptable as well.

To produce the desired design , we need only to determine the desired instantiation parameters. This can be done efficiently through design space exploration mostly at the expense of machine and not human time. Trading off human work at early stages (pre-instantiation) of the design for, mostly, machine time in latter (post-instantiation) stages is why this step can be postponed. In other words, being able to synthesize instances, from the generalized form, quickly and cheaply (in terms of human work/time invested) allows us to postpone the synthesis step and spend more design time determining the instantiation parameters.

In such a design flow we are not forced to make algorithmic decisions early, restraining the design within an unknown region of the design space without being able to exactly describe it. With the infrastructure in place, and most groundwork already done, we are free to explore various algorithmic solutions and equivalent algorithm forms taking into account the cross-layer interactions without the clutter of low-level implementation details. As we will see, this allows us to tweak the design and achieve the level of performance and

flexibility present in algorithm-first design approach.

Achieving this ideal design flow is not currently feasible, but we can come reasonably close. Most of the desired properties of the generalized system design can be achieved through metaprogramming techniques. Similar reasons lead to macro and template system developments in software design. Thus, it is natural to extend these well tested techniques by applying them at all the levels of the design hierarchy, from top level algorithms down to software and hardware implementations. However, even metaprogramming facilities of high-level languages have their limitations. We will have to concede to making some of the design decisions outside of the flexibility of the proposed framework like other design paradigms do. Such decisions, as we will see, are mostly in aspects of the design that are hard to parameterize or describe within the metaprogramming facilities of the language we use (e.g. topological properties of the processor architecture). However, we strive to minimize such design decisions, as they introduce all the undesirable effects we seek to avoid.

We note that a design flow with seemingly similar characteristics could be developed within the context of High Level Synthesis (HLS). Detailed analysis of the proposed flow in this thesis will clarify the difference and how HLS and the proposed flow address different shortcomings of the traditional design approaches. HLS is concerned, mostly, with hardware design and not on hardware-software codesign. To achieve the flexibility we desire, the code for HLS would have to be written in a template form, anyways. Finally, HLS puts too much pressure on the input language and the compiler [27]. Current compiler technology is not advanced enough to provide provably optimal manipulation of the computation structure described by the input code [28]. Thus, it is easy to end up in a situation where design and debug are quite tedious as small code changes result in dramatic, discontinuous performance change after HLS tool is applied. We offer a more general and more controlled alternative. Every design is explicitly controlled at multiple levels by insisting on human-designed, and human-readable, descriptions.

Recently, similar ideas, under the term *chip generators*, started appearing in research relating to the design productivity and cost of Application Specific Integrated Circuit (ASIC) designs [29–31]. Our work has similar scope, and while we prototype and evaluate on Field Programmable Gate Array (FPGA) there is nothing precluding us from using the same methodology in ASIC development. The main differences between our and similar works is

in the mechanics of how the flexible, generalized solution should be constructed, the level of control over design that we wish to retain for designers at all levels of hierarchy and our focus on latency-critical system design.

What we are proposing is the construction of Domain Specific Language (DSL) at each level of the design hierarchy. Unlike other works dealing with the design flows for ASIC and FPGA for embedded applications we embed our DSL in powerful, high-level languages. By doing so we enable incremental construction of the solution through the use of standard development practices. It is this process of discovering the desired properties of system templates at each level of the hierarchy that we find most valuable for the efficient design. Works in chip generators [?] did not focus on efficiency of template construction.

In this work we will guide the reader through the design process of one such template-based solution. We will describe the process of evolving the template structure at each level of the hierarchy through design space exploration.

1.1.3 . Example Design: Model Predictive Control

The main design example we use to show our design flow is a Model Predictive Control (MPC) control algorithm [2]. This choice was motivated by the high volume of control and system architecture publications dealing with efficient implementation of the algorithm in the past few years [6, 7, 16, 17, 19, 20, 22, 23, 32].

The algorithm is based on interior point solver for Quadratic Programming (QP) problems [33]. It is an iterative algorithm that requires solving a linear system of equations at each step followed by a one dimensional function optimization by line search. Due to this, it is considered as a computationally very demanding load and rarely used in embedded environments despite many advantages it offers in handling the system constraints explicitly [2, 6].

We do not advertise the MPC for any particular application. Our intention is to showcase a methodology for arriving to high performance numerical offloading engines in a consistent and reliable manner that enables design reuse. Implementation of the MPC control engine is to showcase our methodology on an algorithm considered challenging to implement in embedded environment.

1.2 Contributions

The contributions of this work are:

1. Definition of template-based hardware-software co-design flow for embedded computation offloading.

   Templates offer generality of the design while suffering minimal performance degradation of the resulting instances. Using them promotes design reuse and opens up designer time for analyzing cross-layer performance tradeoffs of the system. Utilizing them as flexible design layer interfaces provides excellent frame for discovering good solutions to a given problem through negotiations between designers at each level. We must make sure that template parameters have meaningful interpretations at both sides of the design boundary. One would be hard-pressed to build such a general framework without specifying the application sufficiently to be able to exploit the specific structure of the problem. That is why we focus on latency critical numerical algorithms. As it turns out, this narrows the desired implementation sufficiently for our methodology to achieve designs with superior performance to other implementation techniques.

2. Implementation of necessary infrastructure to enable such a design flow for embedded numerical acceleration.

   In our proposed design flow, the compiler becomes the main link between the algorithm and processor descripitons. It serves not only as a programming code generator, but a tool for processor and algorithm matching. It helps us determine desired instantiation parameters through analysis of Data Flow Graph (DFG) of the computation. Compiler maps hardware-level template parameters into meaningful properties of DFG manipulated at the algorithm formulation level and enables their visualization and analysis. Unlike a compiler for a predefined processor architecture, our compiler-like tool has flexible, extensible design enabling quick mockups of new unit types and testing of their influence on the DFG structure and scheduling. Finally, since DFG is available for inspection at multiple stages of the compilation process, we are always aware of the exact latency of the computation and operation count as a surrogate for latency is not necessary. Having this type of tool can significantly change the way

algorithm designers pick implementation candidates because guesswork based on operation count can be replaced by quick experiments on the compiler that will eventually produce the final hardware instance and programming files.

3. Implementations of several demanding, optimization-based, signal processing algorithms that outperform all similar systems reported to date.

   Leveraging the flexibility of the design flow and tools we constructed, we show how a class of demanding numerical applications can be accelerated on a custom hardware-software combination. We present accelerator design at all the levels of hierarchy: hardware, software and algorithm. Using the compiler as our exploration tool, we show how algorithms can be molded into equivalent forms that enable more parallel execution and better interaction with processor primitives, often requiring counter-intuitive decisions from the perspective of traditional design approach. For example, quick mockups of processor functional units help define exact functionality of our predication units, semantics of true and false for efficient implementation of line search in interior point algorithms. The results are control algorithm designs that outperform recently published solutions while achieving high degree of design reuse and flexibility for change at late stages of the design process.

Chapter 2

System Overview

> This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.
> —MORPHEUS, THE MATRIX

In this work we are looking to efficiently develop a flexible implementation of numerical computation engine. Our hardware-software codesign flow will be set up with that purpose in mind. In that sense the choice of template structures and their parameterization, i.e. the small part of design decisions that escape our design flow and the philosophy of lazy (or late) parameter definition, depends on the specific target algorithmic class of interest.

Our intention is to construct a numerical accelerator setup for QP optimization based control algorithms, e.g. MPC. We will use Interior Point (IP) class of algorithms for solving QP problems. These are iterative algorithms that utilize linear algebra kernels and simple decision making (i.e. conditional processing) in each iteration. In the interest of reusability and cost effectiveness of the solution we would like the design to be customizable. This can be done in multiple ways: at the design compile time or through programmable final design. Compile time flexibility, achievable through static polymorphism, is useful in custom tailoring area and some power consumption aspects of the design. This can come handy when we want to provide a maximally optimized processor size for a given size of MPC formulation. On the other hand, run-time programmability of the design can be useful if the same processor instance is to be used in various designs and applications. To showcase

the most general setup and to justify both FPGA and ASIC applications we will provide both options: the design will be customizable at compile time and programmable after implementation.

If some of these properties are not needed the design can be further customized for potential gains in performance, but we will not go into such an implementation. We will see that, despite the very general setup outlined here, proper design-space exploration driven high-level algorithmic choices can bring significant performance gains compared to hand-crafted designs where platform or algorithm was chosen before implementation details and tradeoffs are fully known.

2.1 Template-based Hardware-Software Codesign for Embedded Acceleration

Our design flow infrastructure for the previously defined target problem set is illustrated in Figure 2-1. We will follow general guidelines described in Section 1.1.2, including the structure of design flow sketched in Figure 1-3. Our design spans three co-dependent layers of the accelerator structure: hardware, software and algorithm.

2.1.1 Processor Template

As we show in the Chapter 6, real time latency of computation depends on the amount of parallelism that can be extracted from the algorithm for a particular processor configuration and maximum frequency. Thus, it is of utmost importance to configure the accelerator only after the intended load is known. This reconfigurability is provided by extensive parameterization of the hardware template, outlined in full yellow line in Figure 2-1, by the

- number of operation units of each type,
- number of data memories,
- pipeline latencies of each operation unit type,
- pipeline latencies of all transport layers.

Generalized hardware descriptions can be challenging since majority of Hardware Definition Language (HDL)s do not provide sufficient metaprogramming facilities to build a tem-

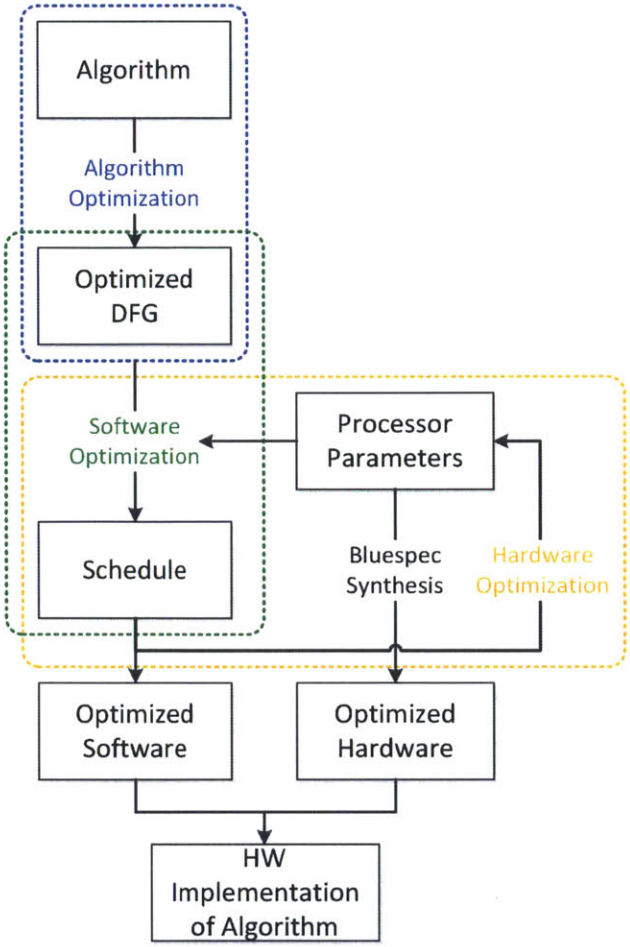


Figure 2-1: Full system design flow chart

plate design. For this reason and for development efficiency Bluespec System Verilog (BSV) was chosen as the implementation language at the hardware level. Careful template development in BSV can leverage its flexibility while mitigating the built-in overhead. Details of the processor template design are outlined in Chapter 3.

2.1.2 Statically Scheduling Compiler

Since the compiler, outlined in broken green line in Figure 2-1, 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. In our system, the compiler serves not only as programming code generator for the algorithm, but as a tool for determining instantiation parameters for the processor template through analyzing properties of algorithm DFG.

The program loads of interest are sets of algebraic expressions to be evaluated, as shown in Listing 2.1 example. Additionally, some forms of conditional or data dependent processing (e.g. min, max, conditional-expression operator) are supported.

The DFG in Figure 2-2 corresponds to the computation in Listing 2.1. Nodes of the graph represent operators and edges represent data flow.

```
1 x = (a + b) - (c + d);
2 y = f + g;
3 z = x * sqrt(e) / y;
```

Listing 2.1: Simple numerical computation

Our 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 and determine the best one for a given application. At the same time, the compiler can provide a range of diagnostic information like bottlenecks in the evaluation graph, the size of the DFG, the number of each type of operation, and the length of the critical path, as a feedback to both algorithm and processor template designers. To do this it uses the DFG of the algorithm and defines cycle-distances on it using the specified processor configuration. Then it attempts scheduling the graph for execution on the given processor, obtaining the schedule cycle-length. Finally, the processor post-implementation (i.e. post-place-and-route) cycle-time is used to calculate real time length of the schedule.

Detailed description of compiler architecture, implementation and interaction with other parts of the system is given in Chapter 4.

2.1.3 Algorithm Formulations

Design flow in Figure 1-3 puts the algorithm designer in charge of the majority of system decisions. Similar things can be done in more common design flows. The mechanism for achieving it, however, is very different in traditional design flows, and so are the results.

In more traditional implementation flows design phases have a serial chain type dependencies, Figures 1-1 and 1-2, forcing sequential decision making. Because of this, information describing tradeoffs in the design and interactions between design components is carried only through long redesign cycles.

In template-based flow we are looking to find a way to postpone decisions (that could be influenced by system parameters outside the template scope) if it is easy and efficient to do

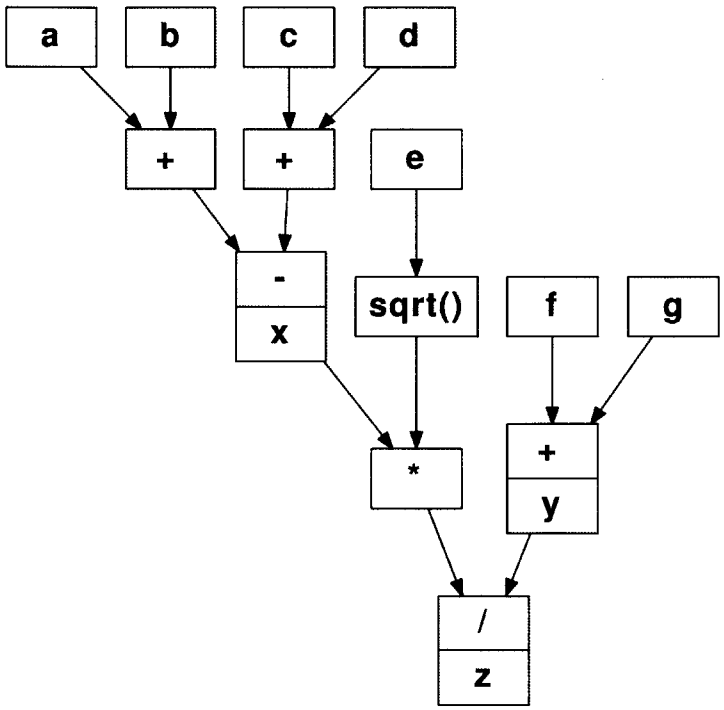


Figure 2-2: Simple data flow graph

so. This approach preserves much of the design flexibility until many of the decisions can be made jointly, at the modest expense of running a design configuration through the tool chain. As this process, essentially template instantiation, does not require much human time or work it is cheap and fast. Thus, low level implementation details of each template can be developed, upgraded or modified independently for the most part, and then checked by running a highly automated set of trials.

While the need for building templates restricts the set of applicable architectures or algorithms to ones that can be efficiently parameterized, the benefits of joint decision making at all levels of the design outweigh this downside. We will demonstrate this in Chapters 5 and 6.

Through design exploration, outlined in dotted blue line in Figure 2-1, we will demonstrate that there exist both the lower and the upper bound on the size of processor structure to execute a given DFG within a given time. Furthermore, we will see that manipulating, although counter-inttuitive, DFG of a computation into an equivalent form with higher number of operations can result in faster execution, in cases where we can afford to have adequate hardware resources.

These two results give particular examples demonstrating weakness of both algorithm-first and platform-first design flows. Coupled with the final measurement results demonstrating superior performance of achieved designs, when compared to recent published works, they completely justify investment in template-based development.

2.1.4 High Level Synthesis Comparison

Recently, various HLS tools have been advertised by major FPGA vendors. Nominally, these tools implement design flows with the same objective we declare in Section 1.1.2: equip the algorithm designer for cross-layer system performance optimization while facilitating design reuse.

However, non-speculative comparison between this work and HLS-based design flows is not possible. Despite HLS tools being available for number of years, very few publications report any results implementing high-performance numerical algorithms. At the time of this writing, no complex embedded controller design using HLS was described in literature, to the best of our knowledge.

Furthermore, the HLS approach usually assumes a high-level software development language (e.g. C, C++, System C) as the design language and aims to deliver full hardware implementation based on low-level hardware primitives. This approach is sometimes referred to as *C-to-gates*.

This is too aggressive from two important design cycle aspects.

In the first place, most languages for software design are not well suited for describing hardware operation [27]. Thus, fine tuning of hardware primitives and their inference from the high-level code is not an option in HLS flows. This problem is emphasized by the fact that most optimizing compilers only optimize through application of heuristic rules [28]. This obscures design process and makes it hard for designers to predict the output code performance change given the change in input code. Hardware synthesis compilers with randomized place-and-route algorithms come to mind: it is not unusual for a strictly simpler design code input to result in poorer performance, often making performance-driven redesign loops very challenging.

Closely related is the problem of efficient testing and debug in HLS flows. Input and output code are separated by multiple levels of indirection and transformation, inside the compiler, and usually not accessible to the designer. They short-circuit all the intermediate

design levels we show in Figure 2-1, going from the input directly to the low-level hardware representation. This makes tracking the changes and relating input and output design representation very complicated, sometimes even impossible.

In sharp contrast to the standard HLS approach stands our implementation of template based flow. We insist on multiple human-made (and thus human-readable) intermediate representations of the design, shown in Figure 2-1. This makes tracking changes, system-wide interactions and performance impacts of changes easier and more understandable. Every template, with its parameters and interfaces serves as a point where a designer can exercise full control of the design at a certain level.

2.2 Summary

While the first chapter outlined the motivation for development of an alternative design flow, this chapter gives an overview of the structure of the proposed solution. We briefly introduce major sections of the thesis, aligned with the logical partitioning of the designed accelerators. Finally, we discuss the differences between the design flow we created and the design with HLS-based flows.

Chapter 3

Processor Template

So, you'd like to make progress, but also at the same time
never be bound by the consequences of your decisions.
Data abstraction is one way of doing this.
—HAL ABELSON

More computing sins are committed in the name of
efficiency (without necessarily achieving it) than for any
other single reason - including blind stupidity.
—WILLIAM WULF

Choosing the computation structure for accelerator construction is not a straightforward process, but it must be done in some sense for any progress towards design completion to be made. This is one of the decisions that cannot be fully deferred, no matter which of the design flows is used. Even in HLS flows the achievable design set is fully determined by the set of primitives the compiler can infer from the high-level language.

In the case of algorithm-first design flow, the decision is usually based on intuitive, and mostly speculative, algorithm partitioning as in [16,17]. Platform-first designs usually settle for an off-the-shelf general-purpose processor, as in [3], or construct a specialized processor after profiling a particular algorithm implementation and partitioning, the approach used in [7].

Technically, the flow in Figure 2-1 could be used to postpone hardware-level development by first spending time to study properties of the DFGs of interest. This route has potential and will be briefly discussed in Chapter 7. However, it would require significant effort in DFG analysis to find general primitives of interest for our computation that do not, at the

same time, reduce generality or presume, even implicitly, a certain algorithm partitioning or coarse-grain schedule of operations. As we are developing a proof of concept system we will not pursue this.

It will become clear in Chapter 6 that current solutions offered in literature can be overtaken with the simplest processor template imaginable. We will proceed assuming that processor units perform simple algebraic operators or basic predication. Each unit has only one function. In other words, we will treat DFG of computation at its most fine-grained form, essentially accounting for every algebraic operation separately.

3.1 Processor Template

The main tool for accelerator construction is exploiting parallelism present in the algorithm of interest. Uncovering parallelization opportunities is crucial for performance improvement [34] and in the case when the load of interest and the target processor are known it is possible to do through DFG analysis [35–37]. Conversely, given a DFG, an optimal processor resource configuration for maximal parallelization can be deduced [35]. In other words, parallelization opportunities are determined by the DFG of the computation, but whether they can be exploited depends on the processor microarchitecture and resources.

This is another reason for working with fine-grained computational model. Every algorithm partitioning introduces, implicitly, a coarse-grained operation schedule, reducing the number of manipulations that can be done with the DFG. It also brings data handoff between units into the picture. This often introduces wait states until a meaningful unit of data is ready at the input of a subsystem, thus underutilizing the resources of that block. We will see this exact effect when we compare FPGA resource utilization of our solution and hand-crafted and human-scheduled solutions in literature, Chapter 6.

That is why our template structure exposes every computation unit to the compiler directly and without any constraints. Any unit can be used at any point in the schedule. We should note that better solutions could be possible. However, finding them would require DFG analysis capabilities beyond the scope of this work and at a higher level than anything in current literature.

With fine-grained computation structure, treating every operation in the DFG independently and having options to schedule it on any available unit of the proper type, the

compiler construction becomes simpler and many optimizations are simpler to achieve. A particular and important point here is that the algorithm can be partitioned and written in any form that is intuitive and convenient for the algorithm designer, while the compiler can violate this logical partitioning and calculate pieces of sequential logical blocks in parallel if it brings performance benefits. This is the process of software pipelining [38, 39].

Even when we impose this general structure onto the processor template we are left with many microarchitectural parameters that can strongly influence the final performance. In particular we need to look into pipeline structure of basic units and the question of processor unit count.

Higher unit count enables more parallel operations to be issued, but increases routing congestion and placement footprint as well as active and leakage power consumption. This limits the highest achievable running frequency. In fact, we will see that augmenting processor resources past a certain point can actually hurt the performance since the frequency will go down but the amount of parallelism that can be extracted will not make up for its decrease. Similar observations have been made in the energy efficiency studies of circuit-architecture codesign [29, 40].

Increasing the number of unit pipeline stages enables higher clocking frequency, but also increases the cycle length of critical path of the DFG. This is the fundamental tradeoff of the low-latency design: real-time latency of evaluation is proportional to the product of cycle-length of the DFG and the cycle-time. These variables are negatively correlated through technology dependent running frequency vs. cycle-latency curves of building blocks (such as post-placement crossbar transport path, floating point cores, etc.).

