A Behavioral Design Flow for Synthesis and Optimization of Asynchronous Systems

John B. Hansen

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2012

Approved by:

Montek Singh

Anselmo Lastra

Sanjoy Baruah

Luciano Lavagno

Michael Theobald

© 2012
John B. Hansen
ALL RIGHTS RESERVED

ABSTRACT

JOHN B. HANSEN: A Behavioral Design Flow for Synthesis and Optimization of Asynchronous Systems.
(Under the direction of Montek Singh.)

Asynchronous or clockless design is believed to hold the promise of alleviating many of the challenges currently facing microelectronic design. Distributing a high-speed clock signal across an entire chip is an increasing challenge, particularly as the number of transistors on chip continues to rise. With increasing heterogeneity in massively multi-core processors, the top-level system integration is already elastic in nature. Future computing technologies (*e.g.*, nano, quantum, etc.) are expected to have unpredictable timing as well. Therefore, asynchronous design techniques are gaining relevance in mainstream design. Unfortunately, the field of asynchronous design lacks mature design tools for creating large-scale, high-performance or energy-efficient systems.

This thesis attempts to fill the void by contributing a set of design methods and automated tools for synthesizing asynchronous systems from high-level specifications. In particular, this thesis provides methods and tools for: (i) generating high-speed pipelined implementations from behavioral specifications, (ii) sharing and scheduling resources to conserve area while providing high performance, and (iii) incorporating energy and power considerations into high-level design.

These methods are incorporated into a comprehensive design flow that provides a choice of synthesis paths to the designer, and a mechanism to explore the spectrum between them. The first path specifically targets the highest-performance implementations using data-driven pipelined circuits. The second path provides an alternative approach that targets low-area implementations, providing for optimal resource sharing and optimal scheduling techniques to achieve performance targets. Finally, the third

path through the design flow allows the entire spectrum between the two extremes to be explored. In particular, it is a hybrid approach that preserves a pipelined architecture but still allows sharing of resources. By varying performance targets, a wide range of designs can be realized. A variety of metrics are incorporated as constraints or cost functions: area, latency, cycle time, energy consumption, and peak power. There are several long-standing challenging problems in resource sharing, many of which have been solved optimally for the first time as part of the research for this dissertation.

Experimental results demonstrate the capability of the proposed design flow to quickly produce optimized specifications. By automating synthesis and optimization, this thesis shows that the designer effort necessary to produce a high-quality solution can be significantly reduced. It is hoped that this work provides a path towards more mature automation and design tools for asynchronous design.

ACKNOWLEDGMENTS

I could not have completed my thesis without the extraordinary support of many people. First of all, I would like to thank my advisor, Montek Singh, for not only his academic support, but personal support; he helped me perform at my best even at times when my motivation was lacking. It's hard to keep track of the countless times he spent late evenings (or early mornings) assisting me with papers and presentations. Thank you, Montek.

Next, I would like to thank my committee for all their help in completing this dissertation. Sanjoy Baruah, Anselmo Lastra, Luciano Lavagno, and Michael Theobald have all provided invaluable feedback on this work. I'd especially like to thank Luciano Lavagno and Michael Theobald for being willing to do so remotely, which I know was a very difficult task.

I would like to thank several others who did not serve on my committee but who have provided assistance along the way. Leandra Vicci and John Thomas have been extremely helpful over the years, particularly in aiding our group with the experimental EUCLID project, and allowing us to use the facilities at the MSL lab. I'd also like to thank John Poulton who served on my M.S. committee and helped evaluate some of my earlier research. On the administrative side, I'd like to thank Janet Jones, Tim Quigg, Jodie Turnbull, Dorothy Turner, and Missy Wood for everything they did to make sure I stayed on track, completed my requirements, and had enough funding to continue.

I would like to thank several funding sources for supporting my research, including DARPA for a grant under the DARPA CLASS (Clockless Logic Analysis, Synthesis and Systems) program, National Science Foundation (NSF) for grants CCF-0702712 and

OCI-1127361, and, of course, the Computer Science department for multiple teaching assistantship opportunities.