Previous observations are common base for accelerator construction in literature. The downside is the necessity of deep analysis of the algorithm before the construction of the accelerator can begin. This lengthens the design process by introducing an unnecessary sequential dependence. Taking the metaprogramming approach would enable construction of template before full processor configuration is known, thus offering more flexibility in design.

Templates are standard tools for code reuse in software. However, few HDLs have sufficiently rich abstraction model to allow similar techniques. This is the main reason why we use BSV in our design flow. It allows structured, top-down approach to system construction through early interface specifications. It is relatively simple to express structures

that could be instantiated with different number of submodules depending on parameters. Finally, achieving data type (in our case number representation) polymorphism is simple and static type checking goes a long way to keep many wiring errors in check.

Templates make generating different processor configurations easy, but have no impact on post-instantiation programmability of the structure. This is of no consequence if FPGA implementation is to be used as the fabric can be reprogrammed with a different processor instance at any time. However, ASIC implementation cannot rely on this. To keep the design flexible and provide possibility of in-system reprogramming in both FPGA and ASIC implementation obtained instances must be programmable. This requirement dictates architectures similar to previous supercomputer designs [41]. Popular systolic arrays [16, 42] are less desirable due to their rigid data flow and limited post-synthesis programmability.

3.2 Template Architecture

The core computation engine of our accelerator is shown in Figure 3-1. Three major logical subsystems are: crossbars, operation and data (memory) units. At this level of hierarchy the design is parameterized by the number of operation and data units, and the pipeline depths of operand and result transport bus, collectively referred to as System Side Bus (SSB). This provides sufficient flexibility to custom fit processor resources to the DFG of interest, as we will see in Chapter 6. Unit count parameters also determine the sizing of crossbar output multiplexers.

For the sake of generality and simplicity of scheduling, Chapter 4, we have an every-to-every crossbar for both data transport directions, Figure 3-1. Every operation unit can load operands from any of the data unit outputs. Similarly, every data unit can store result from the output of any operation unit. Furthermore, Figure 3-1 illustrates pipelining the crossbar transport path. The number of pipeline stages is left as a parameter to be determined during the algorithm compilation process described in Chapter 4.

Note that the crossbar latency shows as an effective latency increase of every operation unit and directly influences the DFG and its critical path cycle-length with a multiplier that depends on the number of operations on the critical path. Due to statically scheduled instruction stream, the latency of all communications over the crossbars is fully deterministic since the traffic pattern is fully determined and accounted for during compilation.

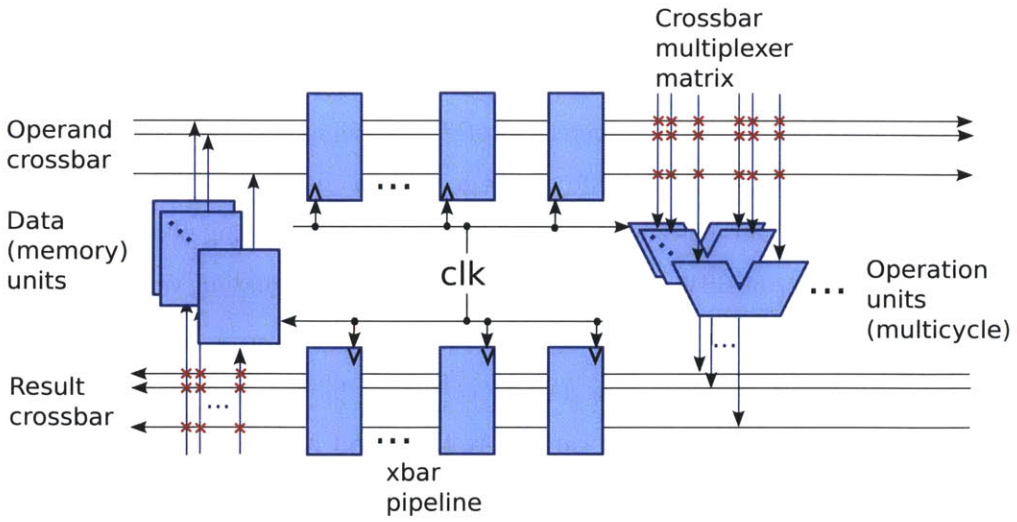


Figure 3-1: Processor architecture

The whole processor core is in a single, high-speed, clock domain. This is significant simplification for the compiler since it allows the compiler to work entirely in cycle-time domain. This decouples the compiler from the post-place-and-route performance of the processor for the purpose of DFG generation, optimization and scheduling.

In the rest of this chapter we present details of BSV implementation of the processor template. The static schedule assumption, a *design contract* between the processor and compiler design, allows simplification of data handoff protocols in hardware but requires care in coding approach to avoid, in our case, unnecessary automatic *micro-protocol handshake* [43] insertion. Similarly, describing polymorphic modules with non-type template parameters requires sidestepping limitations in the language and we present one possible solution.

The key to efficient processor implementation is instantiating all the operation unit types at the same level of hierarchy. This is done most easily by utilizing the *Connectable* class [43] to provide uniform and scalable description of communication between units and crossbars. However, the template should allow for instantiations of processors with specified number of units. To do this we must declare a vector of interfaces to those units [43]. And this is a point where even the flexibility offered by BSV is not enough for a simple, straightforward and self-documenting code.

Instantiating a vector requires specifying the size, which must have a specific type, called *numeric type* in BSV. The trouble arises as BSV does not allow passing numeric

types to *modules*, which are BSV's abstractions used to represent, among other things, actual digital circuits. Only interfaces, essentially protocols for interacting with a module, can have numeric types as parameters mostly for dealing with size relationships between bit-representable objects. Thus, to create a module that can be asked to instantiate a variable number of sub-modules we have two options, both feeling "hackish" and bringing us to the boundary of BSV (or rather its imperative skin emulating Verilog) expressiveness in modeling non-type related polymorphism.

The first option is to create a dummy object of the required size, and pass it to the module in question. The module can ignore the object and only determine the size of it, obtaining a numeric type value needed for sub-system instantiation. The other option is to pass the number of units for sub-system instantiation through the interface the module implements, as shown in Listing 3.1. The listing shows the interface to the processor instances generated from our template. All the numeric parameters are irrelevant at the interface and only meaningful in the implementation of the processor module. The interface announces that the processor presents the serial side of the Universal Asynchronous Receiver/Transmitter (UART) interface to the testing infrastructure.

While this solution works (compiles) and in our template it does not create much trouble as it only happens at the top level, it is far from ideal. The problem is that the number of units in each sub-system are purely implementation detail of the polymorphic module and have nothing to do with the interface the module implements for communication with the outside world. Thus, we are polluting the interface definition with unnecessary and potentially misleading information. It also makes two processors with different number of units appear as if they were implementing different interfaces. This could introduce a lot of overhead and incidental complexity if we were to use sets of processors in some higher level function generators. However, passing processor configuration through the interface instantiation is much more succinct than polluting the processor instantiation by construction of a large number of vacuous objects to smuggle a parameter into the polymorphic module constructor by appearing to be of certain size.

With this interface declaration we can write the skeleton of our processor template. It is, essentially, a large instantiation of all necessary components and a recipe for connecting them, like we show in Listing 3.2. Because all the modules are instantiated at the same level of hierarchy, the only way for modules to communicate is by connecting appropriate

methods of their interfaces through *Connectable* modules [43].

```
interface        PRCIfce#(
    numeric type n_addsub, numeric type n_mul,
    numeric type n_div, numeric type n_sqrt, numeric type n_comp,
    numeric type n_mem, numeric type pps_xbar, numeric type pps_mem_ctl,
    numeric type pps_comp, type op_t )  ;

    interface UARTIfce_srlSide#( Bit#(1) )  conn  ;
endinterface :  PRCIfce
```

Listing 3.1: BSV interface declaration of our processor template

This design approach makes it hard to make use of BSV's implicit ready/enable micro-protocol handshake [43] inserted across interface boundaries. However, in our system design they are unnecessary. Since our processor is statically scheduled, the compiler ensures availability of every resource during code compilation and no unit will try to access another at a moment when the other unit is not ready for communication. Hence, locks and synchronizations are not needed. This will have consequences in unit coding approach since we want to make sure that the handshake hardware is removed from the final processor code.

All the units are numbered, from 0, as they appear in the code. Since BSV does not seem to support non-numeric indexing, e.g. associative data structures, we are forced to use index arithmetic and synthetic naming of units.

```
module     mkProc#() ( ... )
    provisos(
        Add#( TAdd#( n_addsub, TAdd#(n_mul, TAdd#(
                    n_div, TAdd#(n_sqrt, n_comp) ))), 0, n_ops ),
        Add#( n_ops, n_mem, n_units),
        Add#( 0, 0, offset_addsub ),
        Add#( offset_addsub, n_addsub, offset_mul ),
        Add#( offset_mul, n_mul, offset_div ),
        Add#( offset_div, n_div, offset_sqrt ),
        Add#( offset_sqrt, n_sqrt, offset_comp ),
        Add#( offset_comp, n_comp, offset_mem ),
        Add#( offset_mem, n_mem, bound_mem )
    );
    // ***********************************************************
    // instantiate fsBus mux/wires
```

```
16      // ****************************************************************
17      MXBusIfce#( n_units, Bit#(8) ) fsBusMux <- mkOutMXBusAsync( );
18
19      // ****************************************************************
20      // instantiate ssBus muxes (one for each unit)
21      // ****************************************************************
22      // memory muxes
23      Vector#( n_mem,
24           MXBusIfceGen#( n_units, Bit#(32), pps_xbar )) muxes_mem;
25      for ( Integer idx = 0; idx < valueOf( n_mem ); idx = idx + 1 )
26      begin
27          muxes_mem[ idx ] <- mkOutMXBusPipe( (Bit#(32))'(0) );
28      end
29      Vector#( n_units,
30           MXBusIfceGen#( n_mem, Bit#(32), pps_xbar )) muxes_fpus_A;
31      Vector#( n_units,
32           MXBusIfceGen#( n_mem, Bit#(32), pps_xbar )) muxes_fpus_B;
33      Vector#( n_comp ,
34           MXBusIfceGen#( n_mem, Bit#(32), pps_xbar )) muxes_fpus_C;
35      for ( Integer idx = 0; idx < valueOf( n_ops ); idx = idx + 1 )
36      begin
37          muxes_fpus_A[ idx ] <- mkOutMXBusPipe( (Bit#(32))'(0) );
38          muxes_fpus_B[ idx ] <- mkOutMXBusPipe( (Bit#(32))'(0) );
39      end
40
41      for (Integer idx = 0; idx < valueOf(n_comp); idx = idx + 1 )
42      begin
43          muxes_fpus_C[ idx ] <- mkOutMXBusPipe( (Bit#(32))'(0) );
44      end
45      // ****************************************************************
46      // instantiate control units
47      // ****************************************************************
48      // uart ctl
49      BridgeMasterIfce#( n_units ) uart <- mkBridgeMaster();
50
51      // addsubs
52      Vector#( n_addsub,
53           OPUPipeCtrlIfce#( n_mem, NUM_type, pps_mem_ctl, pps_comp ) )
                opus_addsub;
```

```
54      for ( Integer idx = valueOf(offset_addsub);
55            idx < valueOf( n_addsub ); idx = idx + 1 )
56      begin
57          UInt#(7)  idx_ui  = fromInteger(idx);
58          let idx_rel = idx - valueOf(offset_addsub);
59          opus_addsub[ idx ]  <-  mkOPUCtrl( idx_ui, 1 );
60      end
61
62      // muls, divs, sqrts, comps, mems instantiations
63
64      // ************************************************************
65      // connections to the fsBus
66      // ************************************************************
67      mkConnection( fsBusMux.rPort.read, uart.takeResp );
68      mkConnection( uart.driveAddr, fsBusMux.selectBus );
69
70      // addsubs <-> fsbus
71      for ( Integer idx = valueOf(offset_addsub);
72            idx < valueOf(offset_mul); idx = idx + 1 )
73      begin
74          let idx_rel = idx - valueOf(offset_addsub);
75
76          mkConnection( opus_addsub[ idx_rel ].fsBus.putRsp,
77                        fsBusMux.wPorts[ idx ].drive );
78          mkConnection( uart.passData,
79                        opus_addsub[ idx_rel ].fsBus.getData );
80      end
81
82      // muls, divs, sqrts, comps, mems <-> fsbus
83
84      // ************************************************************
85      // connections to the ssBus
86      // ************************************************************
87      messageM( "ATTACHING: addsubs <-> ssBus" );
88      // addsubs <-> ssBus
89      for ( Integer idxADDSUB = 0;
90            idxADDSUB < valueOf(n_addsub); idxADDSUB = idxADDSUB + 1)
91      begin
92          // hook up addressing lines
```

```
93          mkConnection( opus_addsub[ idxADDSUB ].ssBus.selOpA,
94                        muxes_fpus_A[ idxADDSUB ].selectBus );
95          mkConnection( opus_addsub[ idxADDSUB ].ssBus.selOpB,
96                        muxes_fpus_B[ idxADDSUB ].selectBus );
97
98          // hook up output
99          mkConnection( muxes_fpus_A[ idxADDSUB ].rPort.read,
100                       opus_addsub[ idxADDSUB ].ssBus.getOpA );
101         mkConnection( muxes_fpus_B[ idxADDSUB ].rPort.read,
102                       opus_addsub[ idxADDSUB ].ssBus.getOpB );
103
104         // hook up inputs (from memories)
105         for ( Integer idxMEM = 0;
106               idxMEM < valueOf( n_mem ); idxMEM = idxMEM + 1 )
107         begin
108             mkConnection( opus_mem[ idxMEM ].ssBus.writeData,
109                           muxes_fpus_A[ idxADDSUB ].wPorts[ idxMEM ].drive );
110             mkConnection( opus_mem[ idxMEM ].ssBus.writeData,
111                           muxes_fpus_B[ idxADDSUB ].wPorts[ idxMEM ].drive );
112         end
113     end
114
115     // muls, divs, sqrts, comps, mems <-> ssbus
116
117     // ***************************************************************
118     // interface
119     // ***************************************************************
120     PRCIfce#( n_addsub, n_mul, n_div, n_sqrt, n_comp,
121         n_mem, pps_xbar, pps_mem_ctl, pps_comp, NUM_type )       ifce;
122     ifce = interface  PRCIfce
123         interface conn  = uart.srlPort;
124     endinterface :  PRCIfce;
125
126     return  ifce;
127
128 endmodule : mkProc
```

Listing 3.2: BSV module implementation of our processor template

3.3 Generalized Unit Architecture

Uniformity of the design at the top level enables code reuse at the unit level. All the operation and memory units have a very similar, to the number of operands, architecture. A unit diagram, for a two-operand unit, is shown in Figure 3-2.

The main control unit interacts with code memory through a pipelined transport layer. It decodes instructions and produces control signals for the input multiplexer and the slave unit. The slave unit is the unit performing actual work inside every top-level operation, predication and memory unit. In the current design the slave units are synthesized as FPGA cores and wrapped in BSV wrappers.

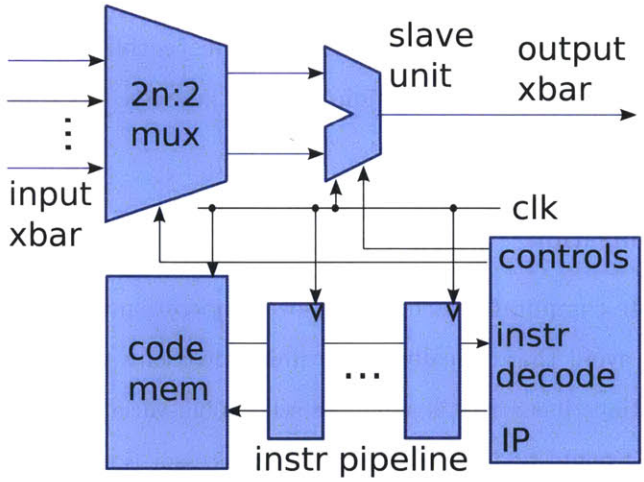


Figure 3-2: Unit micro-architecture

The result of the described design and unit coding approach is a very long instruction word (VLIW) [44] processor with distributed control and code memory. Though unusual, this design strategy simplifies the template design significantly.

In the first place, it simplifies the processor template design by providing a straightforward solution to scaling the number of units. If the design had centralized control unit, that unit itself would have to be a template. Furthermore, much of the code dealing with the data flow at the processor level would have to be replicated in the control unit itself.

Distributed nature of the control logic and instruction memory makes it possible to place these components close to their slave units, see Figure 3-2. This is quite convenient as it reduces the routing congestion and shortens many potentially critical paths in control logic. It is likely that it would have higher impact in an ASIC implementation than in an

FPGA implementation due to additional routing and placement constraints in FPGA.

Every unit is parameterized by the number of pipeline stages of that slave unit and the number of pipeline stages of the code transport path, Figure 3-2. Note that, unlike the latency of data transport path previously discussed, the latency of the code transport path lengthens the overall schedule only by the number of cycles equal to the number of pipeline stages.

3.3.1 Unit control

The described factoring of the processor core is convenient and powerful from the size-scalable template construction point of view. The question that arises is whether every unit must have complete schedule information and what tradeoff applies if they do not. Furthermore, problems of unit synchronization must be addressed in this, effectively, distributed system design.

Unit synchronization

Storing complete computation schedule in every operation unit would be expensive and we would want to avoid that. Ideally, every unit would only store its own activity schedule. The main question that arises is what the scheduling variable should be.

In general, it could be the full state of the processor (i.e. the aggregate state of all sub-units). However, this would significantly complicate processor control logic. Furthermore, it would require communicating unit states, a significant routing and power overhead.

For simplicity we would like to use local variable for scheduling in each processor unit. With units storing only their own schedule, the only such variable can be the cycle-count from some globally observable (i.e. observable by all units) event. This is possible because the whole core is in a single clock domain, and this is the solution we use in control units in Figure 3-2.

Absence of explicit synchronization and reliance on local cycle count greatly simplifies control and auxiliary communication circuitry within the processor, as previously discussed. However, it imposes some fundamental constraints on the schedules the compiler is allowed to produce.

Essentially, computations that pass through multiple physically different units must guarantee timeliness of results. The same constraint any multi-threaded application would

have. As the scheduling variable is just the cycle-count (all unit cycle counts are the same as noted earlier) that means that every unit must guarantee that their outputs will be valid by some cycle-count. In other words, all unit schedules must guarantee cycle-latency of every computation.

Code branching

Guarantees on cycle-latency of computations can be achieved in many different scenarios. In general, code branching could be used but the branches would have to be of equal cycle-length. In other words the shorter branch would have to be padded with No Operation Instruction (NOP) by the compiler. Moreover, when branch is taken the instruction pipeline in Figure 3-2 would have to be flushed and reloaded.

For simplicity of hardware, software and compiler implementation we chose to work without code branching. In other words, the same exact code stream must be executed every time, regardless of the input data. This limits the space of algorithms that can be executed, but we will see that our algorithms of interest can be written in a form that does not require code branching.

Avoiding code branching requires indirection and some primitive unit support, discussed in Chapter 4, to work at algorithm level, but it provides a lot of low-level implementation benefits.

Implementation implications

As we mentioned before, the compiler ensures availability of all the resources, making the implicit BSV micro-protocol [43] unnecessary for guaranteeing the correct functionality of the processor.

BSV handshake signals can be coded up in a way that results in an unloaded signal wire in the Verilog produced, which the synthesis tools remove them during optimization. Added benefit of this design organization is that local synchronization handshakes do not appear in the final design, thus removing a large source of critical paths.

To do this, in low-level core wrappers we tie off enable signals of *Action* methods to high, by declaring them with inhigh, making them always enabled [43]. At the slave-unit level, we re-wrap all the low-level modules in a way that unifies most of their interfaces. In this process we use *Wire* types to provide access to the slave unit without ready/enable micro-protocol.

Inside the unit module, we assert that the low-level core wrapper interface methods are always ready by requesting checking the *no_implicit_condition* scheduling attribute on the rules defining atomic actions of the slave unit state machine [43]. Furthermore, we assert *fire_when_enabled* flag and write rules with no explicit scheduling condition for the BSV compiler to ensure that we also interact with the CoreGen core in every clock cycle. All this is shown, on the example of the multiplier slave unit, in Listing 3.3.

```
1  module      mkXLNXSPFPUMulCore( XLNXSPFPUIfce ) ;
2      // instantiate the wrapped version
3      XLNXMulSPFPUIfce  mulFPU <- mkMulCore_low_level() ;
4
5      // synchronization wires
6      Wire#( Bool )    ceWire  <- mkDWire( True ) ;
7      Wire#( Bool )    rstWire <- mkDWire( False ) ;
8      Wire#( Bool )    ldWire  <- mkDWire( False ) ;
9
10     Wire#( Bit#(32) ) opAWire <- mkDWire( Bit#(32)'(0) ) ;
11     Wire#( Bit#(32) ) opBWire <- mkDWire( Bit#(32)'(0) ) ;
12
13     Wire#( Bool )    rfdWire <- mkDWire( False ) ;
14     Wire#( Bool )    rdyWire <- mkDWire( False ) ;
15
16     Wire#( Bit#(32) ) rsltWire<- mkDWire( Bit#(32)'(0) ) ;
17
18     // rules
19     (*fire_when_enabled, no_implicit_conditions*)
20     rule     clockRunningEXC ( True ) ;
21         mulFPU.ctrl.clockRunning( ceWire._read() ) ;
22     endrule : clockRunningEXC
23
24     (*fire_when_enabled, no_implicit_conditions*)
25     rule     resetStateEXC ( True ) ;
26         mulFPU.ctrl.resetState( rstWire._read() ) ;
27     endrule : resetStateEXC
28
29     (*fire_when_enabled, no_implicit_conditions*)
30     rule     sendOpAEXC     ( True ) ;
31         mulFPU.dataIn.opABus.put( opAWire._read() ) ;
32     endrule : sendOpAEXC
```

```
33
34      (*fire_when_enabled, no_implicit_conditions*)
35      rule    sendOpBEXC    ( True ) ;
36          mulFPU.dataIn.opBBus.put( opBWire._read() ) ;
37      endrule : sendOpBEXC
38
39      (*fire_when_enabled, no_implicit_conditions*)
40      rule    sendOprIn    ( True ) ;
41          mulFPU.ctrl.loadOp( ldWire._read() ) ;
42      endrule : sendOprIn
43
44      (*fire_when_enabled, no_implicit_conditions*)
45      rule    grabRFD    ( True ) ;
46          rfdWire._write( mulFPU.ctrl.getOpRFD() ) ;
47      endrule : grabRFD
48
49      (*fire_when_enabled, no_implicit_conditions*)
50      rule    grabRDY    ( True ) ;
51          rdyWire._write( mulFPU.ctrl.getRdy() ) ;
52      endrule : grabRDY
53
54      (*fire_when_enabled, no_implicit_conditions*)
55      rule    grabRSLT    ( True ) ;
56          let x <- mulFPU.dataOut.resBus.get() ;
57          rsltWire._write( x ) ;
58      endrule : grabRSLT
59
60      // bind methods
61      interface    FPUOperandIfce    dataIn ;
62          interface    GetPut::Put    opABus ;
63              method    Action    put( Bit#(32) opA ) if ( True ) ;
64                  opAWire._write( opA ) ;
65              endmethod :    put
66          endinterface :    opABus
67          interface    GetPut::Put    opBBus ;
68              method    Action    put( Bit#(32) opB ) if ( True ) ;
69                  opBWire._write( opB ) ;
70              endmethod :    put
71          endinterface :    opBBus
```

```
72      endinterface :              dataIn
73
74      interface       FPUResultIfce    dataOut ;
75          interface    GetPut::Get    resBus   ;
76              //method     ActionValue#( Bit#(32) )  get() if  ( True )  ;
77              //
78              // NOTE: as we chose to make this a GetPut::Get, beeing
                    ActionValue, we
79              // had to indicate an "enable" signal in BVI wrapper
80              // Since this binds to nothing, in the Verilog layer, I had to
                    define it
81              // as *inhigh*, but that required calling it in every cycle so we
                    had to
82              // let it go without "rdy" in BVI but "simulate rdy checking" here
83              method     ActionValue#( Bit#(32) )  get()
84                  if  ( ceWire._read() && !rstWire._read() && rdyWire._read() )
                        ;
85                  Bit#( 32 )  x ;
86                  x = rsltWire._read()  ;
87                  return x  ;
88              endmethod :              get
89          endinterface :          resBus
90      endinterface :              dataOut
91
92      interface    FPUControlIfce     ctrl  ;
93          method     Action     clockRunning( Bool clkStatusSet ) if ( True ) ;
94              ceWire._write( clkStatusSet ) ;
95          endmethod :        clockRunning
96
97          method     Action     resetState( Bool rstSignal )     if ( True ) ;
98              rstWire._write( rstSignal )    ;
99          endmethod :        resetState
100
101         // this is already protected by rdy on getXXX in BVI wrapper
102         method     XLNXStatCode  getStatus()            if ( ceWire._read() && !
                rstWire._read() ) ;
103             return  XLNXStatCode {  underflow  : mulFPU.ctrl.getUnderflow()   ,
104                                     overflow   : mulFPU.ctrl.getOverflow()  ,
105                                     invalid_op   : mulFPU.ctrl.getInvalidOp()
```

```
                                        ,
106                                     divide_by_0 : False
107                                 } ;
108         endmethod : getStatus
109
110         method Action takeOp( XLNXOpCode opr )
111             if ( ceWire._read() && !rstWire._read() ) ;
112             ldWire._write( True ) ;
113         endmethod : takeOp
114     endinterface : ctrl
115
116 endmodule : mkXLNXSPFPUMulCore
```

Listing 3.3: BSV Mul slave unit implementation

At this level, most operation units are almost identical, differentiating only in the type of the slave unit they instantiate, Figure 3-2. Every operation unit module implements the Front Side Bus (FSB) protocol state machine, code memory and transport pipeline and operand multiplexers. The only rule that is unique in every unit type is the instruction decoder.

This level of design reuse, enabled by the high level abstractions of BSV, is key for efficient and reusable design.

Instruction encoding

In the current implementation of the processor, no instruction set optimizations were performed. All units have 32-bit wide instruction words. For simplicity of instruction fetch and decode no packing was performed even when it was possible. In general, each instruction encodes one operation, specifying the input multiplexing for each operand and the particular operation performed for multi-operation units (e.g. AddSub, predicator).

3.3.2 Available unit primitives

A small set of low-level algorithmic primitives was developed for encoding of algorithms of interest. Functionality and presence of units performing simple algebraic operation in the processor is obvious. More interesting is the particular functionality of the predicator unit. We present basic parameters of each unit type, for completeness.

Table 3.1 outlines pipeline depth range available to the template for each type of unit available.

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 3.1: Operational unit types and parameters available in processor template

Data units

All data storage, initial and intermediate results, is done by the data storage units. Operation units do not have any local registers.

Slave unit used in data units is a two-port memory block. In every cycle the memory can store one and read one result, with read first semantics if the read and the store are addressing the same cell. Cycle-latency is 1 for data units and it cannot be modified.

Although possible, the current implementation does not parameterize the size of data memories. Memories are fixed to 1kWord, where Word has 32 bits.

Each data unit instruction encodes

- input multiplexer control,
- storage cell address, and
- read cell address.

Algebraic units

Depending on its type, every algebraic unit has appropriate operation unit as the slave, see Figure 3-2. In FPGA prototype implementation operation units are generated from intellectual property core templates (CoreGen).

Algebraic unit instruction encodes

- input multiplexer control for each operand separately, and
- operation type if applicable.

Conditional unit

Predication unit is somewhat more complex than other types of units. It was designed through compiler-enabled analysis and experimentation on various DFGs. Certain conveniences were discovered and used to optimize conditional processing for the target algorithmic class.

Basic functionality of the unit is predication. It instantiates a comparator unit as the slave and enables standard scalar relationships (e.g. $=, \neq, \leq, ...$) to be established.

For conditional processing of vector data, more is needed. For example, if we want to choose the one with smaller norm out of two vectors, because we do not have vectorized units (i.e. all our units operate on scalars) we have to construct conditional vector assignment. Having a conditional scalar assignment is sufficient. For this we implement the **?:** ternary operator present in many programming languages (e.g. C, Verilog, ...). In other words the conditional unit has a mode of operation where it evaluates

$$out = cond\ ?\ inA\ :\ inB.$$

The condition is considered true if it is positive when interpreted as a number, otherwise it is considered false. This is a slight optimization that enables cutting some computation in certain algorithms since computation results have meaning in conditional context. For example, this simple encoding decision avoids having to call predicate on two numbers, a and b, if a partial result $a - b$ is available. This saves multiple cycles and can add up in very serialized computation DFGs.

When working with scalars, having separate predication and conditional assignment can be unnecessary and a source of computation overhead. For this purpose a *fused* mode was introduced. In this mode the conditional value for the ?: is first calculated from inA and inB instead of being loaded as the third data input, which is what happens in pure conditional mode. This fusing of conditional and predicate mode enables short-circuiting of computation in simple calculations like:

$$min(inA,\ inB) = (inA\ \leq\ inB)\ ?\ inA\ :\ inB.$$

All these modes will come handy in our algorithm formulations in Chapter 5.

Conditional unit instruction encodes

- input multiplexer control for all three possible inputs,
- operation mode (predication, conditional or fused), and
- sub-operation (e.g. relation type in predication mode).

3.4 Test Infrastructure and Protocol

For testing of processor instances a *hardware-in-the-loop* testing setup was developed. For each algorithm implementation a Matlab and C++ benchmarks were developed. Processor results and their comparison to these benchmarks will be presented in Chapter 6.

To set up a test for FPGA prototype the Universal Serial Bus (USB) to UART interface to the ML605 development board from Xilinx was used. The processor core is instantiated in a testbed shown in Figure 3-3. The core processor is shown in red, while the testing infrastructure is in green.

The UART controls communication acting as FSB master. All processor core units act as FSB slave units in this context. UART can initiate burst reads or writes with any processor core unit. For this purpose each unit has an FSB address assigned in the process of BSV to Verilog compilation.

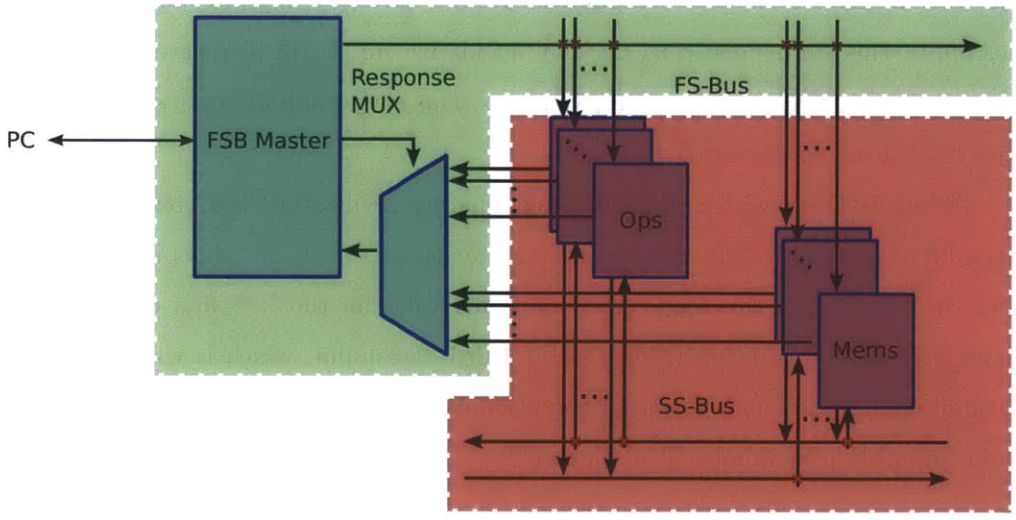


Figure 3-3: Testing setup for the processor

Using this simple protocol computer-side testbench can populate all data and code memories in the processor, and set each unit initial state (e.g. program counter). Broadcast

message to all units can be used to start programs in all processor core unit simultaneously.

FSB is only needed as testing infrastructure, but it has no purpose during normal operation when processor is running as a controller in an embedded system. In deployment, processor instances would need simple Input/Output (IO) units to interface Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC), i.e. sensor and actuator, units. The number of samples obtained in every sampling interval for most control algorithms is small.

3.5 Summary

In this chapter we present the architecture and the coding approach for construction of our compile-time configurable processor template.

Implementation details for describing modules with configurable number of sub-modules, in BSV, was discussed. We argued that the implicit synchronization micro-protocol BSV inserts in the design is unnecessary for this particular example design due to high-level synchronization functionality in our statically scheduling software compiler, described in Chapter 4. Methods for removing the micro-protocol circuits from the Verilog produced during the BSV compilation were discussed as well.

Finally, while BSV was chosen for its static typing and good top-down design support there are other, practical, advantages in using a language with high expressive power. Our design fits in $\approx 4,000$ lines of BSV production code. Even with synthesis boundary around every design module, ensuring that BSV compiler defines the module once and instantiates it multiple times instead of flattening the whole design, the BSV to Verilog code size ration is between, impressive, $10\times$ and $16\times$, depending on the processor size.

Chapter 4

Statically Scheduling Compiler

Computers are good at following instructions, but not at reading your mind.
—DONALD KNUTH

The number of lines of code a programmer can write in a fixed period of time is the same independent of the language used.
—CORBATÓ'S LAW

The design flow shown in Figure 2-1 recognizes the importance of the compiler in implementations seeking high-performance computation.

In embedded development, the compiler is usually treated as an oracle for software performance optimization. We hope to change this perspective by showing how useful the compiler can be in steering the process of hardware and algorithm design in the design flow outlined in Figure 1-3. For this, as we will see, the compiler should have an open and accessible architecture that allows us to see the program at various intermediate stages before the final translation is done and the code tested on the processor.

As progress is made on the compiler and processor implementations, and abstractions that model processor behavior become available, we can slowly morph the initial algorithm representation by replacing parts of it with structures that model evaluation on our processor. In this way we achieve an interactive design environment where the initial proof of correctness of the algorithm is slowly becoming our specialized implementation. The main benefit of this approach is that a functionally correct representation of our computation is available quickly and can be used to steer us towards a good solution for the custom

representation. The fact that it all happens within the same tool and on the same codebase is a major convenience in development.

In our template-based flow, almost all the system parameters meet inside the compiler, making it the backbone of the system design. It is also the only tool algorithm designers would use and as such it has the responsibility of representing the processor template at the algorithm design level. Because of this, our compiler is designed like an exploration tool first and final program translator second. In this sense we believe that an exploratory tool, akin to our compiler, would be useful in embedded computation accelerators even when no software component exists in the final system design.

The point of compiler-like tool in the design flow we follow is to provide sufficient information for the platform and algorithm decisions to be finalized. Whether a software component will eventually be implemented is less relevant, the compiler should offer us tools to make well informed decisions about the particular platform resources and algorithm formulation that work best together. This should be contrasted with algorithm-first and platform-first approach, which in the best case can profile one of the sides and never both together including the interaction between them.

As such, the compiler should give full control over the process to the user. Every operation on the DFG, from micro-optimizations to their visualization should be easily accessible and controlled by the user. Furthermore, the compiler itself should offer possibility to efficiently change its primitives if it is beneficial to do so.

Finally, the compiler should offer convenient and reusable input primitives for the target algorithmic class. In conjunction with reasonably fast compilation process, this enables efficient reimplementation and provides incentive to experiment with the design throughout the design cycle since every change in the algorithm, compiler or hardware component can be quickly evaluated.

4.1 Compiler Flow

Working with the compiler as the system design exploration tool proceeds in a few phases: algorithm input, DFG exploration and final program translation. Processor configuration is a pivotal piece of information at every stage of the process. Logical partitioning of the compiler is shown in Figure 4-1.

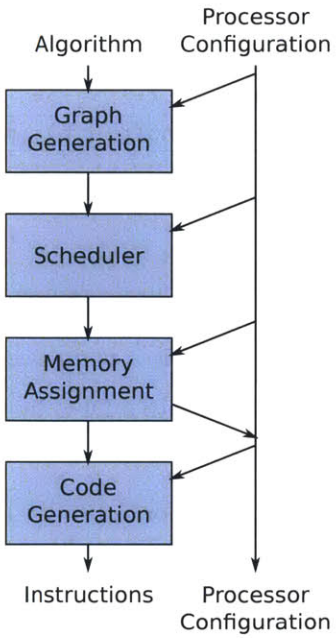


Figure 4-1: Compiler flow showing the four main stages and the interactions between the compiler and the processor configuration.

At the algorithm input stage, the set of primitives offered by the processor is of vital importance. For compiler design simplicity we should write algorithms in a form that can be expressed easily in terms of these primitive forms offered by the processor. Violating this would face us with the difficult task of mapping one language to another efficiently in the compiler, a problem encountered in HLS. We prefer to guide compiler by providing a more meaningful set of basic forms that designers can reason about easily.

During DFG exploration, the processor microarchitecture, the pipeline depths directly map to distance metrics on the DFG.

Finally, the number of processor units has major impact on scheduling and overall timing achievable.

4.1.1 Input Methods

There are two input methods the compiler provides for the algorithm designer. Both were designed for efficiency of representation and ease of testing against standard software implementations of numerical algorithms.

Assignment	=
Algebraic operators	+, -, *, /, sqrt()
Relational operators	<, <=, >, >=, ==, !=
Conditional function	cond(p,a,b), min(a,b), max(a,b)

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

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 [45, 46].

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.

```
addsubs 6 latency 5;
muls 5 latency 3;
divs 1 latency 15;
sqrts 1 latency 18;
xbar 1 1;
```

Listing 4.1: Example header for medium processor

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

As discussed in Chapter 3, the 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, 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.

```
d_x = A_x - B_x;
d_y = A_y - B_y;
d_z = A_z - B_z;
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

We instead decided to leverage already quite flexible function templates in C++ to design a more powerful DSL for the class of algorithms of interest.

C++ Template Based DSL

When using this input method, the compiler no longer has the traditional compiler flow. In a sense, in this situation the compiler might be thought of as a library that we link against. We write our algorithm, compile and link it with the compiler itself and run the resulting program as a tool for algorithm exploration.

We illustrate this on an example. A simple C++ function template, in this case implementing Euclidean norm calculation, is shown in Listing 4.3.

```
template <class T>
T norm(T* in, int size) {
    T temp = 0;
    for(int i = 0; i < size; i++) {
        temp += in[i] * in[i];
    }
    return sqrt(temp);
}
```

Listing 4.3: C++ function template for Euclidean norm calculation

To use this template we would write a program similar to the mock-up code in Listing 4.4.

If the C++ compiler, compiling the code snippet, is given the `TESTBENCH` flag it will compile with `typedef float dataType;`, thus instantiating the norm template with `norm<float>( float*, int )`. At the `main()` the tesbench code, `do_testbench()`, will be invoked to test the results. A full example, along with an explanation of the mechanics of the flow, can be found in the appendix.

If `TESTBENCH` flag is not defined the C++ compiler will compile with our `graphMaker` class, thus instantiating `norm<graphMaker>( graphMaker*, int )`. Then our custom compilation routines will be called to analyze the resulting computation graph and provide

```
#include "compiler.hpp"
#include "norm.hpp"
#include "testbench.hpp"

#ifdef TESTBENCH
typedef float dataType;
#define dataInit = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0}
#else
typedef graphMaker dataType;
#define dataInit
#endif

int main() {
    dataType input[10] dataInit;
    dataType out = norm<dataType>( input, 10 );
    #ifdef TESTBENCH
    do_compile();
    #else
    do_testbench();
    #endif
    return 0;
}
```

Listing 4.4: Example main function for compiling `my_algorithm`

reports or programming files.

The fact that the code can be targeted for software testing or for compilation to our processor by making an, essentially, trivial change on a handful of code lines is just a convenience. The important thing to note here is that both the testbench `norm<float>()` and the programming `norm<graphMaker>()` come from the exact same template code. This is where the correctness and performance of the algorithm meet in our design flow for embedded computation.

The need for algorithmic prototype and reimplementation of it for high-performance disappears. These two implementations become, for the most part, one. Early implementations of the algorithm can start well before the compiler and the processor template are fully developed, using `float` data type to prove the correctness of the algorithm implementation. At this stage we should not worry about performance of the algorithm implementation, just yet [47]. This, first, implementation is used to prototype the algorithm itself, in some form and representation, as one would do in any other design flow.

The important thing to remember is that, in this process, we always have the luxury of quickly reverting to a known data model (e.g. `float` in our example) to verify logical correctness of any large change in the code. As extensive testing is easy to access at each

step of development, we have significantly cut debug and reimplementation loops at the system-wide level. The result is a very quick feedback, almost interactive development with the fully functional system at all times.

At the same time we keep an explicit handle on processor abstractions and implementation through the processor template and the matching basic primitives in the compiler. In this way we keep very tight control of implementation at all the design levels. This allows us to control, to a good extent, the complexity of the design space the compiler needs to deal with, something that cannot be achieved in standard HLS. Besides BSV, depending on the direction of development, some other tools might be appropriate for this kind of hardware definition. Particularly interesting, due to open and transparent compiler, is Chisel [48].

The compiler does not behave as an oracle with opaque internal structure and unclear functional behavior. In this way we avoid problems of hardware inference from language constructs with insuficient expressiveness [27]. Exactly this is the point of human-created, human-readable templates at each level of the design.

To do all this, we developed the `graphMaker` class. This class offers basic abstractions for modeling the set of DFGs our processor template can execute efficiently. More concretely, this class offers facilities to build graphs with nodes chosen from the set of the operations supported by our processor, as discussed in Chapter 3.

The way to efficient implementation of such a class is through operator overloading offered by object oriented languages, C++ in this case.

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 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.

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

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

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

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.

The **graphMaker** class declaration can be seen in Listing 4.5.