Outside of the realm of computer science, I would like to thank my friends for providing a welcome distraction from the stress of research. While this list cannot be fully enumerated, I would particularly like to mention James Culp, Gennette Gill, and Diane Losardo.

Finally, I would like to thank my family; I could never have completed this research without their support. This thesis is dedicated to the memory of my grandfather, John K. Hansen, who I miss greatly.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	Arithmetic and Logic Unit
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
CDFG	Control/Data-Flow Graph
DAG	Directed Acyclic Graph
$\frac{dE}{dL}$	Derivative of total energy consumption with respect to total latency
$\frac{de}{dt}$	Derivative of operation energy consumption with respect to time
DFG	Data-Flow Graph
DSE	Design-Space Exploration
FPGA	Field-Programmable Gate Array
GALS	Globally-Asynchronous Locally-Synchronous
HDL	Hardware Description Language
HLS	High-level Synthesis
NoC	Network-on-Chip
SoC	System-on-Chip

Chapter 1

Introduction

1.1 Motivation and Goals

1.1.1 Domain

Most of digital hardware today is clocked or synchronous. However, synchronous hardware design is facing significant challenges as we push for higher clock speeds and more complex chips. Aside from the incredible task of optimizing designs year after year, physical properties of circuits at the current scale and speed are becoming major roadblocks. For example, skew associated with high-fanout clock signals puts a great burden on the designer by increasing design time, transistor variability reduces the yield of chips at fabrication, and energy consumption at low process sizes and billions of transistors burdens the consumer (as well as the environment). Because of variability, designers must either slow chips by introducing large safety margins or contend with lower yields.

Especially as we shift more and more towards mobile hardware such as laptops and cell phones, consumer demands on chips are shifting; design processes should as well. We need chips with greater energy efficiency (for longer battery life) and better electromagnetic compatibility (*e.g.*, lower noise emission for chips on cell phones). At the same

time, we must improve designer efficiency by reducing design effort. Moore's law suggests that more transistors will become available, and, as a result, designer productivity must go up to produce more complex chips in the same time frame. Therefore, we need to be able to re-use components; they must be flexible and modular. Unfortunately, clocking interferes with re-usability due to global timing requirements.

Beyond conventional computing, emerging technologies are trending increasingly towards domains where global clocks become impractical. Multi-core and distributed systems, globally-asynchronous locally-synchronous systems (GALS), and network-on-chip (NoC) are examples where the top-level system integration is already becoming elastic in nature. Technologies even further out on the horizon, such as quantum and DNA-based computing are expected to have unpredictable timing as well, further highlighting the need for an alternate paradigm to global clocking.

Due to these demands, asynchronous or "clockless" design is emerging as a promising alternative to synchronous design with the potential of alleviating many of the next generation design challenges. Rather than relying on a clock to manage the flow of computation, a request and acknowledge handshake paradigm is used to control computation. As a result, managing large-scale clock distribution is avoided (improving energy efficiency). Asynchronous chips are robust to changes in voltage and temperature, more resistant to the side effects of process variation, and produce significantly less electromagnetic noise. Chips can also be designed with average case throughput in mind, rather than the worst case with clocking. Perhaps most important is that designs can be much more modular: rather than managing a deep clock tree, only local timing assumptions at a modules interface must typically be considered.

The research presented in this dissertation therefore targets the design of asynchronous systems, specifically high-level synthesis of custom chips (rather than conventional microprocessors). While the work in this dissertation targets asynchronous

ASICs, the research results produced may certainly be applicable to many other domains, from synchronous design to multi-core computing to distributed systems.

1.1.2 Objectives

Despite the significant advantages asynchronous design can provide, several challenges remain to be addressed before greater mainstream adoption. The primary challenge of asynchronous design is that the current design tools are much less mature than synchronous design tools. As a result, designers who have practiced synchronous design for several decades may not easily make the switch to an asynchronous paradigm. Therefore, the majority of the research in this dissertation is aimed at making asynchronous design easier by reducing designer effort and improving performance. This dissertation focuses on building a top-to-bottom design flow to address this challenge.

The objective of this work is to produce a fully-automated design flow that:

- boosts performance through a suite of optimizations, including: parallelization, pipelining, loop-pipelining, arithmetic decomposition and decoupling (including at the bit-level), and communication optimization,

- provides design-space exploration by performing the synthesis tasks of scheduling, allocation, and binding of shared resources in an automated fashion, and

- allows a whole spectrum of designs to be explored by varying constraints, with implementations ranging from highly pipelined to control-driven, as well as exploring the space in-between,

- optimizes for several metrics including area, latency, throughput, power, and energy.

1.1.3 Thesis Statement

Design-space exploration of asynchronous systems can be automated effectively in order to rapidly produce high-quality implementations with significantly reduced designer effort.

1.2 Past Approaches and Current Challenges

While several approaches have been previously proposed to target high-level synthesis, none have effectively traded off optimality, performance, and other performance metrics while simultaneously allowing for rapid, easy design.

Two of the most well-known synthesis tools for the design of asynchronous systems — Haste (Haste, 2008) and Balsa (Edwards and Bardsley, 2002; Bardsley and Edwards, 2000) — rely on syntax-directed translation of behavioral specifications. Produced circuits match the input specification one-to-one: every language construct in the specification is directly implemented as a distinct hardware object, all sequencing in the specification is preserved, and every arithmetic operation becomes a distinct arithmetic unit (no automated resource sharing). This paradigm allows for rapid design times; however, performance of produced circuits is quite low (*e.g.*, 10-100MHz for Haste) and span large areas on chip.

Research presented in (Nielsen, 2005; Nielsen et al., 2004; Nielsen et al., 2009; Jensen and Nielsen, 2007) leverages these existing Haste and Balsa flows to perform resource scheduling, allocation, and binding in an automated fashion. Their solution receives as input a high-level specification or control/data-flow graph (CDFG), and produces a resource-shared version in the original source language (either Haste or Balsa), with a target of minimizing area. However, their method does not target performance, and is restricted to the syntax-directed compilation approach of their back-end.

Several other synthesis approaches exist in the asynchronous domain. Budiu et al. (Budiu, 2003) introduced the approach of spatial computation, which compiles ANSI C specifications directly into hardware. However, this approach explicitly forbids resource sharing; each computation is given its own dedicated function unit. A recent approach by Gill (Gill, 2010) targets analysis and optimization of existing pipelined systems constructed in a hierarchical fashion, but cannot handle sharing of resources.

De-synchronization (Cortadella et al., 2006; Andrikos et al., 2007) is an entirely different approach in which existing *synchronous* tools are leveraged to create a synchronous netlist, which is later converted into an asynchronous version by removing the clock and replacing it with local asynchronous controllers. While this approach leverages the significant research behind mature synchronous design tools, replacing low-level clock signals with handshaking does not allow the designer to exploit system-level concurrency as well as a top-down asynchronous design approach.

Many well-known synchronous approaches exist that specifically target performance enhancement through optimization. A recent approach by Kondratyev et al. (Kondratyev et al., 2011) performs synthesis that targets high performance implementations; the authors' primary aim being feasible, real-world design-space exploration. The SPARK (Gupta et al., 2003) framework converts high-level specifications in C to VHDL, performing several powerful high-level optimizations such as parallelization and loop transformation in the synthesis process. AutoPilot/AutoESL (Coussy and Morawiec, 2008) is a proprietary solution for converting specifications written in C variants to FPGAs. These tools and methods specifically target the synchronous realm and therefore are not easily transferable to the synchronous realm; as I will show later, naïvely porting synchronous design methodologies to asynchronous design may result in suboptimal solutions.

Several other synthesis approaches exist, for more detail, surveys of both asyn-

chronous (Beerel et al., 2010; Taubin et al., 2007) and synchronous (Coussy and Morawiec, 2008) approaches are available.

Despite the breadth of research in this area, there remains a need for an asynchronous design flow that can produce fast, resource-shared implementations with minimal design effort. This thesis attempts to fill that void by contributing a comprehensive design flow, one that permits rapid design and allows the designer to easily trade-off and optimize for several performance metrics.