```
/**
 *  \brief  This class creates a dependencyGraph through overloaded
 *  operations including +,-,*, and /
 *
 *  An example of creating a dependencyGraph using this class can be
 *  seen below:
 *  \code{.cpp}
 *      graphMaker a,b,c,d,tmp1,tmp2,sum;
 *      tmp1 = a * b;
 *      tmp2 = c + d;
 *      sum = tmp1 + tmp2;
 *  \endcode
 */
class graphMaker {
    public:
        graphMaker();
        graphMaker(double constant);
        graphMaker(int constant);
        graphMaker(string nodeName_req);
        graphMaker(const graphMaker& x);
        ~graphMaker();

        graphMaker operator+=(const graphMaker &rhs);
        graphMaker operator-=(const graphMaker &rhs);
        graphMaker operator*=(const graphMaker &rhs);
        graphMaker operator/=(const graphMaker &rhs);
```

```
27          graphMaker operator=(const graphMaker &rhs);
28
29          int getNodeID() const;
30          void renameNode(string name_req);
31
32          static void newGraph();
33          static schedGraph *graph;
34          static int node_group;
35          // Various optimizations
36          static bool keep_nodes; // keeps constants and nodes from previous
                graph
37          static map< pair<schedGraph*,schedNode*>, pair<schedGraph*,schedNode*>
                > node_translation;
38          static bool optimize;
39          static bool reuse_sub_expressions;
40
41          friend graphMaker operator+(const graphMaker &opA, const graphMaker &
                opB);
42          friend graphMaker operator-(const graphMaker &opA, const graphMaker &
                opB);
43          friend graphMaker operator-(const graphMaker &opB);
44          friend graphMaker operator*(const graphMaker &opA, const graphMaker &
                opB);
45          friend graphMaker operator/(const graphMaker &opA, const graphMaker &
                opB);
46          friend graphMaker sqrt(const graphMaker &opA);
47          friend graphMaker min(const graphMaker &opA, const graphMaker &opB);
48          friend graphMaker max(const graphMaker &opA, const graphMaker &opB);
49          friend graphMaker cond(const graphMaker &opA, const graphMaker &opB,
                const graphMaker &opC);
50          friend graphMaker eq(const graphMaker &opA, const graphMaker &opB);
51          friend graphMaker neq(const graphMaker &opA, const graphMaker &opB);
52          friend graphMaker lt(const graphMaker &opA, const graphMaker &opB);
53          friend graphMaker lteq(const graphMaker &opA, const graphMaker &opB);
54          friend graphMaker gt(const graphMaker &opA, const graphMaker &opB);
55          friend graphMaker gteq(const graphMaker &opA, const graphMaker &opB);
56
57          friend bool isZero( const graphMaker &x );
58          friend bool isOne( const graphMaker &x );
```

```
59          friend bool isTrue( const graphMaker &x );
60          friend bool isFalse( const graphMaker &x );
61
62          friend bool isConstant( const graphMaker &x );
63          friend double constVal( const graphMaker &x );
64
65      private:
66          void ensureValidNode() const;
67          bool isConstant() const;
68          float getConstantVal() const;
69
70          /// The pointer to the last calcNode this class represented
71          schedNode *node;
72          schedGraph *myGraph;
73  };
```

Listing 4.5: C++ **graphMaker** declaration

Once a candidate DFG, for a computation of interest, is constructed, our compiler offers various tools to evaluate its performance and, potentially, aid the designer in transforming it to an equivalent but more efficient form. We describe the compiler functionality and how it impacts the overall accelerator design flow. Details of the compiler implementation can be found in [49].

4.1.2 DFG Exploration and Optimization

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 are, technically, terminal nodes that are not used in further computation. Some represent results of the algorithm, but not all desired results are sinks, some are intermediate nodes. An example DFG can be seen in Figure 4-2.

There are some restrictions on DFG properties for them to be acceptable in our system.

Since the directed edges in the DFGs represent the order in which operations are done in the algorithm, there are no loops allowed in DFGs that should execute in our system. If there were a loop in a DFG it would imply that an operation needs to be computed in order

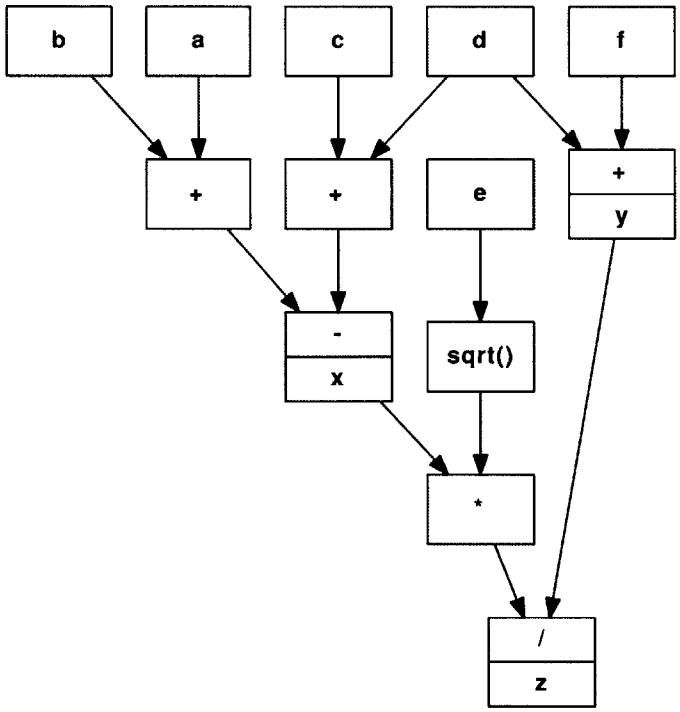


Figure 4-2: DFG example

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 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 4-2 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.

Another subset of processor parameters enters the picture at this stage. Microarchitectural parameters of operation units, namely the cycle-latencies, essentially determine distance on the computation DFG. The DFG in Figure 4-2 is annotated with distance information corresponding to the processor configuration in Listing 4.1.

One of the main tools for acceleration, including acceleration of numerical code, is finding parallelization opportunities [34]. Given a DFG, the parallelization opportunities are easy

to spot for certain graph structures [35]. However, many graphs can perform the same computation. In other words, the set of all DFGs can be partitioned into equivalence classes based on computations they perform. Not all the graphs in one equivalence class perform the same. Thus, it is important to chose the right graph structure for implementation.

Some equivalence transformations require planning, ingenuity and abstract thinking. We will see few such transformations in Chapter 6. Such optimizations usually introduce new objects into computation and combine them to desired effect. Achieving this automatically in the compiler would be quite challenging. On the other hand, some transformations can be done by simple local rule application. Analogous, and more well known example, of this difference is automatic derivative and anti-derivative finding. While the first is quite simple, the second is much more challenging [50].

For the most part, the set of optimizations that can be automatically performed or suggested by the compiler is based on heuristics aiming to increase number of parallel operations that can be issued at each cycle. Whether that really helps scheduling in every processor configuration is unclear [28]. However, with full control over optimizations performed, the algorithm designer can decide whether to use an optimization option or not. In our experience, most of the heuristics we present usually result in increased performance.

In this context, we will generally assume that the objective is to manipulate the computation DFG into an equivalent form with the shortest possible critical path. In other words, the longest chain from any source to any sink of the graph should me minimized, while preserving all the computation results. An example would be arranging a large sum into a balanced binary tree. Though intuitive, this is a heuristic: it does not guarantee shorter execution time after scheduling on a given processor. However, it does achieve the shortest cycle-time if sufficiently powerful processor is used [35].

Even with this qualification achieving the optimal packing of a DFG into a form with, globally, minimum critical path is an unresolved problem. In this implementation we will resort to local optimizations on the DFG and experimentation with formulation to achieve a good final timing result.

Most of these local optimizations will be based on the algebraic structure of numbers used in computation. For example, we will utilize associativity and commutativity of $+$ and $*$ operators to rebalance computation DFGs during exploration phase.

If fixed-point numbers are used this is always an option. However, floating-point num-

ber representation violates these relationships and the results might be inaccurate [51]. At this point there is no answer that can be given at the implementation level for something like this. It is the responsibility of the algorithm designer to evaluate any potential optimization in terms of performance and the correctness of the result. The compiler should offer tools to perform them if requested to do so. Our compiler infrastructure helps greatly with this with previously mentioned processor unit modeling that help designer seamlessly track the accuracy of the implementation by comparing it to the, well debugged, software implementation with minimal effort.

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. A node-collapsed version of the DFG in Figure 4-2 is shown in Figure 4-3. 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.

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-

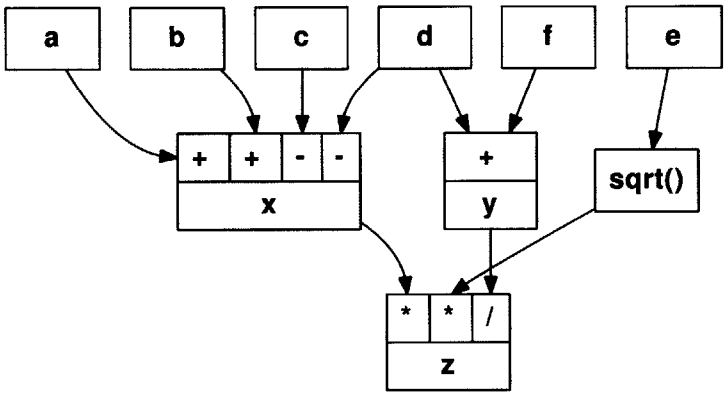


Figure 4-3: DFG with collapsed nodes

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.

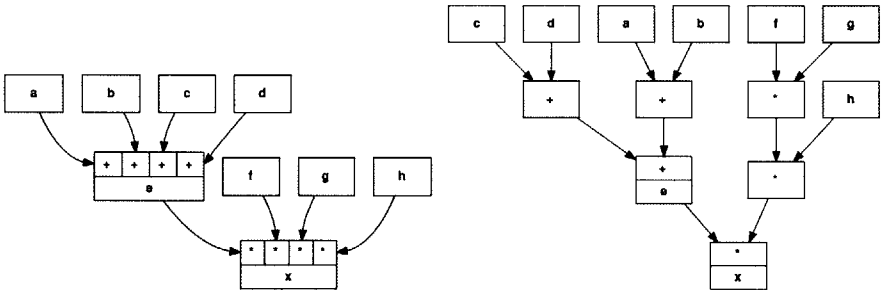
When expanding super-nodes, the goal is to expand the nodes in such a way that the critical path remains as short as possible. If the DFG is a single super-node of additions, then when expanding that node, the ideal configuration would be a balanced binary tree of additions because that has the shortest 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 4-4 shows a pair of super-nodes expanded optimally and expanded into balanced trees.

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.

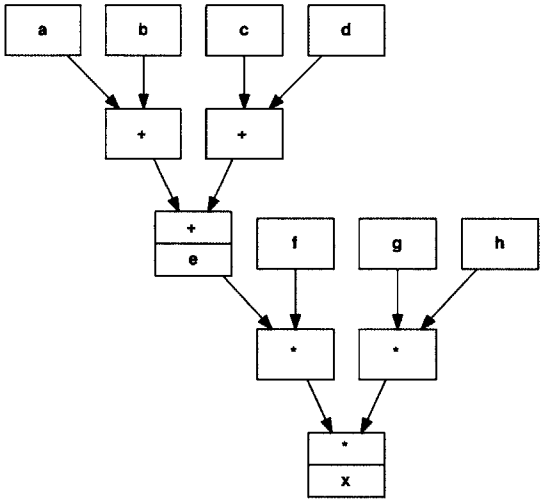
Constant Folding

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 [28]. If there are nodes in the DFG that depend only on constants, then the node can be evaluated and replaced with a constant, at compile time, before scheduling. Additionally, if there are



(a) DFG with super-nodes

(b) Optimal expansion



(c) Suboptimal balanced tree expansion

Figure 4-4: Super-node expansion: optimal and suboptimal

nodes that are being operated on by the identity element of the operation, those can be simplified too.

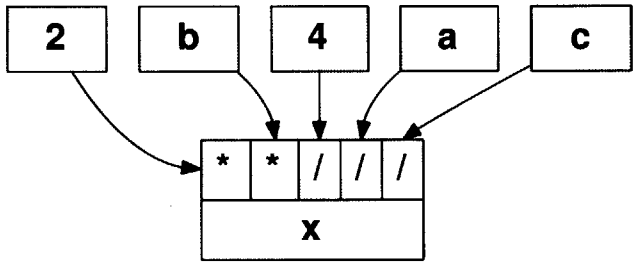
This optimization can also be performed on super-nodes to reduce the number of constants a super-node is dependent on. If two inputs in a super-node are constants, they can be replaced with the constant equal to the combination of the two constants. For example, the equation

$$x = \frac{2b}{4ac} \tag{4.1}$$

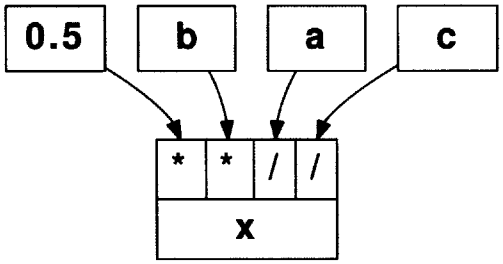
can be expressed as a super node as shown in Figure 4-5a. This super-node has a $\times 2$ and a $\div 4$ so the two of the can be replaced with a $\times 0.5$ resulting in the super node shown in

Figure 4-5b. This new super-node represents the optimized equation

$$x = \frac{0.5b}{ac}. \tag{4.2}$$



(a) Before optimization



(b) After optimization

Figure 4-5: Super-node representations of $x = (2b) \div (4ac)$ for constant folding optimization

Additionally, some select operations that depend on only one constant can be optimized as well using algebraic properties of 0 and 1 [28]. 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 $b \div 1$ 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.

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 form of this is replacing $a - a$ and

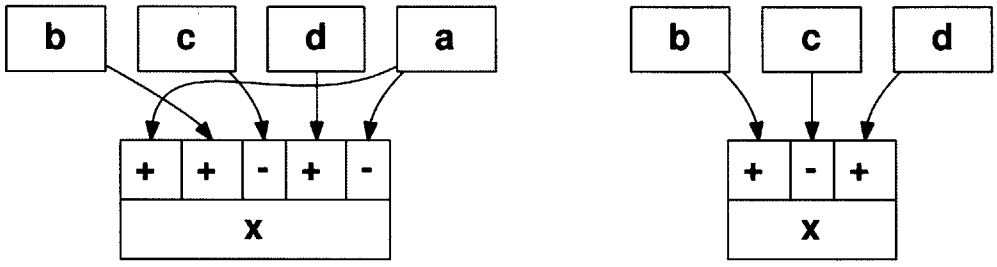
$a + (-a)$ with 0. For multiplication, this optimization replaces $a \cdot (a)^{-1}$ and $a \div a$ with 1.

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

$$x = (a + b) - (c - (d - a)) \tag{4.3}$$

can be expressed as a super node as shown in Figure 4-6a. 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 4-6b. This new super-node represents the equation

$$x = b - c + d. \tag{4.4}$$



(a) Unoptimized (b) Optimized

Figure 4-6: Super-node representations of $x = (a + b) - (c - (d - a))$ for inverse operation optimization

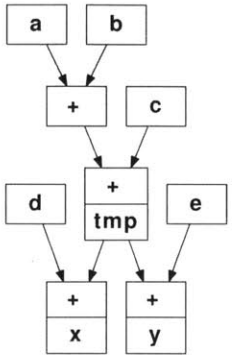
Operation Duplication

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.

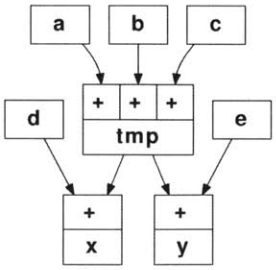
Even though it is not intuitive, sometimes it is advantageous to increase the number of operations in order to shorten the critical path and reduce the latency bound. This is the foundation for the operation duplication optimization; duplicating an intermediate result so trees can be rebalanced easier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

Consider the algorithm in Listing 4.6.

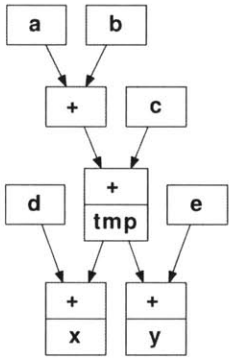
The DFG for this algorithm can be seen in Figure 4-7a. If all subtrees of the DFG made of associative operations are collapsed into super-nodes, the DFG is the one shown in Figure 4-7b. This algorithm cannot be collapsed into a single super-node because x and y both depend on *tmp*. Therefore, when the super-node is expanded, the resulting DFG as seen in Figure 4-7c is the same as the initial DFG.



(a) Initial DFG



(b) DFG with super-nodes



(c) Rebalanced DFG

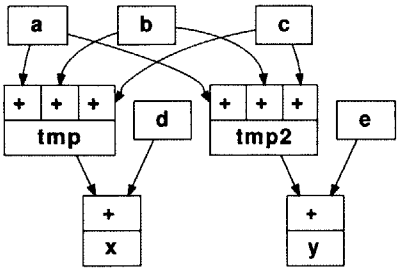
Figure 4-7: Tree rebalancing for `tmp=a+b+c; x=tmp+d; y=tmp+e;` without duplicating nodes

If the super-node for *tmp* is duplicated into a second node *tmp*2, then x could depend

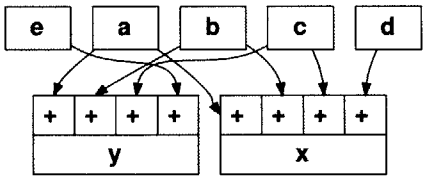
```
1 tmp = a + b + c;
2 x = tmp + d;
3 y = tmp + e;
```

Listing 4.6: Simple subexpression duplication example

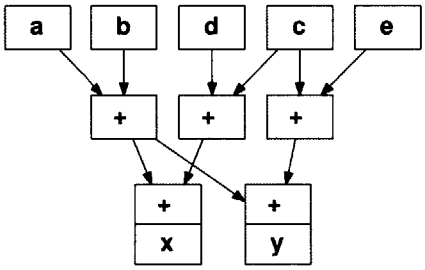
on *tmp* and *y* could depend on *tmp*2 like in Figure 4-8a. At this stage, the DFG can be fully collapsed into two super-nodes, one for x and one for y. These super-nodes can be expanded more efficiently than than the super-node in Figure 4-7b. When expanded, the super-nodes in Figure 4-8b become the DFG seen in Figure 4-8c.



(a) DFG with duplicated super-node



(b) Fully collapsed DFG



(c) Rebalanced DFG

Figure 4-8: Tree rebalancing for `tmp=a+b+c; x=tmp+d; y=tmp+e;` with duplicating nodes

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.

4.1.3 DFG Scheduling and Code Generation

Unlike optimizations we discussed, scheduling of a given DFG graph is a well studied problem with many satisfactory results [35–38]. At this stage of compilation the final piece of information about the processor instance enters the picture: the number of operation units and storage memories determines how big of a piece of the DFG can be issued for execution

at each cycle.

To schedule the DFG we perform node prioritization to determine urgency of each node. More urgent nodes are scheduled earlier if possible. While scheduling it is assumed that infinite memory resources are available and all intermediate results are simply named, but not assigned an address. After scheduling, the schedule is post-processed to assign memory locations to each computation result.

Depth Priority

At this stage, the DFG contains information about all the required operations, but it needs information about relative importance of nodes for achieving an efficient schedule. 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 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

operation 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 4-2, 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 $f + g$.

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 be executed in 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.

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 operation unit at a time, and only one variable can be read from a memory at a time (even though multiple units 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.

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.

4.2 Summary

This chapter presented a custom optimizing compiler design. In our template-based design flow, the compiler is used as the main design exploration and steering tool. This central position in the design dictates a flexible and extensible design for the compiler.

The compiler front-end is a DSL embedded in C++, using C++ template metaprogramming. This enables rapid development of compiler-side models of processor units. The reason is the easy access to testing of new constructs through simple changes in instantiation parameters.

The core compiler is a configurable set of low-level optimization routines operating on DFGs. It offers full control of optimizations performed as well as easy access to many intermediate measures of the DFG performance.

The compiler back-end is the compiler in the narrow sense, usually referred to as "the compiler" in embedded software development. It provides DFG scheduling and translation into programming files for the processor instance of interest.

Throughout the chapter, we emphasized the relationship between the processor instance configuration and compiler functionality in each compilation stage.

Chapter 5

Algorithms

An algorithm must be seen to be believed.

—Donald Knuth

Programs must be written for people to read, and only incidentally for machines to execute.

—Gerald Sussman

In this chapter we analyze a few interesting algorithms often found in embedded acceleration literature. The material presented here explains the underlying structures for discussion of the results in Chapter 6. In particular, we derive the exact formulations we use to construct specific accelerators in Chapter 6. This is presented through a process of transforming formulations from the forms used in literature to the form used in our formulation, for each algorithm variant. The reader should bear in mind that the discussion presented in this chapter is kept at the most generic level to maximize the utility of achieved formulations. We look into particular MPC problems, as concrete instances of the general methods in this chapter, in the next chapter where we also comment on the workload, in terms of sizes of matrices and schedule lengths, required for calculating each problem instance.

When manipulating pure numerical computation, we pretend to pack the problem formulation in a form that reduces the size of matrices that undergo any kind of matrix decomposition. There are two main reasons for this. First, smaller matrices require fewer operations to be decomposed. Furthermore, as will be observed in examples in Chapter 6, matrix decompositions have very sequential DFG and it is often beneficial to substitute a decomposition of a large matrix for a decomposition of a smaller matrix and multiple

matrix-vector multiplies, if the same result can be achieved. The reason for this is that it is trivial to parallelize the matrix vector multiply. The second reason is numerical stability of computation. While matrix pivoting can be done with the resources our processor template offers, it is desirable to avoid it when speed is important and most implementations do so [3,32]. Growth factors in matrix decompositions are related to the size of matrix [51] and smaller matrices can be accommodated with smaller number representation while achieving desired accuracy.

Special attention is paid to conditional (or data dependant) processing parts of algorithms. As explained in Chapter 3, it is not hard to implement branching infrastructure in our processor template. However, code branching comes with instruction pipeline flushing, usually costing multiple operation cycles to re-populate. It is best if conditionals can be avoided or hidden in predication atoms. In this chapter we will see that achieving this requires changing the data structures used in the algorithm or analyzing conditionals to derive simplified forms.

The main focus is on several variants of MPC algorithm with efficient implementations reported in recent literature [3, 7, 22, 32]. Being computationally very demanding these make good demonstration of the capabilities of the design flow. In particular, we wish to demonstrate the capability of this flow to guide us in finding a good overall system configuration and the appropriate algorithm form. At the same time, high volume of publications reporting efficient implementations gives us an opportunity to set the performance bar and evaluate our approach as the implementation technology for this and similar problems in our scope.

Most MPC implementations are built on top of a QP optimization algorithm. To show as much of the design behavior and simplify the exposition we present the MPC design incrementally. In this implementation we use IP method for QP because of its efficiency in practice.

5.1 Interior Point Method for Quadratic Programming

For our implementation we follow the Mehrotra's predictor-corrector method as presented in [21]. We repeat it here for completeness and easy reference.

In general we are interested in solving the following optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^T Qx + q^T x \\ \text{subject to} \quad & Gx \leq h,\ Ax = b, \end{aligned} \tag{5.1}$$

given the positive semidefinite matrix $Q_{n\times n}$, matrices $G_{i\times n}$, $A_{e\times n}$ and vectors $q_{n\times 1}$, $h_{i\times 1}$ and $b_{e\times 1}$.

Strictly speaking, the problem defined in Equation (5.1) could be infeasible. This happens when the regions defined by the inequalities and equalities happen to be empty or do not intersect. Understandably, this is undesirable in real-time and mission-critical system functions, but it can not be addressed at the implementation level. In case of a possibly infeasible QP the algorithm designer must specify the desired implementation behavior.

Usually, MPC formulations do not have this problem. In the formulations we are implementing this is actually impossible. In cases where it is the flexibility of the template-based design flow is ideally suited for handling such problems even very late in the project development.

The first step towards the solution of (5.1) is substitution of inequality constraints with equality constraints and a simpler inequality constraint. The problem becomes

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^T Qx + q^T x \\ \text{subject to} \quad & Gx + s = h,\ Ax = b,\ s \geq 0, \end{aligned} \tag{5.2}$$

where vector $s_{i\times 1}$ is the vector of slack variables.

We assume that the problem in (5.1) is strictly feasible, an assumption that will be justified when we present MPC formulations in the next section. This implies that the Karush-Kuhn-Tucker (KKT) conditions

$$\begin{aligned} & Ax = b \\ & Gx + s = h, && s \geq 0 \\ & Qx + q + G^T z + A^T y = 0, && z \geq 0 \\ & s_k z_k = 0,\ k \in \{1, \ldots, i\}, \end{aligned} \tag{5.3}$$

where $z_{i\times 1}$ and $y_{e\times 1}$ are dual variables corresponding to the inequality and equality con-

straints, respectively, are necessary and sufficient conditions of optimality [33].

In our MPC formulations we will avoid using equality constraints. This is a performance driven optimization that, technically, limits the set of MPC problems we can solve. However, all the MPC whose implementations we could find in literature can be written in the form with no equality constraints. Thus, we can simplify the (5.1) we treat, and improve its performance, by removing the equality constraints from it. The exact QP we will be solving from now on is given by

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^T Qx + q^T x \\ \text{subject to} \quad & Gx + s = h,\ s \geq 0, \end{aligned} \tag{5.4}$$

with the corresponding KKT conditions

$$\begin{aligned} & Gx + s = h, && s \geq 0 \\ & Qx + q + G^T z = 0, && z \geq 0 \\ & s_k z_k = 0,\ k \in \{1, \ldots, i\}. \end{aligned} \tag{5.5}$$

From this point on, our objective is to find the solution to the KKT conditions (5.5) for the Problem (5.4). To do this we will follow the program outlined in [21]. The solver will operate in two distinct phases: initialization and iteration.

5.1.1 Initialization

To initialize the solver for (5.2) we must solve the system of equations [3]

$$\begin{pmatrix} Q & G^T & A^T \\ G & -I & 0 \\ A & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ z \\ y \end{pmatrix} = \begin{pmatrix} -q \\ h \\ b \end{pmatrix}. \tag{5.6}$$

As mentioned, we will not need the equality constraints, so we simplify the initialization to

$$\begin{pmatrix} Q & G^T \\ G & -I \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix} = \begin{pmatrix} -q \\ h \end{pmatrix}, \tag{5.7}$$

by considering the simplified QP form (5.4). Since all matrices involved in initialization phase are constant many operations needed can be precomputed. This means that the

general (5.6) and the simplified form (5.7) have very similar computation costs.

We could solve the (5.7) directly in the form given. The matrix is non-singular and it has LDL^T decomposition. However, an alternative form might be more appropriate. To derive it we use the special structure of the initialization matrix.

From the second row we can write

$$z = Gx - h, \tag{5.8}$$

which, after substitution into the first row, yields

$$(Q + G^T G)x = -q + G^T h. \tag{5.9}$$

We will initialize by solving (5.9) and (5.8). By doing this we trade off the size of the (5.7) matrix decomposition, for a smaller matrix (5.8) decomposition and a matrix-vector multiplication (5.9). This formulation was selected through compiler experiments where it was observed that matrix-vector multiplication can be parallelized much more effectively than most matrix decompositions. Details of this tradeoff exploration can be found in Section 6.1.

With the solution obtained we can set $x^{(0)} = x$. Following the program in [21], we next calculate $s^{(0)}$ as

$$\begin{aligned} \alpha_p &= \inf\{\alpha | -z + \alpha \geq 0\} \\ s^{(0)} &= \begin{cases} -z & \alpha_p < 0 \\ -z + (1 + \alpha_p) & \text{otherwise,} \end{cases} \end{aligned} \tag{5.10}$$

where vector to scalar addition is understood coordinate-wise. This calculation has a conditional and we must transform it for efficient execution. We use the following:

$$\begin{aligned} \alpha_p &= \max z \\ \beta_p &= \begin{cases} 0 & \alpha_p < 0 \\ 1 + \alpha_p & \text{otherwise} \end{cases} \\ s^{(0)} &= -z + \beta_p. \end{aligned} \tag{5.11}$$

The expressions for $s^{(0)}$ in (5.10) and (5.11) are equivalent, but the latter can be compiled

into a static, finite search tree. The conditional was moved to the calculation of the scalar value β_p and then a simple addition is applied to all the coordinates of z.

Similarly, we transform the expression for $z^{(0)}$

$$\begin{aligned} \alpha_d &= \inf\{\alpha | z + \alpha \geq 0\} \\ z^{(0)} &= \begin{cases} z & \alpha_d < 0 \\ z + (1 + \alpha_d) & \text{otherwise,} \end{cases} \end{aligned} \tag{5.12}$$

to the form used in the formulation

$$\begin{aligned} \alpha_d &= -\min z \\ \beta_d &= \begin{cases} 0 & \alpha_d < 0 \\ 1 + \alpha_d & \text{otherwise} \end{cases} \\ z^{(0)} &= z + \beta_d. \end{aligned} \tag{5.13}$$

We start the solver iteration phase from $(x^{(0)}, s^{(0)}, z^{(0)})$.

5.1.2 Iteration

In every iteration we calculate the direction for the next move as a sum of two directions. These are the predictor and corrector step of the iteration. As the directions in both steps are calculated from the same linear system but with different right-hand sides [21] we will derive a generalized form and then specialize for each step.

Both the predictor and corrector step are obtained by solving the system of the following form

$$\begin{pmatrix} Q & 0 & G^T \\ 0 & S^{-1}Z & I \\ G & I & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta s \\ \Delta z \end{pmatrix} = \begin{pmatrix} x^{rh} \\ s^{rh} \\ z^{rh} \end{pmatrix} \tag{5.14}$$

where we use Z and S to denote diagonal matrices with vectors z and s on the diagonal, respectively. We also simplify the system from [21] by ignoring equality constraints due to the MPC formulation we are using.

To solve this system we start from the third row in (5.14) writing

$$\Delta s = z^{rh} - G\Delta x \tag{5.15}$$

and further substitution in the second row yields

$$\Delta z = s^{rh} - (S^{-1}Z)z^{rh} + (S^{-1}Z)G\Delta x. \tag{5.16}$$

Finally, the first row can now be expressed as

$$(Q + G^T(S^{-1}Z)G)\Delta x = x^{rh} - G^T s^{rh} + G^T(S^{-1}Z)z^{rh}. \tag{5.17}$$

Matrix $(Q+G^T(S^{-1}Z)G)$ is guaranteed to be positive semidefinite, and in all MPC formulations it is positive definite. The LDL^T decomposition will always exist for it. Thus, the system can be solved by LDL^T decomposition in (5.17), followed by the forward-backward substitution. After we solve for x, other variables follow from (5.15) and (5.16).

In every iteration we are looking to improve the current solution $(x^{(k)}, s^{(k)}, z^{(k)})$ and produce a new one $(x^{(k+1)}, s^{(k+1)}, z^{(k+1)})$. The number of iterations can be fixed [3, 17] or determined by an exit condition calculated from the current solution. With these general considerations we evaluate an IP iteration as follows [21]

1. Solve the predictor step system

$$\begin{pmatrix} Q & 0 & G^T \\ 0 & S^{-1}Z & I \\ G & I & 0 \end{pmatrix} \begin{pmatrix} \Delta x^{aff} \\ \Delta s^{aff} \\ \Delta z^{aff} \end{pmatrix} = \begin{pmatrix} -(G^T z^{(k)} + Qx^{(k)} + q) \\ -z^{(k)} \\ -(Gx^{(k)} + s^{(k)} - h) \end{pmatrix} \tag{5.18}$$

   using the approach outlined in (5.17, 5.16, 5.15).

2. Calculate

$$\begin{aligned} \alpha_{cc} &= \sup\{\alpha \in [0,1] \mid s^{(k)} + \alpha\Delta s^{aff} \geq 0,\ z^{(k)} + \alpha\Delta z^{aff} \geq 0\}, \\ \sigma &= \left(\frac{(s^{(k)} + \alpha_{cc}\Delta s^{aff})^T (z^{(k)} + \alpha_{cc}\Delta z^{aff})}{s^T z} \right)^3, \\ \mu &= \frac{(s^{(k)})^T z^{(k)}}{i}, \end{aligned} \tag{5.19}$$

where i is the number of inequality constraints as denoted in (5.1). An efficient way to calculate α_{cc} will be discussed after the algorithm outline.

3. Using the solution $(\Delta x^{aff}, \Delta s^{aff}, \Delta z^{aff})$ solve the corrector step system

$$\begin{pmatrix} Q & 0 & G^T \\ 0 & S^{-1}Z & I \\ G & I & 0 \end{pmatrix} \begin{pmatrix} \Delta x^{cc} \\ \Delta s^{cc} \\ \Delta z^{cc} \end{pmatrix} = \begin{pmatrix} 0 \\ S^{-1}(\sigma\mu - \Delta z^{aff} \odot \Delta s^{aff}) \\ 0 \end{pmatrix} \tag{5.20}$$

where $\odot$ denotes the Hadamard (i.e. componentwise) product of vectors. For solving we use the approach outlined in (5.17, 5.16, 5.15). Note that the matrix in both the predictor (5.18) and the corrector (5.20) system is the same matrix. This means that the matrix appearing in (5.17) is the same in both cases. Thus, its LDL^T decomposition need to be calculated only once. This is the standard practice in IP solvers. Furthermore, compiler constant folding will simplify expressions (5.16, 5.15) for the corrector step calculation. At this point we can calculate the final move direction

$$\begin{aligned} \Delta x &= \Delta x^{aff} + \Delta x^{cc}, \\ \Delta s &= \Delta s^{aff} + \Delta s^{cc}, \\ \Delta z &= \Delta z^{aff} + \Delta z^{cc}. \end{aligned} \tag{5.21}$$

4. Update the current solution by

$$\begin{aligned} \alpha_{ls} &= \min\{1,\ 0.99\sup\{\alpha \geq 0 \mid s^{(k)} + \alpha\Delta s \geq 0,\ z^{(k)} + \alpha\Delta z \geq 0\}\}, \\ x^{(k+1)} &= x^{(k)} + \alpha_{ls}\Delta x, \\ s^{(k+1)} &= s^{(k)} + \alpha_{ls}\Delta s, \\ z^{(k+1)} &= z^{(k)} + \alpha_{ls}\Delta z. \end{aligned} \tag{5.22}$$

We now look at the conditionals in steps 2 and 4 and their efficient formulation within the constraints and resources of a statically scheduled system like ours.

In step 2 we are calculating $\alpha_{cc} = \sup\{\alpha \in [0,1] \mid s^{(k)} + \alpha\Delta s^{aff} \geq 0,\ z^{(k)} + \alpha\Delta z^{aff} \geq 0\}$. In this case components of Δz^{aff} and Δs^{aff} could have negative sign and the expressions cannot be simplified as easily as between (5.10) and (5.11).

In this case we will rely on the evaluation context and the semantics of the result where

it is used to derive a substitute form:

1. Every step of initialization and iteration preserves $z \geq 0,\ s \geq 0$ constraints given in KKT conditions (5.3).

2. Result α_{cc} is expected in $[0, 1]$ range.

The purpose of these calculations is to determine how far we can move in the desired direction without violating primal or dual feasibility in (5.3). The way they do it is by calculating when the constraint would be violated if we moved only along one coordinate and then find the minimum of all such moves.

Note that we can never exit the feasibility region along any coordinate r where $\Delta z_r^{aff} \geq 0$, since $\alpha \geq 0$ and $z_r^{(k)} \geq 0$. Thus, in that case we are free to just return the maximum meaningful value for α which is 1 in this case. Let us define $\mathcal{P}_v = \{\ r\ |v_r \geq 0\}$, the set of coordinate indices for which a given vector is non-negative.

Similarly, we can define the set of coordinate indices for which a given vector is strictly negative, $\mathcal{N}_v = \{\ r\ |v_r < 0\}$. On this coordinate subset, under the assumptions on (z, s, α), we can write

$$\alpha_{cc} = \min\{1, (s^{(k)} \oslash |\Delta s^{aff}|)_{\mathcal{N}_{\Delta s^{aff}}}, (z^{(k)} \oslash |\Delta z^{aff}|)_{\mathcal{N}_{\Delta z^{aff}}}\} \tag{5.23}$$

where $\oslash$ denotes the componentwise division of vectors and index sets indicate the vector coordinate indices where the expression is valid and should be evaluated.

In a statically scheduled system data dependent processing is not straightforward. We have already discussed certain data dependent processing cases where static schedule performs as well as more flexible code execution scheme. However, the variable-size *min* in (5.23) is more challenging. Luckily, we should not treat it as a function with variable number of arguments, nor do we have to.

We prefer solutions with guaranteed and easy to check execution time and variable-length argument list could be problematic in such cases. However, the fixed number of inequality constraints and our earlier observations allow us to construct a single expression (and schedule) that functions correctly in all cases. As expected, if considered separately, the latency of such a schedule is fixed and equals the worst case latency over all the cases that it can correctly handle. For reasons that we discuss in latter chapters, this is not a

problem in practice for this set of applications.

Taking into account our previous observations we can write

$$\begin{aligned}\theta^z &= (z^{(k)})_{\mathcal{P}_{\Delta z^{aff}}} + (-\Delta z^{aff})_{\mathcal{N}_{\Delta z^{aff}}},\\ \theta^s &= (s^{(k)})_{\mathcal{P}_{\Delta s^{aff}}} + (-\Delta s^{aff})_{\mathcal{N}_{\Delta s^{aff}}},\\ \alpha_{cc} &= \min\{1, z^{(k)} \oslash \theta^z, s^{(k)} \oslash \theta^s\},\end{aligned} \tag{5.24}$$

where we use the same convention for index set subscript: the expression should be evaluated only for denoted coordinate indices, otherwise ignored.

An alternative formulation is given by

$$\begin{aligned}\theta^z &= (z^{(k)})_{\mathcal{P}_{\Delta z^{aff}}} + (\max\{z^{(k)}, \ -\Delta z^{aff}\})_{\mathcal{N}_{\Delta z^{aff}}},\\ \theta^s &= (s^{(k)})_{\mathcal{P}_{\Delta s^{aff}}} + (\max\{s^{(k)}, \ -\Delta s^{aff}\})_{\mathcal{N}_{\Delta s^{aff}}},\\ \alpha_{cc} &= \min\{z^{(k)} \oslash \theta^z, s^{(k)} \oslash \theta^s\}.\end{aligned} \tag{5.25}$$

The form to be used depends on the implementation of floating point units, whether we expect numerical problems when when approaching the feasible set boundary within desired accuracy. We might also consider how the evaluation graph for this calculation fits in the overal algorithm timing.

The same apprach can be used to transform expression for α_{ls} in (5.22).

5.2 Model Predictive Control Formulations

After treating the general case of QP in the previous section, in this one we show how to write several MPC variants in the form of (5.4). In this work we focus on efficient implementations of numerical DSP for embedded systems. It is not our intention to argue any particular control scheme. As we noted during derivation in Section 5.1.2, we will not need the equality constraint handling.

5.2.1 MPC: Linearizing Pre-Equalization

In this application we use our framework and infrastructure to design a linearizing pre-equalizer using MPC. The example is taken from [3]. The problem setup is shown in Figure 5-1. We use the standard convenience representation for implementation of the transfer

function of a linear system [52]. A Hammerstein structure, a saturation-type nonlinearity followed by a stable Linear Time Invariant (LTI) system, is linearized by MPC. The goal of the controller is to minimize the effects of the saturation. It is assumed that the controller has access to the system model and the scalar input signal u with T sampling intervals of lookahead.

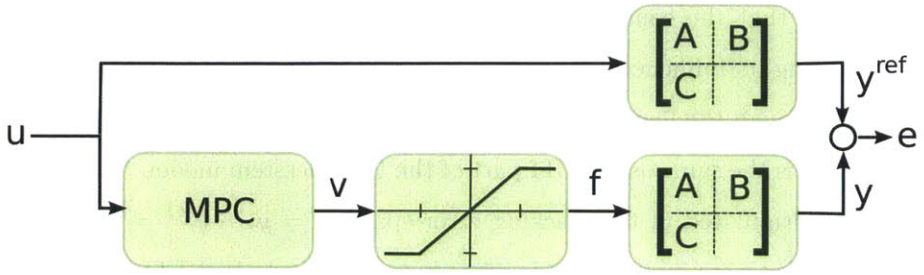


Figure 5-1: MPC as linearizing pre-distortion

Full details of the model and derivation can be found in [3, 21].

The LTI part of the dynamical system under control is modeled in the State Space (SS) form as

$$\begin{aligned} x_{\tau+1} &= Ax_\tau + Bf_\tau \\ y_\tau &= Cx_\tau, \end{aligned} \tag{5.26}$$

where x_τ is the state of the model at time τ, y_τ is the output vector, and A, B and C are constant matrices.

In this setup the forcing term is the output of a saturation when driven with the input signal, i.e. $f_t = sat(\nu_t)$, as shown in Figure 5-1. The role of MPC as the linearizing pre-equalizer is to control the system under the condition that the control signal ν never enters the saturation region of the system control input, thus enabling us to write $sat(\nu_t) = \nu_t$ [3].

We start our derivation by repeating their basic MPC form

$$\begin{aligned} \min \quad & \sum_{\tau=t+1}^{t+N} e_\tau^2 + \tilde{x}_{t+N}^T P \tilde{x}_{t+N} \\ \text{s.t.} \quad & \tilde{x}_{\tau+1} = A\tilde{x}_\tau + B(\nu_\tau - u_\tau) \\ & e_\tau = C\tilde{x}_\tau \\ & |\nu_\tau| \leq 1, \ \tau = \{t, \ldots, t+N-1\}. \end{aligned} \tag{5.27}$$

We deviate slightly from the original formulation in [3] in exclusion of e_t from the objective,

since it is a constant. Additionally, we renamed the prediction/control horizon length to $N = T + 1$, to avoid confusion with matrix transpose and to achieve a more compact notation.

The problem is formulated as a minimum square error tracking of the reference signal on the prediction horizon. The controller is forced to minimize the tracking error under the condition that the control signal $\nu_{\{t:t+N-1\}}$ never exceeds the saturation value, effectively removing its influence. The reference signal is calculated by passing the input vector $u_{\{t:t+N-1\}}$ through the linear system with the desired response. In this example the desired linear system is the same as the LTI part of the Hammerstein model. Thus, the whole problem can be formulated in the tracking error space $e_\tau = y_\tau - y_\tau^{ref} = C(x_\tau - x_\tau^{ref}) = C\tilde{x}_\tau$, where $\tilde{x}_\tau$ is the system state vector error between the desired LTI and the Hammerstein model. It is assumed that in every horizon the initial state error $\tilde{x}_t$ is a known vector constant.

Note that the control $\nu_\tau = 0$ for all τ is a strictly feasible point for the problem (5.27). This means that the solver could always return some solution, possibly achieving poor reference tracking performance. This justifies our strict feasibility assumptions in the previous section. In other words, all the steps of the QP solve algorithm will be able to execute regardless of the parameters in (5.27). Thus, no special handling of the infeasibility is necessary for these MPC problems.

In Section 5.1 we hinted that for MPC applications of interest the QP solver need not be capable of handling equality constraints. The first step in our reformulation of the problem in (5.27) is to remove the equality constraints. This is achieved by finding a closed form solution for $\tilde{x}_{t+k}$ and e_{t+k} [17,22].

We do this inductively, calculating

$$\begin{aligned}
\tilde{x}_{t+1} &= A\tilde{x}_t + B(\nu_t - u_t), \\
\tilde{x}_{t+2} &= A\tilde{x}_{t+1} + B(\nu_{t+1} - u_{t+1}) \\
&= A(A\tilde{x}_t + B(\nu_t - u_t)) + B(\nu_{t+1} - u_{t+1}) \\
&= A^2\tilde{x}_t + AB(\nu_t - u_t) + B(\nu_{t+1} - u_{t+1}), \\
&\vdots \\
\tilde{x}_{t+k} &= A^k\tilde{x}_t + \sum_{j=0}^{k-1} A^{k-j-1}B(\nu_{t+j} - u_{t+j}).
\end{aligned} \tag{5.28}$$

If we denote $v_{\{i1:i2\}} = (v_{i2}, \ldots, v_{i1})^T$, for any vector v and two integer indices $i1$ and $i2$, we can write this in the compact form

$$\begin{aligned} e_{\{t+1:t+N\}} = \Bigg(\begin{pmatrix} CA^N \\ \vdots \\ CA \end{pmatrix} \tilde{x}_t \quad - \begin{pmatrix} CB & \ldots & CA^{N-1}B \\ & \ddots & \vdots \\ & & CB \end{pmatrix} u_{\{t:t+N-1\}} \Bigg) \\ + \begin{pmatrix} CB & \ldots & CA^{N-1}B \\ & \ddots & \vdots \\ & & CB \end{pmatrix} \nu_{\{t:t+N-1\}}. \end{aligned} \tag{5.29}$$

Similarly, we can write the expression for the final state in a compact form

$$\begin{aligned} \tilde{x}_{t+N} = \Big(A^N \tilde{x}_t \quad - \Big(B \quad AB \quad \cdots \quad A^{N-1}B \Big) u_{\{t:t+N-1\}} \Big) \\ + \Big(B \quad AB \quad \cdots \quad A^{N-1}B \Big) \nu_{\{t:t+N-1\}}. \end{aligned} \tag{5.30}$$

We can observe that both the error vector $e_{\{t+1:t+N\}}$ and the final state error $\tilde{x}_{t+N}$ are affine functions of control variable $\nu_{\{t:t+N\}}$. The same holds for the first order difference of the control variable

$$\Delta\nu_{\{t:t+N-1\}} = \begin{pmatrix} 1 & -1 & 0 & \ldots & 0 \\ & 1 & -1 & & \vdots \\ & & \ddots & & -1 \\ & & & & 1 \end{pmatrix} \nu_{\{t:t+N-1\}} - \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \nu_{t-1} \end{pmatrix}, \tag{5.31}$$

often added to the MPC objective to smooth the control variable [3, 17].

From these observations we can reformulate the original MPC in (5.27) as

$$\begin{aligned} \min \quad & ||e_{\{t+1:t+N\}}||_I^2 + ||\tilde{x}_{t+N}||_P^2 \\ \text{s.t.} \quad & |\nu_\tau| \leq 1, \ \tau = \{t, \ldots, t+N-1\}. \end{aligned} \tag{5.32}$$

Note that in this formulation only the control moves are considered the optimization variables. Furthermore, we see that the objective is always a sum of weighted quadratic function of some affine mapping of the optimization variables. To derive the final, and general, packing of this MPC variant into desired QP form we can consider one such quadratic function.

Dropping the subscripts for legibility we can write

$$\begin{aligned} ||H\nu + r||_W^2 &= (H\nu + r)^T W (H\nu + r) \\ &= \nu^T (H^T W H)\nu + 2(r^T W H)\nu + r^T W r. \end{aligned} \tag{5.33}$$

Noting that the scaling and the constant offset do not change the argument where the optimum is achieved, we can write te final form of objective components

$$\frac{1}{2}\nu^T (H^T W H)\nu + (r^T W H)\nu, \tag{5.34}$$

exactly the form assumed in QP formulation (5.4) if we put $Q = H^T W H$ and $q = H^T W^T r$. Using this form and the fact that all the components would be scalled by the same factor we can write the objective of (5.32) as

$$\frac{1}{2}\nu_{\{t:t+N-1\}}^T (H_e^T H_e + H_{\tilde{x}}^T P H_{\tilde{x}})\nu_{\{t:t+N-1\}} + (r_e^T H_e + r_{\tilde{x}}^T P H_{\tilde{x}})\nu_{\{t:t+N-1\}}, \tag{5.35}$$

where we read $H_e, r_e, H_{\tilde{x}}, r_{\tilde{x}}$ by comparing $e_{\{t+1:t+N\}} = H_e \nu_{\{t:t+N-1\}} + r_e$ with (5.29) and $\tilde{x}_{t+N} = H_{\tilde{x}} \nu_{\{t:t+N-1\}} + r_{\tilde{x}}$ with (5.30). Again, we see the objective form suitable for our QP solver by putting $Q = H_e^T H_e + H_{\tilde{x}}^T P H_{\tilde{x}}$ and $q^T = r_e^T H_e + r_{\tilde{x}}^T P H_{\tilde{x}}$.

Note that r_e is not a compile time constant, but depends on $u_{\{t:t+N-1\}}$ while Q is fully known at compile time. The compiler will recognize this and will, for example, precompute the LDL^T decomposition needed for the QP initialization step. It will leave the q partially computed, postponing the final evaluation for run-time when the input signal over the horizon is known. While the compiler performs some optimizations (e.g. constant folding, common sub-expression elimination, etc.) the results depend strongly on the actual formulation. Further discussion and concrete examples regarding performance of various equivalant formulations can be found in Chapter 5.

Finally, we deal with the remaining, inequality, constraints in (5.32). A standard trick in this situation is to replace the absolute value constraints $|\nu_\tau| \leq 1$ with double inequaliy $-1 \leq \nu_\tau \leq 1$. Equivalently, we can write

$$\begin{pmatrix} I \\ -I \end{pmatrix} \nu_{\{t:t+N-1\}} \leq \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{5.36}$$

understanding that right-hand side constants must be of appropriate size.

With this change we arrive to our final MPC formulation

$$\begin{aligned} \min \quad & \frac{1}{2}\nu_{\{t:t+N-1\}}^T(H_e^T H_e + H_{\tilde{x}}^T P H_{\tilde{x}})\nu_{\{t:t+N-1\}} \\ & + (r_e^T H_e + r_{\tilde{x}}^T P H_{\tilde{x}})\nu_{\{t:t+N-1\}} \\ \text{s.t.} \quad & \begin{pmatrix} I \\ -I \end{pmatrix} \nu_{\{t:t+N-1\}} + s = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, s \geq 0. \end{aligned} \tag{5.37}$$

This MPC formulation follows the exact form shown in (5.2) and can be evaluated using the method presented in the previous section.

5.2.2 MPC: Constrained Reference Tracking

A second variant of MPC controller is developed for direct comparison with [7, 17, 22, 32]. While they all implement a variant of MPC control, in their formulation the system error prediction horizon N_p and the forcing/control horizon N_c are different. Mostly for managing the computation complexity, it seems. Also, the objective in this case is given directly as a reference to be tracked at the output. This should be compared to the indirect reference specification in the previous section where the reference was the response of an LTI to an input sequence.

We will present packing of the most general implementation, given in [22], and specialize for other cases. In this example, the system under control, i.e. the plant, is modeled as an LTI in SS form (5.26). The exact MPC formulation we are treating in this section is given by

$$\begin{aligned} \min \quad & \sum_{\tau=1}^{N_p} |y_{t+\tau} - y_{t+\tau}^{ref}|^2 + r \sum_{\tau=0}^{N_c} |\Delta\nu_{t+\tau}|^2 \\ \text{s.t.} \quad & ||\nu_{\{t:t+N_c-1\}}||_\infty \leq c_1 \\ & ||\Delta\nu_{\{t:t+N_c-1\}}||_\infty \leq c_2 \\ & ||(y_{\{1\}})_{\{t+1:t+N_p\}}||_\infty \leq c_3. \end{aligned} \tag{5.38}$$

Just like in the state error space we can derive

$$\begin{aligned} x_{t+1} &= Ax_t + Bf_t, \\ x_{t+2} &= Ax_{t+1} + Bf_{t+1} \\ &= A(Ax_t + Bf_t) + Bf_{t+1} \\ &= A^2x_t + ABf_t + Bf_{t+1}, \\ &\vdots \\ x_{t+k} &= A^kx_t + \sum_{j=0}^{k-1} A^{k-j-1}Bf_{t+j}. \end{aligned} \tag{5.39}$$

Remembering that in this example reference output is explicitly given we can write down the expression for the tracking error over the horizon, similar to (5.29), as

$$\begin{aligned} e_{\{t+1:t+N_p\}} = &\left(\begin{pmatrix} CA^{N_p} \\ \vdots \\ CA \end{pmatrix} x_t \quad - y^{ref}_{\{t+1:t+N_p\}} \right) \\ &+ \begin{pmatrix} CB & \dots & CA^{N_p-1}B \\ & \ddots & \vdots \\ & & CB \end{pmatrix} \nu_{\{t:t+N_p-1\}}. \end{aligned} \tag{5.40}$$

The control horizon N_c length is taken into account by putting $\nu^T_{\{t:t+N_p-1\}} = (0, \dots, 0, \nu^T_{\{t:t+N_c-1\}})$ for the final form of the error prediction over the horizon

$$\begin{aligned} e_{\{t+1:t+N_p\}} = &\left(\begin{pmatrix} CA^{N_p} \\ \vdots \\ CA \end{pmatrix} x_t \quad - y^{ref}_{\{t+1:t+N_p\}} \right) \\ &+ \begin{pmatrix} CA^{N_p-N_c}B & \dots & CA^{N_p-1}B \\ \vdots & & \vdots \\ & & \end{pmatrix} \nu_{\{t:t+N_c-1\}}, \end{aligned} \tag{5.41}$$

as given in [22, 32].

Note that (5.41) is an affine form, just like (5.29, 5.30, 5.31). Thus, everything we said

in the previous section holds. In particular, it is easy to see how the objective (5.38)

$$\min \; ||e_{\{t+1:t+N_p\}}||_P^2 + ||\Delta\nu_{\{t:t+N_c-1\}}||_R^2, \tag{5.42}$$

where $R = rI$, can be expressed in the form of (5.4) by using the (5.31, 5.33, 5.35). The same was done for (5.32).

To finish the formulation in [22] we need to show how additional constraints (e.g. constraints on output variable range, state range, differences of control or state variables, etc.) can be handeled. For example, in [22] the actual formulation of the MPC problem has constraints on the maximum absolute value of the control variable ν, its first difference $\Delta\nu$ and the first coordinate of the output vector y. In our notation we can write