1.3 Contributions

In this section I will outline the contributions made in this thesis, starting with the proposed design flow, then stepping through chapter-by-chapter to highlight the contributions discussed in each. The contributions made in this thesis have been published in (Hansen and Singh, 2008; Hansen and Singh, 2010b; Hansen and Singh, 2010a; Hansen and Singh, 2012; Gill et al., 2006; Gill et al., 2009).

1.3.1 Proposed Design Flow

The proposed design flow is shown in Figure 1.1. In this figure, italicized items indicate existing tools. Paths with dashed arcs represent existing design flows.

The leftmost path, highlighted with a dashed arc, represents the original Haste design flow. This path starts with a high-level behavioral specification and passes it through the Haste compiler to produce a final implementation. This is a syntax-directed process, producing directly mapped circuits. Alternatively, the rightmost path is manual design, in which a designer typically creates a manually optimized design in a structural hardware description language and passes it through the physical mapping steps. Some optimization and synthesis tools may also exist on this path.

My proposed design flow is shown in the center of the figure. Here, a choice of three options is presented. The leftmost path is a data-driven pipelining flow, described in Chapter 3, that performs a source-to-source conversion of a behavioral specification into an equivalent data-driven pipelined source specification. This path does not perform any automated resource sharing for conserving area, instead targeting high-performance pipelined implementations. This path leverages the existing Haste tools as a back-end.

The center path provides an alternate approach that performs resource sharing in a synthesis step. This path allows the designer to trade off area, performance, power, and energy using an automated design-space exploration approach. This approach will be described primarily in Chapter 4, with energy and power considerations discussed in Chapter 6.

The rightmost path provides a hybrid approach, combining both resource-sharing and high-performance pipelining to target high-performance, low area circuits. This section will target synthesis using a *multi-token* scheduling approach, one in which multiple instances of a problem are being solved concurrently by the circuit. This approach is described in detail in Chapter 5.

Now, let us step one-by-one into each chapter, illustrating how the contributions made in each allow paths in the proposed designed flow to be realized.

1.3.2 Compiler and Source-Level Optimization

The first step in the proposed design flow, in which a behavioral specification is converted to an intermediate representation, is described in Chapter 3. This chapter proposes a novel source-to-source compiler, which incorporates several concurrency-enhancing optimizations, including parallelization, arithmetic optimization, and communication optimization. In addition to these optimizations, the compiler includes a back-end path to produce a pipelined, optimized specification back in the original source