$$\begin{aligned} \min \quad & ||e_{\{t+1:t+N_p\}}||_P^2 + ||\Delta\nu_{\{t:t+N_c-1\}}||_R^2 \\ \text{s.t.} \quad & ||\nu_{\{t:t+N_c-1\}}||_\infty \leq c_1 \\ & ||\Delta\nu_{\{t:t+N_c-1\}}||_\infty \leq c_2 \\ & ||(y_{\{1\}})_{\{t+1:t+N_p\}}||_\infty \leq c_3. \end{aligned} \tag{5.43}$$

In the previous equation, the constraint on the output vector, $y_{\{t+1:t+N_p\}}$, uses the same index set notation imposed earlier: the constraint is evaluated only for the first coordinate of the output vector, in this case.

First we note that all the constraints in (5.43) can be written as ∞-norm bounds of an affine map of the vector of optimization variables ν. Thus, constraint cases by showing how to bring a general constraint form

$$||G\nu + e||_\infty \leq h \tag{5.44}$$

into the form used in (5.4), where the derivations for general QP handling hold. This is simple if we follow the same steps as for (5.36). We can easily write

$$\begin{aligned} & ||G\nu + e||_\infty \leq h \\ \Leftrightarrow & -h \leq G\nu + e \leq h \\ \Leftrightarrow & \begin{pmatrix} G \\ -G \end{pmatrix} \nu \leq \begin{pmatrix} h - e \\ h + e \end{pmatrix}. \end{aligned} \tag{5.45}$$

With this simple transform, all the constraints in (5.43) can be written in the form the QP in (5.4) assumes. This is done by chosing the appropriate matrix G and vectors e and h for each constraint of interest.

5.3 Summary

This chapter presented algorithmic formulations of example QP and MPC problems. The formulations are developed both for execution on general purpose architectures, and with some customizations, on our templated-hardware architecture. We will see, in Chapter 6, that some of these ideas improve performance even on more commonly available architectures.

Transformations presented are outside of the reach of automatic optimizations by any current compiler. The reason is the understanding of the concepts involved and equivalence relations between them. This should be contrasted with the rule-based automatic local optimizations of the DFG outlined in Chapter 4.

The solutions we offer come in two flavors. For dealing with the purely numerical computation we resort to reducing the operation counts and DFG equivalence transforms exposing parallelism. This will become fully apparent in Chapter 6. When it comes to conditional (e.g. data dependant) processing we resort to indirection to hide decision making inside atomic actions offered by the processor. While the technique is not general it brings quite an advantage where applicable as it allows conditional processing without code branching.

Chapter 6

Results and Evaluation

For a successful technology, reality must take precedence
over public relations, for nature cannot be fooled.
—RICHARD FEYNMAN

In this chapter we finally show the end results of the work done until now. While previous chapters dealt with methods for template-based DSL construction at each level of hierarchy this one shows how we pick template parameters and how well the resulting instances perform. We also open discussion of the design tradeoffs we observe and how they affect implementation strategies we discussed in Chapter 1.

Due to our flexible design description, we will be able to see how processor configuration influences the algorithm performance. On one side a processor with more units can schedule a DFG in fewer cycles. However, such a processor is larger and it, as a rule, cannot achieve clock frequencies (cycle time) that a smaller processor could. Such conflicting trends, originating at different layers of the design stack, cannot be studied in traditional design flows. There, at the point when the full system design can be evaluated for performance, many parts of the design are firmly locked and completely inflexible. Changing them would result in many cascading changes throughout the system, due to serial dependencies we described in Chapter 1. However, our flexible design description spanning all the design layers allows us to observe these cross-layer tradeoffs and give answers to design questions that cannot even be asked in traditional design routes. A new set of tradeoffs arise, challenging basic assumptions of both algorithm-first and platform-first design approach.

We will show that achieving a given real-time computation latency can be possible only within a certain processor size window. Having a processor too big would make it run too

slow, while a processor too small would need too many cycles to execute the calculation. This is especially troublesome in platform-first design where the size of the processor is often set before the particular algorithm implementation is chosen.

In a similar vein, we will show that choosing algorithms based on the operation count can be a poor strategy if latency sensitive system is being implemented on a parallel architecture. This example should serve as a warning bell if an algorithm-first route is being considered for a design. We will see concrete examples of interesting, real-world, designs that benefit from optimizations that increase the number of nodes in the DFG, but break dependencies enabling better parallel scheduling.

Finally, we will show that the approach taken in this thesis results in superior performance and size of the large class of numerical computation designs when compared to recently published, hand-crafted implementations.

We start with simple linear algebra kernels and build up to the MPC algorithm we initially set out to implement efficiently in hardware-software codesign flow. Various alternative implementations on embedded and desktop processors are built for comparison and tradeoff study.

We present a detailed case study of several MPC implementations in the proposed design flow and compare to multiple implementation reports in recent years [6, 7, 17, 22]. The algorithmic details for each MPC variant are given in Chapter 5. In the interest of fair and clear comparison, in each case we implement the exact MPC variant treated in the work to which we compare. Where possible we analyze the performance gains by reporting performance of our system for different configurations.

While the results clearly show superior performance over all the recent implementations of MPC in every respect, we should keep in mind that this section is only interesting as a justification of the work done so far. It is the process of building a DSL in high-level language templates that brought us here. Using this approach we are able to produce a description of a whole region in a design space, instead of producing a single design instance as usually done. Furthermore, the process of constructing the parameterized model at each level of design hierarchy naturally leads to matching of template parameters across the stack, facilitating communication and discovery of best set of primitives for implementation.

All the reported timing and performance results are obtained by implementing the designs on a Xilinx Virtex-6 (ML605) prototyping board. The device on these boards has

-1 (slowest) speed grade, so slight further improvements in FPGA implementations are possible by utilizing a faster device. Reported clock speed is the highest Phase-Locked Loop (PLL) clock speed for the core processor we could impose and successfully place-and-route the design.

The run-times reported for our processor instances are calculated, due to static scheduling, by multiplying the schedule cycle-length with the cycle-time (i.e. the inverse of the achieved processor clock frequency).

All run-times reported for desktop and embedded processors were obtained by profiling the execution of our software implementations, as well as available implementations reported in other works [6]. To improve accuracy in solve-time measurement, we run the **MPC!** (**MPC!**) function for a large number of times to reduce any harness-induced overhead cost through averaging.

6.1 LDL^T decomposition

LDL^T is a simple linear algebra kernel that, in its basic form, does not require any conditional and data dependant processing. It is a simple and straightforward numerical evaluation that allows us to demonstrate our flow.

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 [53]

$$D_{ii} = A_{ii} - \sum_{k=0}^{i-1} L_{jk}^2 D_{kk} \tag{6.1}$$

$$L_{ij} = \frac{1}{D_{jj}} \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} D_{kk} L_{jk} \right) \tag{6.2}$$

Those two formulas can be used to create an algorithm to compute L and D. However, 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}. This is undesirable from both efficiency and accuracy standpoint.

Removing these multiplications by hand optimization results in the algorithm shown in Algorithm 6.1. This is the first algorithm we will explore.

Algorithm 6.1 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**
    $P_{ij} \leftarrow A_{ij} - \sum_{k=0}^{j-1} P_{ij} L_{jk}$ ▷ P_{ij} is the product of L_{ij} and D_{jj}
    $L_{ij} \leftarrow P_{ij}/D_{jj}$
  end for
end for

6.1.1 DFG Scheduling Modes: Throughput and Latency Limits

There are two limiting factors for our processor in this algorithm: the critical path, and the total number of each type of operation. The critical path for an $N \times N$ LDL^T decomposition, as reported by our compiler, is made up of $N-1$ adds, $N-2$ subtracts, $N-1$ multiplies, and $N-1$ divides. That means that one lower bound for algorithm execution time on our processor is the amount of time it takes for those operations to be performed sequentially, that is, the next one starts only when the previous one is completed. This lower bound increases linearly with the size of the problem.

An $N \times N$ LDL^T decomposition has $\frac{1}{6}(N^3 - N)$ additions/subtractions, $\frac{1}{6}(N^3 - N)$ multiplies, and $\frac{1}{2}(N^2 - N)$ divides. Considering the required throughput of the processor to finish the algorithm in a certain amount of time produces another lower bound for total execution time. The processor needs at least enough time to issue all of the operations in the algorithm, so it needs at least $\frac{1}{6}(N^3 - N) \div n_{\text{addsubs}}$ cycles to issue the add and subtract commands to the addsub where n_{addsubs} is the number of addsub units. Likewise, it needs at least $\frac{1}{6}(N^3 - N) \div n_{\text{muls}}$ cycles to issue all of the required multiplies, and at least $\frac{1}{2}(N^2 - N) \div n_{\text{divs}}$ cycles to issue all of the required divides. This lower bound increases quadratically or cubicly depending on which unit is constraining performance.

For a given processor configuration, the critical path will limit performance for smaller problem sizes, and the throughput will limit performance for larger problem sizes. This appears in Fig. 6-1 as a linear increase in latency for smaller problem sizes, and a cubic increase in latency for larger sizes.

As we hinted in the introduction to this chapter, different processor sizes achieve different

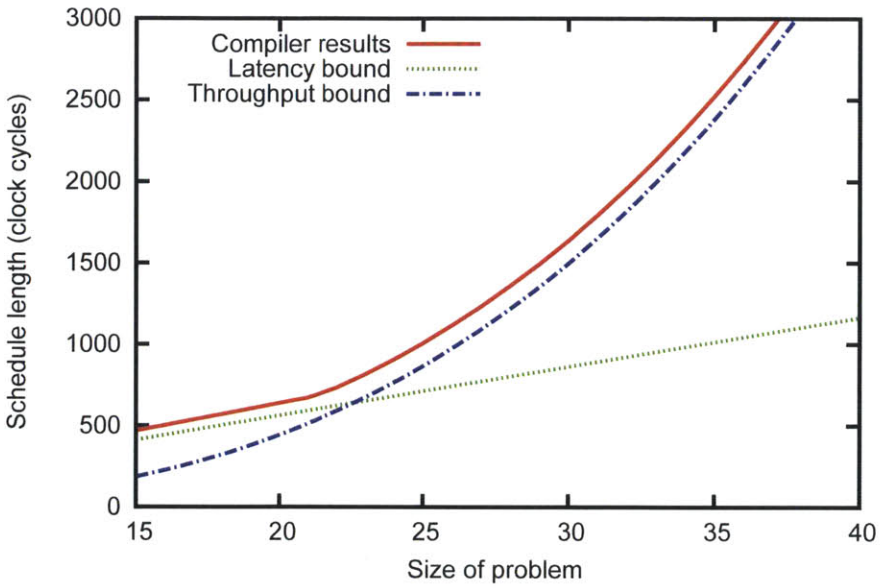


Figure 6-1: Performance bounds for LDL^T decomposition example

maximal clock frequencies (i.e. minimal cycle times). This means that slopes, when we observe the real time on the y-axis, of the execution time curves are different depending on the processor size on which they execute. This is clearly visible in Figure 6-2, where we analyze real time latency of LDL^T computation for the processor configurations shown in Table 6.1.

This observation gives us our first cross-layer system tradeoff.

6.1.2 Minimal and maximal processor size for latency optimization

To study the interaction between the processor size/configuration and the achievable latency of computation we synthesized three processor configurations. We target each processor configuration for execution of LDL^T decomposition of a different size. The parameters of each processor are shown in Table 6.1.

Choosing processor configurations, for a given algorithm, is a trial and error process. This is, mostly, a consequence of unpredictable nature of place-and-route implementation step in FPGA and ASIC toolchains. The only real solution, in view of inconsistencies in randomized algorithms in FPGA implementation process, at this point in time is evaluation of all (or most) possible configurations of the processor. This would, however, require significant computational resources and quite some time. Finding the optimal design is

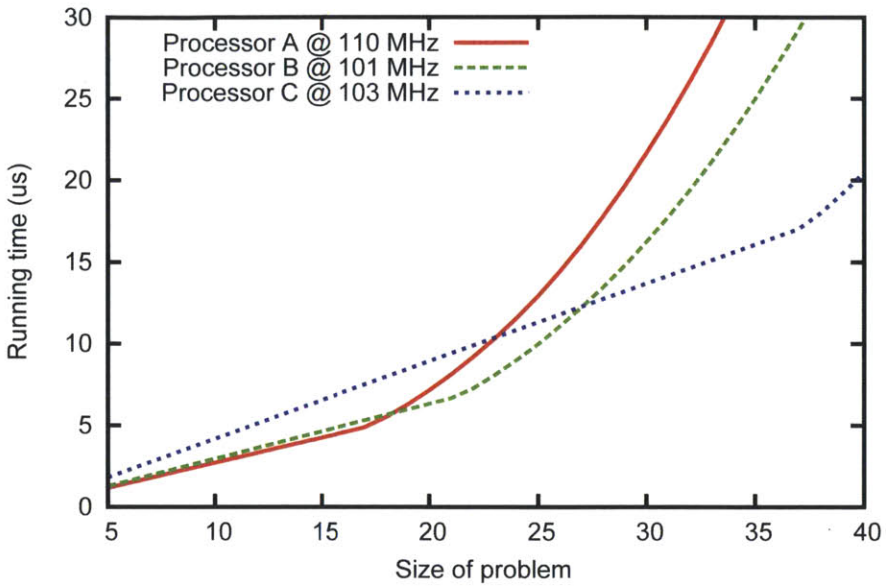


Figure 6-2: LDL^T latency results for custom accelerators

not our goal, however. We are satisfied with the possibility to locally tune designs from some initial point and improve its performance, in an intuitive and somewhat controllable manner, late in the design process.

We will use a type of local optimization heuristic to search for interesting instances. In particular, we will start with a processor that has one of each type of units with latency of one. As we saw in the previous section, such processor will be throughput limited for any reasonable problem size. Then, we start adding units to the processor, checking cycle lengths of schedules in every neighbouring configuration point. For this process, we consider any processor with at most one additional unit of each type a neighbor. As we do this, we typically observe diminishing returns from adding new units. This is because we are exhausting the finite parallelization opportunities in the computation DFG. We stop when the increase in resources does not improve cycle-time for more than a few percent.

It can be readily observed that processors B and C have the same performance for 27×27 matrix LDL^T. As we initially explained, this happens because larger processors are more challenging for implementation tools, generally achieving lower clock frequencies but executing more operations in parallel.

We can note that any processor with less resources than processor B, say processor A, would enter the latency-bounded mode of operation earlier than the processor B. This

Table 6.1: Processor parameters for studying processor size and latency on LDL^T decomposition example algorithm

		Processor A	Processor B	Processor C
Target LDL^T size		10×10	20×20	40×40
Addsubs	Number	1	2	7
	Latency	2	3	4
Muls	Number	1	3	7
	Latency	2	3	4
Divs	Number	1	1	1
	Latency	10	10	14
Comps	Number	1	1	1
	Latency	1	1	1
Crossbar latencies		1	2	2
Number of memories		6	16	35
Clock frequency (MHz)		111	101	103

would result in increased latency for the 27×27 decomposition, observable in the plot. Thus, to meet the latency of $12\mu s$, processor B is the processor with minimal resources that we can use. This is a direct consequence of inability to exploit parallelization opportunities of the DFG in question.

On the flip side, if we now analyze the intersection between latency curves for processors A and B we see that both achieve $\approx 6\mu s$ for 18×18 matrix decompositions. Any processor with more resources than processor B, for example processor C, would run at a lower clock frequency due to its size. Thus, due to hardware implementation constraints we also have the maximal resources we can use to meet certain latency of computation.

This is quite intuitive. As we increase the processor size we can exploit more parallelization opportunities in the DFG. However, the processors are becoming slower as we increase their size. The amount of additional parallel operations we can issue with each additional unit decreases and at some point the parallelization gain is completely offset by the processor speed degradation.

While the quantitative aspects of this analysis might depend on the template structure, or more generally the architecture, qualitatively this effect should always exist in system designs where many sub-systems must communicate. This reveals a potential weakness of traditional design flows: early decisions might make it infeasible to reach specified

performance, no matter in which direction we try to err when deciding on initial platform questions. Even worse, these problems cannot even be observed, much less quantified, when a flexible and reconfigurable design description is not available.

6.1.3 Performance comparisons

We looked at three sizes of LDL^T decompositions (10×10, 20×20, and 40×40) and chose processor parameters that are well suited for each size. The chosen processor parameters can be seen in Table 6.1.

For each set of parameters, the processor itself can be synthesized and tested. The chosen parameters are inserted into the processor template, and the templated is compiled into a bit file for the FPGA. The chosen algorithm is compiled for the processor, and then the bitfile and the schedule are loaded onto the board.

To verify the correct operation of the processor and the schedule, we designed a system to automatically load the board with test data, run the processor, read the results, and calculate the error. For the LDL^T decomposition test case, we load the board with a random symmetric indefinite matrix and start the processor. Once the decomposition is done on the processor, we read the calculated results $\widehat{L}$ and $\widehat{D}$ from the board. To make sure $\widehat{L}$ and $\widehat{D}$ are accurate and the board works, we look at the forward and backward errors. The forward error is $L - \widehat{L}$ and $D - \widehat{D}$ where L and D are the exact results, and the backward error is $A - \widehat{L}\widehat{D}\widehat{L}^T$ [51]. If these errors are sufficiently close to zero, then the processor is working. If the processor is working, then we know how many clock cycles it took complete the algorithm from the compiled code thanks to static scheduling. Fig. 6-2 shows the performance of each processor setup.

As points of comparison, we use an Intel Core i7-3930k @ 3.2 GHz, an Intel Xeon X5460 @ 3.16 GHz, an older AMD Athlon 64 Processor 3200+ @ 2.00 GHz, and a Broadcom BCM2835 with an ARM1176JZFS core @ 700 MHz found on a Raspberry Pi board. The test code is highly optimized custom code for LDL^T decomposition with minimal overhead. As a comparison for our custom LDL^T code, we also use LAPACK to perform the same problem on the Core i7 processor.

Note that, when it comes to latency of computation without conditionals, our FPGA hosted processor instances cannot catch up to the, quite impressive, i7 architecture. However, the gap between i7 execution times and our execution times is far smaller than the

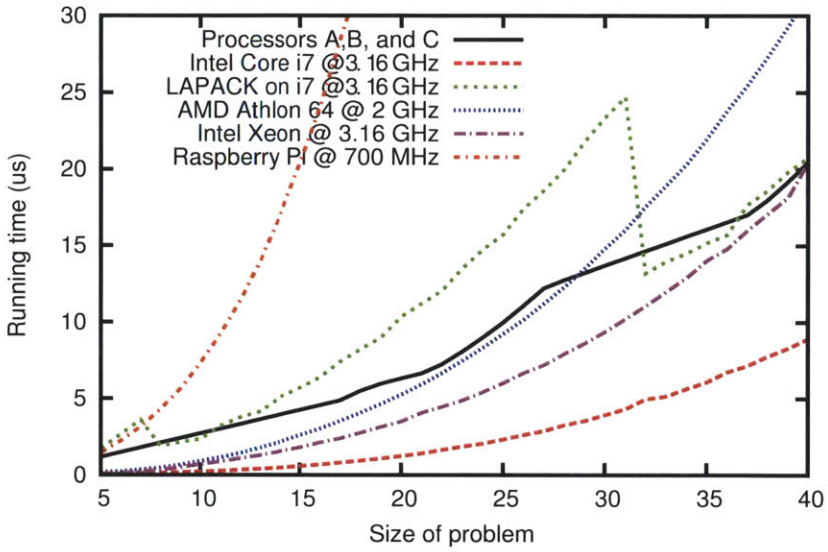


Figure 6-3: LDL^T latency results compared to other processors

gap between operating frequencies. While running at $\approx 30\times$ higher frequency, i7 achieves $\approx 4\times$ lower latency. This means that our processors extract quite a bit of parallelism from the problem DFG. On the flip side, when compared to the ARM core our processors show orders of magnitude improvements at similar power budgets.

To measure the power within the Core i7 processor we used the Intel Power Gadget. The power was measured for the Raspberry Pi board through an external power supply without any devices or displays connected to the Raspberry Pi board.

While idle, Intel i7, Raspberry Pi and FPGA, design B, based processors consume 3.05W, 1.57W and 1.05W, respectively. While running 20×20 LDL^T decompositions, the processors consume 17.97W, 1.73W and 3W, respectively.

6.2 MPC: Linearizing Pre-Equalizer

This section presents results of implementing the algorithm described in Section 5.2.1. The example is taken form [3]. The implementation methodology used in the original work was platform-first: their system takes problem descriptions in an abstract form, i.e. a language specifically designed for modeling certain optimization problems, and produces C code that can later be compiled for the appropriate architecture.

The results reported in [3] use The GNU Compiler Collection (GCC) and report timing

Table 6.2: Profiling results for CVXGEN [21] generated MPC

	gcc i7-3930k @ 3.2GHz	icc i7-3930k @ 3.2GHz	gcc RPi
Init (μs)	6	5	740
Iteration (μs)	15.5	13.5	790
Init + 10x Iter (μs)	161	140	8640
Energy/solve (mJ)	2.9	2.52	13.0

results on Core 2 Intel desktop processor. They do not give the number of iterations the code executes, but they do state the that it takes approximately $500\mu s$ for a solve.

We profile their code on Intel Core i7-3930k @ 3.2GHz with GCC and with Intel C Compiler (ICC) compilers and Raspberry Pi with an ARM-based processor. We do this by replicating the example in section "Linearizing Pre-Equalization" in [3] in CVXGEN and modifying the example test function to run the solve $100,000$ times with fixed number of iterations per solve. By changing the number of iterations we can reasonably accurately measure both the iteration time, showing up as the proportionality constant in these measurements, and the setup time, assumed to be equal to the translation of the linear curve representing run time as a function of the number of iterations. The results of our measurements are summarized in Table 6.2. We also calculate the *energy per solve* by measuring power on i7 and Raspberry Pi and multiplying it with the solve time.

At this point we should note that the algorithm used in [3] does not solve the IP iteration exactly. The linear system of equations arising is handled approximately, and then an iterative correction step can be applied. The profiling results in Table 6.2 use the algorithm settings from [3,6], with one iteration correction step. The details of the approach are outlined in [6,21]. In contrast, the algorithm we use, described in Chapter 5, does not need the correction step. Note that the authors in [6] state that most interesting MPC problems converge in 10 IP iterations with one iteration of the correction.

We solve the linear system arising in our QP formulation exactly. For this particular application both control and prediction horizon are the same $N_p = N_c = N = 16$. This means that the LDL^T decomposition in each IP iteration is done on a 16×16, dense symmetric positive definite matrix.

The exact computation we perform is described in Section 5.2.1. We analyze achievable performance on two processor instances and using two slightly different DFGs. Both

formulations produce the exact same outputs, given the same inputs, i.e. they are functionally equivalent. Processor settings and the results of our experiments can be seen in Table 6.3. We should note that the best result performs on par with, quite impressive, i7 processors at about 30× frequency penalty and several times less power consumption. Compared to an embedded processor, an ARM core available on popular Raspberry Pi boards, the custom solution shows orders of magnitude improvements. Integration into an ASIC, if economically justified, would improve these results several times over.

The first DFG, denoted DFG_1 in the table, we solve both the predictor and the corrector step by forward-backward substitution, using a single LDL^T decomposition. This is the standard way to solve the IP iteration. Analysis of this DFG reveals that both forward and backward substitution are very sequential operations. These are, essentially, recursive relations for solving lower and upper triangular systems of linear equations. The recursive quality makes this operation hard to parallelize.

An operation that is trivial to parallelize is matrix-vector multiplication. The fact that forward-backward substitution has a sequential chain-type DFG also means that during that phase of the algorithm very few units are utilized in the processor. Combining these two observations we speculate that a sufficiently large processor might have opportunity to use LDL^T decomposition to find inverse of the decomposed matrix while the forward-backward substitution is being used to calculate the predictor step. Then we use this inverse to calculate the corrector step, benefiting from the fact that this matrix-vector multiply is easy to parallelize. This computation is denoted as DFG_2.

Note that Processor Z and Processor S cannot execute DFG_2 nearly as fast as DFG_1. This is because the number of available operation units is low and the processors transitions into throughput-limited mode. In throughput limited execution the number of operations determines the time for evaluation, and we prefer a DFG with lower number of operations. This observation explains the use of operation count as the algorithm selection criterion: most embedded, and even desktop, processors have very few computation units available and work in throughput-limited mode all the time.

On the other hand, Processor L has sufficient number of units to exploit parallel opportunities in DFG_2. The resulting latency is about 20% lower than for DFG_1, on Processor L. Note that both Processor S and Processor L perform similarly on DFG_1. They achieve the same performance but in different modes of operation, as we discussed in Section 6.1.

Table 6.3: MPC performance for different processor resources and DFGs

		Processor Z		Processor S		Processor L	
Addsubs	Number	1		2		5	
	Latency	3		3		2	
Muls	Number	1		2		5	
	Latency	3		3		2	
Divs	Number	1		1		1	
	Latency	10		10		10	
Comps	Number	1		1		1	
	Latency	1		1		1	
Crossbar latencies		1		1		2	
Number of memories		8		18		25	
Clock frequency (MHz)		140		125		108	
DFG		DFG_1	DFG_2	DFG_1	DFG_2	DFG_1	DFG_2
Init Cycle-time		763	763	547	547	518	518
Iteration Cycle-time		2241	4210	1548	2186	1358	1125
Iteration Op Count		4183	8468	4183	8468	4183	8468
Init + 10x Iter (μs)		165.5	306.2	123.98	179.1	130.5	108.9
Energy/solve (mJ)		N/A	N/A	0.33	0.48	0.44	0.47

In view of *utilization wall* causing *dark silicon*, an observation that stagnating power budgets and Moore's low exponential growth of transistor counts result in exponential decrease in the number of transistors that we can afford to switch at the maximum speed [54, 55], this is good news: hardware-software codesign can help us find high-performance solutions while avoiding throughput maximization as the main performance driver.

To test this MPC implementation we set up a hardware-in-the-loop experiment. As a reminder, the block diagram of the equalization setup is given in Figure 5-1. The LTI plant is simulated on a desktop computer, while the control actions are obtained from the processor implementation. The results of one such simulation is shown in Figure 6-4. In the figure, reference signal (ref) and the outputs from controllers implemented in C++ (mpc) and on our processor (proc) overlap completely. The output from the system without pre-distortion to ameliorate the effects of saturation (sat) exhibits poor tracking of the reference (ref).

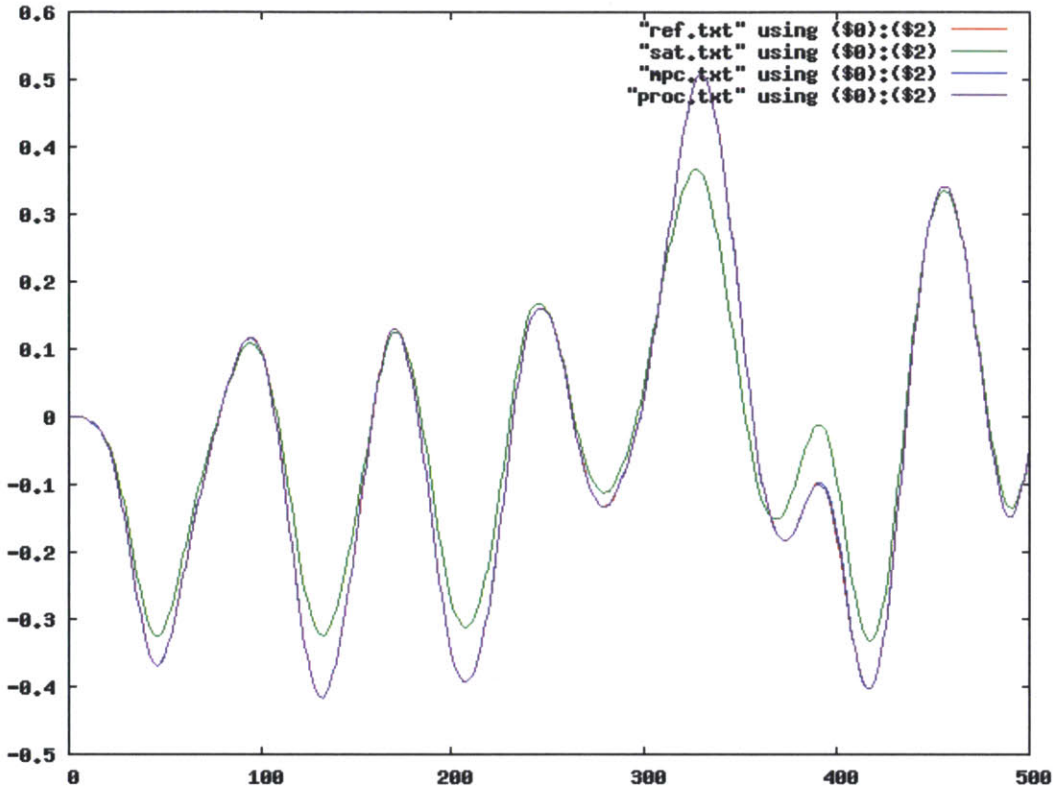


Figure 6-4: MPC as linearizing pre-equalizer

6.3 MPC: Constrained Reference Tracking

In this section we show and analyze results of applying our design flow to the MPC variant described in Section 5.2.2. As in the previous sections, we report post-place-and-route timing and test the resulting processor instances on an FPGA board.

As comparison points we use two hand-crafted MPC implementations given in [7, 17]. These formulations are computationally less demanding than the variant in the previous section as the control horizon is much shorter. For this example we will use $N_p = 20$ and $N_c = 3$, just like in cited works. In this case, the main latency contribution from the formulation outlined in Chapter 5 is LDL^T decomposition of a 3×3 matrix and the forward-backward substitution when solving the linear system of equations.

It should be noted that in [7], further explained in [32], the approach taken is the mixed design route discussed in Section 1.1.1. The platform is custom constructed based on the results of profiling of MPC load. The acceleration approach in [17] is algorithm-first with

Table 6.4: MPC performance for different processor pipeline structures and throughputs

		Processor X	Processor Y	Processor Z	Processor Q
Addsubs	Number	1	1	1	1
	Latency	1	2	3	4
Muls	Number	1	1	1	1
	Latency	1	2	3	4
Divs	Number	1	1	1	1
	Latency	7	7	10	12
Comps	Number	1	1	1	1
	Latency	1	1	1	1
Crossbar latencies		1	1	1	1
Number of memories		8	8	8	8
Init Cycle-time		190	203	223	243
Iteration Cycle-time		377	417	472	526
Clock frequency (MHz)		85	100	140	142
Init + 10x Iter (μs)		46.6	43.7	35.2	38.7

fixed-point number representation to speed up basic operations.

For the sake of easy comparison, we will present timing for algorithm initialization and ten interior point iterations. While results in [17, 32] do not necessarily run for ten full iterations, they do report per-iteration timing, so this comparison is possible. At this point we should note that our formulation uses single precision floating point numbers, allowing for more accurate calculation than either of the works we use for comparison. We compare at ten iterations as that would be a reasonable number of iterations for implementations with a fixed iteration number. At the end of the section we will show evidence that our formulation typically converges much more quickly.

6.3.1 Latency vs. Throughput: Diminishing returns

The configurations and performance of several processors used to implement the algorithm are shown in Table 6.4. Each successive processor instance increases pipeline depths of the operation units that participated in post-place-and-route critical paths. In this experiment, all the processors have the same number of computation units and only differ in the pipelining configuration. Once the critical paths move into instruction decoding logic, that

we have not parameterized, we start seeing very slow increase in frequency from additional pipelining of operation units and the overall performance starts decreasing.

We note that, in this example, an optimal pipeline structure and the corresponding, shallow, latency optimum can be observed. The effect parallels the size-latency relationship observed in Section 6.1. Interestingly, neither the minimum cycle-latency nor the minimum cycle time produce the minimum latency.

We should be careful when using the results in Table 6.4 for drawing conclusions about other design flows. All the examples in the table rely on the compiler for prioritization and scheduling of operations. Thus, even for unfavorable configurations the compiler will strive to minimize the schedule cycle length and the variation of the resulting latency might seem small. Speculatively, due to the compiler actions, we remain close to some local minimum in the design space. On the other hand, the gap between results from [7, 17] and our implementation indicates that without the compiler to guide design decisions we could end up quite far from this minimum.

In the algorithm-first strategy, where the execution order is implicitly specified by the algorithm partitioning, we are more prone to poor performance if the microarchitecture is not properly chosen. In the platform-first strategy compiler tools can help with software optimization for latency performance. However, the results in Table 6.4 show that tuning of the hardware platform could bring us between 10% and 20% in latency performance.

In Figure 6-5 we perform the test from [7], comparing our processor implementation to our double-precision floating-point C++ implementation of the MPC algorithm from [7,17]. In the figure, the MPC controller is repositioning a second order system after a step change on the reference input. In the tests, our formulation converges to surrogate duality gap $\approx 10^{-6}$ in at most five iterations for every control sample and disturbance profile we tried. The report in [17] fixes the number of iterations to 4, but does not report the duality gap.

6.3.2 Performance comparison

For comparison, the work in [7] reports, and is quoted in [17], $450\mu s$/iteration in an FPGA design achieving $50MHz$ clock frequency. The implementation needs more than $4.5ms$ for 10 iterations and initialization. The result can be significantly improved by different system partitioning and using fixed-point arithmetic to $18\mu s$ initialization and $28\mu s$/iteration in a $20MHz$ FPGA design [17]. The result is $298\mu s$ solve time for initialization and ten

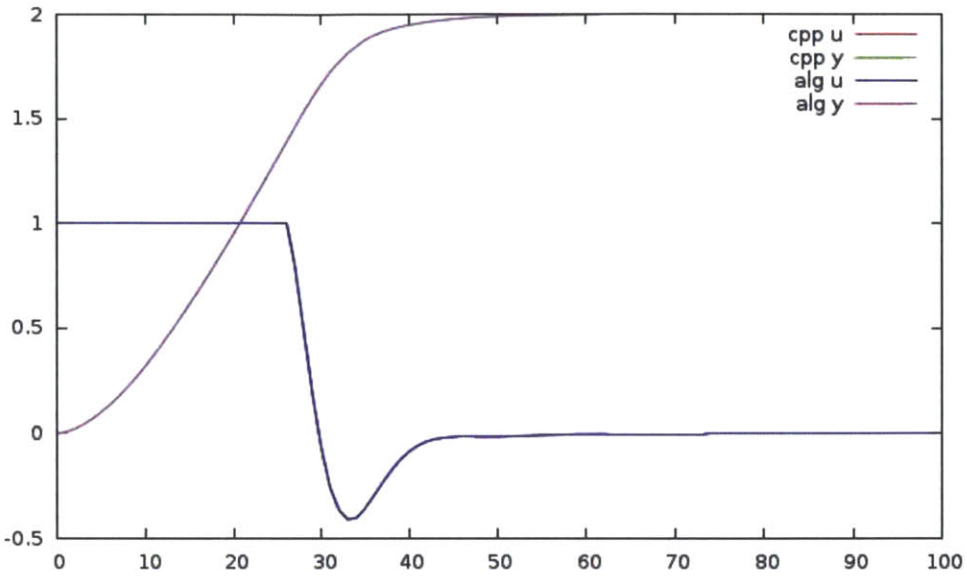


Figure 6-5: Control (u) and output (y) signal for step reference when controlled by an MPC implemented in C++ and on our processor

iterations of the interior point algorithm.

We should compare that to our best result of $\approx 35\mu s$ for the same solve. It is very interesting to note that even if run at $20MHz$, the cycle frequency used in [17], every processor in Table 6.4 would outperform the competing solution. The most logical choice would be to run Processor X, as the lowest cycle-latency processor for the calculation, and that would reult in the solve time of $\approx 200\mu s$, a significant improvement over the solution in [17]. This is a clear win for the compiler-centric design. It allows for algorithmic explorations that are impossible to perform by hand, and results in a better algorithm form that offers more parallelization opportunities.

Power is not reported for designs in [7, 17], but [17] does report resource utilization. Resource utilization comparison to our Processor Y can be seen in Table 6.5.

The design we achieve is not only much faster, but more efficient in resource utilization in almost every category. This is a great example to see how complex the space we have to navigate is: without proper tools to illuminate the path, we can easily get overwhelmed by the incidental complexity of HDLs and low-level implementation details.

Both designs we use for comparison report impressive performance improvements over other solutions. However, they deliver an instance, with a lot of custom, hand-crafted

Table 6.5: Resource utilization comparison between our design and a hand-crafted MPC implementation

FPGA resource usage comparison		
resource	[17]	Processor Z
Occupied slices	8,040	3,572
Slice registers	10,704	6,924
Slice LUTs	20,923	8,665
Block RAMs	16	68
DSP	96	3

design and custom number representations (in both cases less than 32 bits per value). On the other hand, our design instances for this comparison were generated from the same code as all the previous examples. Furthermore, we use single-precision floating-point number representation for which numerical analysis exists and guarantees on performance can be given.

The higher utilization of block Random Access Memory (RAM) resources is inherent for the design approach we took. The template described in Chapter 3 is programmable and the majority of our RAM utilization is from code memory in operation units. A small part of RAM resources is utilized for data storage units. It is likely that any design with requirements for run-time programmability would require more RAM resources than a customized, hand crafted solution with hard-coded execution flow and state machines, such as [17, 32].

On the flip side, all data-path related resources, i.e. logic, registers and DSP blocks, have much better utilization in our design. The reason for this is finer granularity of our design than in [17]. Due to this, software pipelining [38], happening in the compiler, can reuse the same resource, say a multiplier, for both LDL^T decomposition and, later on, line search or current solution update. Contrasted with this is the approach usually taken in algorithm-first designs, such as [16, 17] where an algorithm is parsed into blocks and each block is implemented as a separate hardware module. In such approach resource sharing is much harder to achieve and every block, usually, ends up having its own computation resources without possibility of reuse.

6.4 Summary

In this chapter we validated our design flow by presenting evidence that accelerators constructed using this flow outperform MPC solutions reported in recent literature. The potential for design reuse that our flow offers is evident if we remember that all particular implementations we test for comparison with other works are instantiated from the same set of templates described in this thesis. This is in sharp contrast to most other works on this topic where single instances are usually reported.

Going further than just delivering well performing design instances, we exploit the flexible structure of our design to look at cross-layer design tradeoffs. We show examples of real designs whose tradeoffs would be hard, if not impossible, to capture in any traditional design flow.

As we saw in Section 6.1, platform-first approach could have problems determining the hardware size for implementation as erring either on the side of too few resources at higher speed or too many at lower could result in failure to meet latency requirements. On the other hand, as we see from the examples in Section 6.2, the algorithm-first route can fail when parallel architectures are available. Furthermore, the results confirm that latency can be optimized at a different design point than throughput. Thus, latency minimization might need different design than throughput maximization, where simple pipelining tends to work well. Finally, comparison of results in Section 6.2 and Section 6.3 suggest that significant insight into computation process, rather than simple local rebalancing of DFG might be needed for best latency results.

This is the true reason for postponing design decisions through templating: capturing tradeoffs and interactions between algorithm, software and hardware components and design decisions at each level.

Chapter 7

Conclusions and Future Directions

If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.
—Guy L. Steele Jr.

This thesis outlines the design and verification of an alternative development infrastructure for construction of numerical accelerators for latency-critical applications in embedded and other tightly constrained environments.

The main idea of the framework is to minimize the number of immutable, a priori decisions in the design. At the same time, we want to keep full control of design at all the levels of the hierarchy. Our main goal was to provide guidelines for construction of such a methodology. As one possible solution, we presented the details of a template-based design infrastructure leveraging the power of well supported hardware and software design languages.

Having shown how the infrastructure can be constructed, we provided several examples of designs achieved in the proposed design flow to verify the utility such approach. Interestingly, despite very simple and general architectural features of our chosen design template, they performed surprisingly well. Our implementations of a few variants of the popular and challenging MPC algorithm (a computationally demanding control strategy based on QP optimization), outperform the best designs in recent literature we could find. Furthermore, we have seen that despite severe clock-frequency penalty in FPGA implementation this flow can be used to find implementations that perform on par with the powerful desktop and server processors, at a fraction of their power consumption. These results validate the old

wisdom from the software world: the choice of algorithm for the implementation substrate is at least as important as low level optimizations.

The developed framework gives us the ability to make an informed decision. It does so by offering us tools to explore cross-layer tradeoffs, usually completely hidden or at least obfuscated by traditional design flows.

The price to pay for this design flow is its development complexity, which luckily does not impact the users. The reward from using it are insights into system-level tradeoffs and levels of effort reuse unmatched by any of the more restricted environments. Judging by our results for MPC controller implementations, the price is well worth it.

7.1 Extensions

Our aim in this work was to showcase a design flow that can be constructed from tools available to most hardware designers, and to evaluate the results achievable through such approach. Thus, only the micro-architectural optimizations related to computation were performed. As a consequence the design could be extended in many key areas. We mention a few that we find most interesting.

Processor template was chosen for simplicity of hardware description and flexibility of compiler design. It is unclear whether full flexibility in choosing the operand and destination memories is necessary for efficient scheduling of DFGs. It might be possible that a contract between compiler and processor could be established, increasing the complexity of compilation but reducing the complexity of the processor by reducing the size of the operand and result crossbars. As communication delays dominate the achievable cycle-period time for all our processor instances (communication-to-logic delay on every critical path involving SSB is $\approx 2:1$) reducing it would be a significant boost in performance. Furthermore, reducing the number of wires in the crossbar would reduce routing congestion that is major obstacle for implementing the current design. Furthermore, the algorithm designer could choose to use different number representations (e.g. 8-bit floating point numbers) or test novel fused units (e.g. mockup more complex fused unit than the common multiply-add) and quickly test the impact of such decisions by running them through our compiler. After confirming their utility he could propose additional units to the processor template to further test how they influence processor cycle speeds.

Memory subsystem used in the current proof-of-concept system is quite simplistic. All operation require a full read from shared, global memories and a store to those memories. No local memory storage exists in operation units. Thus, we cannot expect any benefits from data locality in scheduling, and the compiler does not pursue such optimizations. It is conceivable that a better, hierarchical mix of small local and large global memories would enable better packing and shorter interconnect delays in physical realization while giving more opportunities for optimization to the compiler.

Instruction set was implemented for simplicity. Instructions carry very low information content and quite a few have high number of unused bits. No compression of instruction stream was implemented, not even the most rudimentary one. Thus an unit that is idle for a while will still load a NOP instruction from its code memory on every cycle. Furthermore, code memories cannot be repopulated while the processor is running. This severely limits schedule sizes we can run per iteration.

While all the mentioned extensions have potential to boost the performance, these are essentially implementation improvements. Big challenges are still in the design flow.

7.2 Challenges

One of the biggest challenges in this design flow is the complexity of the infrastructure needed for adoption of this approach. The solution spans various environment and languages, using advanced features of each.

While on the software side it is easy to find languages with great metaprogramming facilities, languages for software design do a very poor job of describing hardware in a way that gives full control of the synthesized circuits to the designer.

On the flip side, advanced hardware design languages, such as BSV, offer a lot of tools for exact definition of hardware. Unfortunately, the metaprogramming facilities exist in a very restricted form. They were, clearly, an afterthought for the language designers and one has to resort to all kinds of tricks to write a template such as ours in this environment. A full blown template, or even better macro, system would be very useful in efficiently achieving generalized circuit descriptions we need.

As a goal, we should seek to develop a unified development, platform, where both software and hardware metaprogramming could be done within a homogeneous environment,

with no need for a priori specification of the implementation substrate for any component of the system. Judging from the current state of hardware-software codesign landscape, we have some work to do.

Appendix A

Flow Mechanics

In Listing A.1 we show a simple program for calculating L_2 norm of a vector. The program first defines the **norm** template, tests it in C++ and finally compiles for execution on our processor with specified configuration. The source is extensively commented and easy to follow.

At the beginning of the source file we define the template we want to test. Then, in the **main()**, we first do a quick test of the template using **float** type, followed by declaration of the leafs/sources (i.e. input variables) of the graph, designated as **in[10]**, a 10 element array of **graphMaker** objects. Then we call the function template **norm** to construct the initial DFG. By configuring the processor, using **setProcessor**, we indicate scheduling and memory assignment constraints for the compiler. Finally, we configure optimizations and call **compile()**.

```
1  #include <iostream>
2
3  #include "rsh.hpp"
4
5  using namespace std;
6
7  template <class T>
8  T norm( T* in, int n ) {
9      T tmp = 0;
10     for( int i = 0 ; i < n ; i++ ) {
11         tmp += in[i] * in[i];
12     }
13     return sqrt( tmp );
```

```
14 }
15
16 int main (int argc, char* argv[]) {
17     compileJob *cj = new compileJob();
18     processor *p = NULL;
19
20     // function template used with floats to test simple case of algorithm
21     float test_vector[3] = {3.0, -4.0, 12.0};
22     cout << "norm( test_vector, 3 ) = " << norm( test_vector, 3 ) << endl;
23
24     // enable front-end optimizations that are performed by the graphMaker
           class
25     graphMaker::optimize = true;
26
27     // initialize inputs and name them using name_array function
28     graphMaker in[10];
29     name_array( in, 10 );
30
31     // call the templated function that generates the graph for the
           computation
32     graphMaker result = norm( in, 10 );
33
34     // rename the result so it is designated as a target of the computation
35     result.renameNode( "Norm" );
36
37     // tell the compileJob to use the graph generated by the graphMaker class
38     cj->useGraphMaker();
39
40     // output the initial graph of the algorithm to a dot file
41     cj->getSchedGraph()->writeDotFile("norm_graph_initial.dot");
42
43     // specify the processor parameters and pass the processor to the
           compileJob
44     p = new processor( 2, 3, 2, 2, 1, 10, 1, 18, 1, 1 );
45     p->setXBarDelays( 1, 1 );
46     p->setNumMems( 13 );
47     cj->setProcessor( p );
48
49     // set the optimization level and compile
```