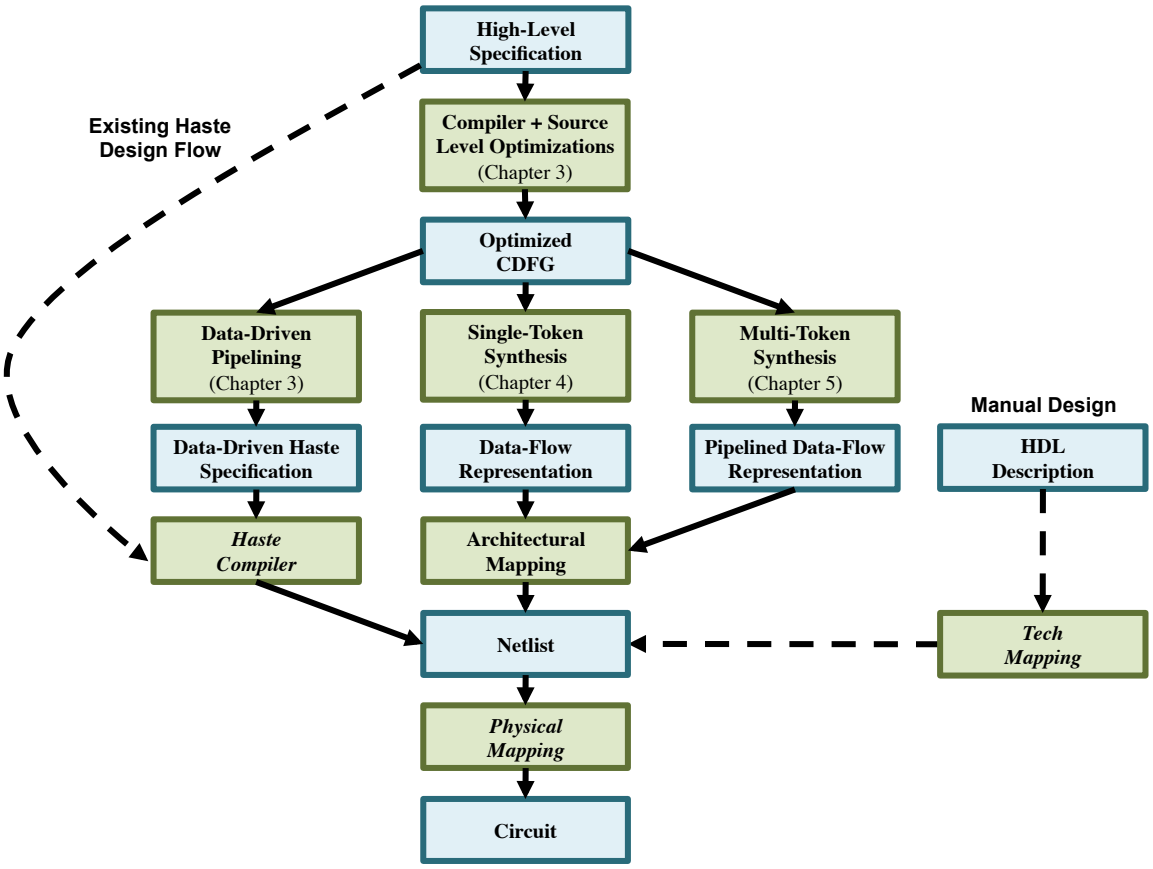

Figure 1.1: Proposed design flow

language (Haste), to be fed into an existing syntax-directed design flow. In addition to having its own synthesis path, the proposed compiler also produces an intermediate representation that is used in the other two synthesis paths, described in Chapters 4 and 5.

1.3.3 Optimal Resource Sharing and Scheduling

In Chapter 4 I attack the problem of resource sharing in high-level synthesis. Unlike the area-hungry approach of Chapter 3 that focuses solely on performance, this chapter will present a shared-resource approach to high-level synthesis that is both fast and optimal. I will present a novel string-based formulation to the scheduling problem and an efficient branch-and-bound strategy to target the problem of resource scheduling, allocation, and binding in an optimal fashion. This chapter will introduce several tight bounds and optimizations in the branch-and-bound framework that effectively prune the scheduling and allocation search spaces, enabling the designer to explore a wide variety of potential solutions in a short period of time.

The work presented in this chapter will provide several scheduling options to the designer, including latency-minimization under resource or area constraints and area-minimization under a latency constraint. The approach has been extended to incorporate mappings of operations to multiple function unit types, including multi-purpose function units such as ALUs, while still providing optimal solutions rapidly. Because the allocation space itself can become broad when considering a wide variety of function units, I will also present a strategy for enumerating the search space of allocations in a dynamic fashion to improve performance.

1.3.4 Pipelining With Shared Resources

Chapter 5 will present an alternate synthesis approach for shared-resource architectures, in which a high-performance, minimal-area pipelined implementation is the ultimate goal. Unlike Chapter 4, which focused on latency as a performance metric, this chapter will target throughput as a constraint, and as a result produce *multi-token* (*i.e.*, pipelined) schedules. In this chapter I will introduce a pipelined data-flow architecture in which data travels directly from source to destination with optimal buffering, synchronizing only when data is needed for computation. The target architecture is distinct from the data-driven pipelines of Chapter 3.

In this chapter I will introduce a pipeline synthesis method for minimizing area under a throughput constraint, allocating the minimum number of function units and buffers required to meet the performance target. I will extend this optimal method for use with large, real-world examples via a heuristic hierarchical approach; this approach will be robust enough to handle both loops and conditionals, allowing for a rich set of input specifications. This approach provides a method for exploring a full spectrum of designs, ranging from high-performance pipelines to low-area, control-driven implementations, simply by tightening and relaxing constraints.