```
50      cj->setOptimizationLevel( 2 );
51      cj->compile();
52      // results of the compilation are stored in text files within folders in
            the current directory
53
54      // additional results can be printed for quicker access
55      cout << cj->results() << endl;
56
57      // information about the DFG
58      cout << "latency bound = " << cj->getSchedGraph()->getLatencyBound() <<
            endl;
59      cout << "throughput bound = " << cj->getSchedGraph()->getThroughputBound()
             << endl;
60
61      // output the final graph of the algorithm to a dot file
62      cj->getSchedGraph()->writeDotFile("norm_graph_final.dot");
63
64      return 0;
65  }
```

Listing A.1: Simple program for testing `norm` template and compiling it to a processor configuration

Compiling this program with C++ produces an executable. Running the executable will perform compilation of the computation defined in **norm** template. The outputs from this program are schedule files, one per processor unit, that can be loaded into FPGA as soon as the corresponding processor instance bitfile is programmed into it.

Our numerical compilation process also produces two *Graphviz* files, *norm_graph_initial.dot* and *norm_graph_final.dot*, that can be rendered to produce images showing the computation graph before and after optimizations are performed. We show these images in Figures A-1 and A-2. Note that the final graph shows the schedule and memory assignments.

The described flow of processor template instantiation, low-level (by the compiler) and high-level (by human designer changing the **norm** template) graph optimizations is our main tool for finding efficient and high-performance configurations from the algorithmic level. We use graph renderings and statistics, printed by the compiler when it runs, to diagnose bottlenecks and make optimization decisions.

Finally, good processor candidates are implemented and tested for highest clock fre-

quency to relate the cycle-latency of computation to the real-time latency.

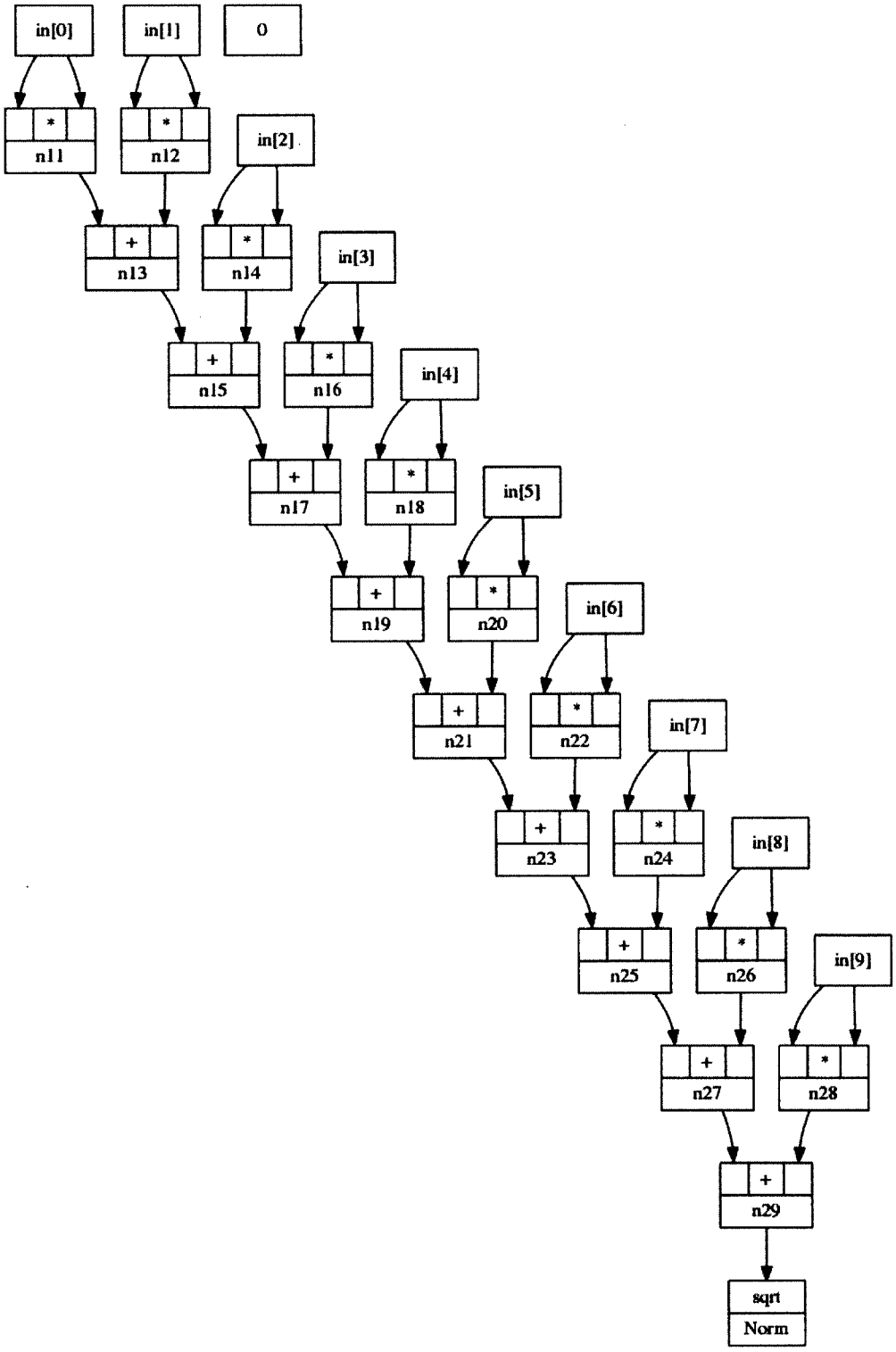


Figure A-1: Initial computation graph as produced by the **norm** template

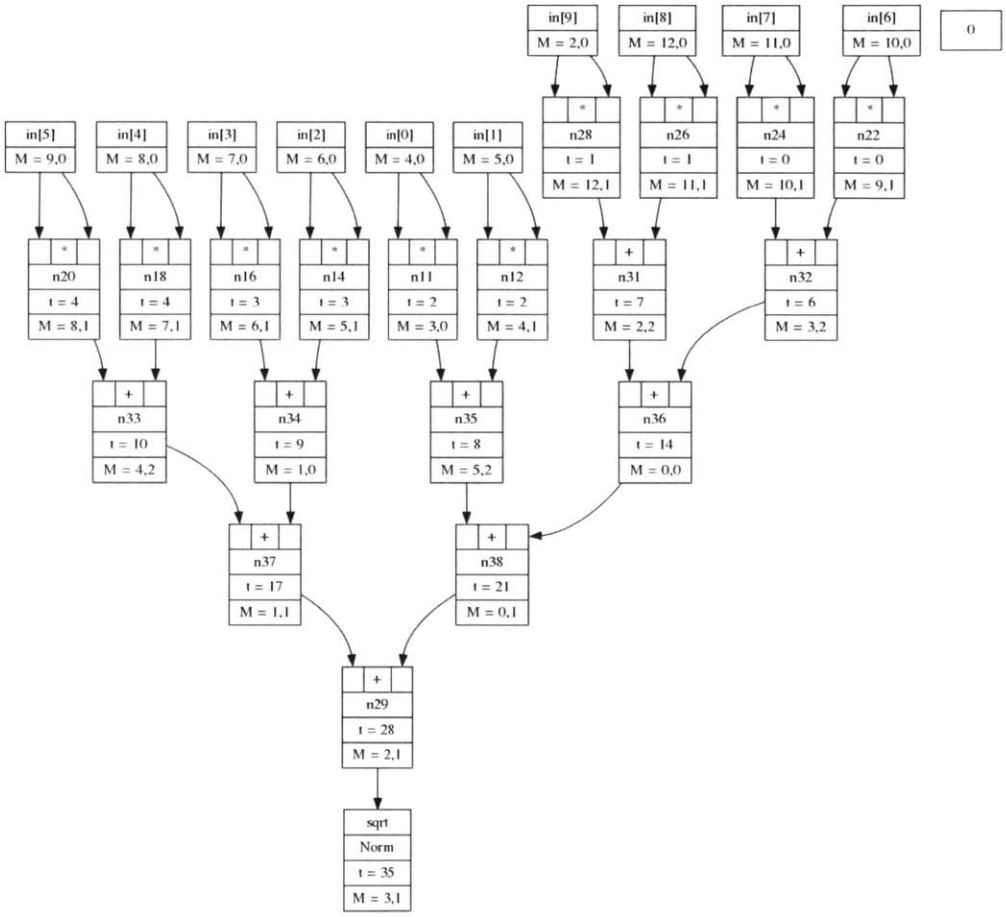


Figure A-2: Final computation graph after optimization, scheduling and memory assignment

Bibliography

[1] A. Kirmani, A. Colaço, F. N. C. Wong, and V. K. Goyal, "CoDAC: A compressive depth acquisition camera framework," *Acoustics, Speech and Signal Processing*, pp. 3809–3812, 2012.

[2] A. Bemporad and M. Morari, "Robust model predictive control: A survey," *Robustness in identification and control*, 1999.

[3] J. Mattingley and S. Boyd, "Real-time convex optimization in signal processing," *Signal Processing Magazine, IEEE*, no. May, pp. 50–61, 2010.

[4] A. Bry, A. Bachrach, and N. Roy, "State estimation for aggressive flight in gps-denied environments using onboard sensing," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2012)*, (St Paul, MN), 2012.

[5] J. W. Roberts, R. Cory, and R. Tedrake, "On the controllability of fixed-wing perching," *2009 American Control Conference*, pp. 2018–2023, 2009.

[6] J. Mattingley, Y. Wang, and S. Boyd, "Code generation for receding horizon control," *2010 IEEE International Symposium on Computer-Aided Control System Design*, pp. 985–992, Sept. 2010.

[7] L. Bleris, P. Vouzis, M. Arnold, and M. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," *2006 American Control Conference*, p. 6 pp., 2006.

[8] S. Bayraktar and E. Feron, "Experiments with small helicopter automated landings at unusual attitudes," *arXiv preprint arXiv:0709.1744*, pp. 1–20, 2007.

[9] R. Cory and R. Tedrake, "Experiments in fixed-wing UAV perching," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Reston, Virigina), pp. 1–12, American Institute of Aeronautics and Astronautics, Aug. 2008.

[10] F. Chen, A. P. Chandrakasan, and V. Stojanovic, "Design and analysis of a hardware-efficient compressed sensing architecture for data compression in wireless sensors," *J. Solid-State Circuits*, vol. 47, no. 3, pp. 744–756, 2012.

[11] M. Pajic and R. Mangharam, "Topological Conditions for In-Network Stabilization of Dynamical Systems," *IEEE Journal on Selected Areas in Communications*, pp. 1–14, 2013.

[12] J. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, 1992.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[14] P. Koopman, "Design constraints on embedded real time control systems," 1990.

[15] H. J. Ferreau, T. Kraus, M. Vukov, W. Saeys, and M. Diehl, "High-speed moving horizon estimation based on automatic code generation," *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 687–692, Dec. 2012.

[16] Y. Shoukry, M. El-Kharashi, and S. Hammad, "MPC-On-Chip: An Embedded GPC Coprocessor for Automotive Active Suspension Systems," *Embedded Systems Letters, IEEE*, vol. 2, pp. 31–34, June 2010.

[17] K. Basterretxea and K. Benkrid, "Embedded high-speed Model Predictive Controller on a FPGA," *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 327–335, June 2011.

[18] N. Yang, D. Li, J. Zhang, and Y. Xi, "Model Predictive Control System Based on FPGA and A Case Study," in *World Congress*, pp. 9266–9271, 2011.

[19] T. Poggi and S. Trimboli, "Explicit hybrid model predictive control: discontinuous piecewise-affine approximation and FPGA implementation," *... of Automatic Control*, no. 2002, pp. 1350–1355, 2011.

[20] A. Wills, A. Mills, and B. Ninness, "FPGA Implementation of an Interior-Point Solution for Linear Model Predictive Control," *Preprints of the 18th IFAC World ...*, pp. 14527–14532, 2011.

[21] J. Mattingley and S. Boyd, "CVXGEN: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, pp. 1–27, Nov. 2011.

[22] K. Ling, B. Wu, and J. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," *Proc. 17th IFAC World ...*, pp. 15250–15255, 2008.

[23] J. L. Jerez and G. A. Constantinides, "Parallel MPC for Real-Time FPGA-based Implementation," in *Architecture*, pp. 1338–1343, 2011.

[24] D. Froß, J. Langer, and A. Froß, "Hardware implementation of a Particle Filter for location estimation," *Indoor Positioning and ...*, no. September, pp. 15–17, 2010.

[25] T. A. Johansen, W. Jackson, R. Schreiber, and P. Tø ndel, "Hardware Architecture Design for Explicit Model Predictive Control," no. 5, pp. 1924–1929, 2006.

[26] P. Vouzis, L. Bleris, M. Arnold, and M. Kothare, "A Custom-made Algorithm-Specific Processor for Model Predictive Control," *2006 IEEE International Symposium on Industrial Electronics*, pp. 228–233, July 2006.

[27] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed-solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72–76, 2010.

[28] S. S. Muchnick, *Advanced compiler design and implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[29] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, 2010.

[30] A. Solomatnikov, A. Firoozshahian, O. Shacham, Z. Asgar, M. Wachs, W. Qadeer, S. Richardson, and M. Horowitz, "Using a configurable processor generator for computer architecture prototyping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 358–369, 2009.

[31] G. Ezer, "Xtensa with user defined dsp coprocessor microarchitectures," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, pp. 335–342, 2000.

[32] P. Vouzis and M. Kothare, "A system-on-a-chip implementation for embedded real-time model predictive control," *Control Systems ...*, vol. 17, no. 5, pp. 1006–1017, 2009.

[33] S. Boyd and L. Vandenberghe, *Convex Optimization.* New York, NY, USA: Cambridge University Press, 2004.

[34] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 18-20, 1967, spring joint ...*, pp. 1–4, Dec. 1967.

[35] T. Hu, "Parallel sequencing and assembly line problems," *Operations research*, vol. 9, no. 6, pp. 841–848, 1961.

[36] R. P. Nix, J. J. O. Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduline Compiler," vol. 37, no. 8, 1988.

[37] C. Ramamoorthy, K. Chandy, and M. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," *Computers, IEEE Transactions on*, vol. 100, pp. 137–146, Feb. 1972.

[38] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," *SIGPLAN Not.*, vol. 23, pp. 318–328, June 1988.

[39] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein, "Software pipelining showdown: optimal vs. heuristic methods in a production compiler," *SIGPLAN Not.*, vol. 31, pp. 1–11, May 1996.

[40] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 26–36, ACM, 2010.

[41] G. R. Beck, D. W. L. Yen, and T. L. Anderson, "The cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, May 1993.

[42] C. Mead and L. Conway, *Introduction to VLSI systems.* 1980.

[43] *Bluespec SystemVerilog Reference Guide.* 2012.

[44] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 3rd Edition.* Morgan Kaufmann, 2003.

[45] *Lexical Analysis With Flex, for Flex 2.5.37.* The Flex Project, 2012.

[46] C. Donnelly and R. Stallman, *GNU Bison - The Yacc-compatible Parser Generator.* Boston, MA, USA: Free Software Foundation, 2012.

[47] P. Graham, *ANSI Common Lisp.* Upper Saddle River, NJ, USA: Prentice Hall Press, 1996.

[48] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language.," in *DAC* (P. Groeneveld, D. Sciuto, and S. Hassoun, eds.), pp. 1216–1225, ACM, 2012.

[49] A. C. Wright, "A Statically Scheduling Compiler for a Parameterized Numerical Accelerator," Master's thesis, Massachusetts Institute of Technology, 2013.

[50] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs.* Cambridge, MA, USA: MIT Press, 2nd ed., 1996.

[51] N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, second ed., 2002.

[52] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback Control Theory.* Prentice Hall Professional Technical Reference, 1991.

[53] G. W. Stewart, *Matrix Algorithms.* Society for Industrial and Applied Mathematics, 1998.

[54] M. B. Taylor, "Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 1131–1136, ACM, 2012.

[55] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," *SIGARCH Comput. Archit. News*, vol. 38, pp. 205–218, Mar. 2010.