1.3.5 Energy and Power Considerations

Because of the increased necessity for low-power and low-energy implementations, particularly due to the trend towards mobile computing, in Chapter 6 I extend the approach first presented in Chapter 4 to incorporate energy and power. This extension involves creating several new bounds for faster branch-and-bound search space exploration as well as a new minimization strategy specifically targeting minimum energy implementations. This modification extends an already rich set of scheduling strategies to provide all of the following synthesis options to the designer: energy minimization, latency

minimization, and area minimization under the bounds of energy, power, latency, and area.

In this chapter, I will also present a strategy for minimizing energy by performing voltage scaling as a post-scheduling step, in order to squeeze out even more energy savings by exploiting available slack in the schedule. This section will incorporate both heuristic and optimal methods for energy minimization, as well as a method to minimize energy while limiting the number of unique voltage levels.

1.4 Significance of Contributions

My work in Chapter 3 is the *first approach for automatic rewriting of asynchronous high-level specifications through parallelization and pipelining* to obtain higher concurrency. As a result, my approach obtains dramatic performance improvements even while using an underlying syntax-driven translation tool. This work overcomes a significant and long-standing shortcoming of state-of-the-art asynchronous design flows, which tend to be syntax-driven (Haste, 2008; Edwards and Bardsley, 2002). By efficiently transforming the specification through automated parallelization and automated pipelining using my approach, the same syntax-driven tools can now produce implementations that are much more concurrent and, therefore, yield higher performance.

My work in Chapter 4 is the first to demonstrate that the asynchronous (*i.e.*, continuous-time) resource scheduling problem is different and harder than the synchronous (*i.e.*, discrete time) problem. My work is the *first exact solution to the asynchronous resource sharing problem.* Prior work has either focused on heuristic asynchronous approaches or synchronous approximations. The key idea in my work is to solve for the relative order (partial order) of operations, as opposed to solving for their absolute timing.

Prior to this work, the problem of optimal resource sharing for pipelined (*i.e.*, multi-token) systems has been an unsolved problem, for both synchronous and asynchronous systems. The work in Chapter 5 is the *first exact approach, whether synchronous or asynchronous, for optimal resource sharing in multi-token systems.* Prior approaches have generally solved only a part of this problem, *e.g.*, some assume the number of tokens is given, some use a discrete-time approximation, others are heuristic. My approach is the first to optimize over the full joint search space consisting of all allocations, schedules and bindings of resources, all possible buffer insertions (*i.e.*, slack matching), and all token counts. Efficient search space pruning techniques are introduced to make this approach efficient; further speed up and scalability to larger problem sizes is obtained by my hierarchical method.

The majority of prior approaches to resource sharing (synchronous as well as asynchronous) do not consider power or energy as part of the scheduling step. The approaches that do consider these metrics typically treat power or energy only as secondary cost functions, or only provide heuristic solutions. My approach of Chapter 6 incorporates total energy consumption and peak power dissipation as first-class cost functions during the scheduling step. To the best of my knowledge, this is the *first exact approach to provide optimal resource sharing under energy/power constraints.*

The bulk of the work of this dissertation likely is applicable also to synchronous design, including *synchronous elastic systems.* In particular, my continuous-time asynchronous resource scheduling approaches (Chapters 4-6) can likely be directly applied to the discrete-time flavor of this problem as merely a special case. Interestingly, while working on the asynchronous problem, I was forced to think out-of-the-box—*i.e.*, in terms of relative order instead of absolute time—because asynchronous systems do not have a notion of clocking and absolute time. It turns out that, even though application to synchronous design was beyond the scope of this dissertation, it is likely that my

relative order approach is not only applicable to, but highly efficient for, synchronous systems as well. Similarly, the work of Chapter 3 is likely to be applicable to synchronous systems as well because at the behavioral level there is little to distinguish asynchronous and synchronous systems.

1.5 Organization of Thesis

The remainder of this thesis is organized as follows. In Chapter 2, I will give relevant background on architectures, languages and representations, and analysis methods. In Chapter 3, I will discuss the data-driven design approach and several source-level optimizations to improve the concurrency of a specification. In Chapter 4, I will present a shared-resource scheduling methodology, focusing on optimality and efficiency. In Chapter 5, I will introduce an alternative *pipelined* scheduling strategy, one which allows multiple problem instances to be computed simultaneously on the same set of resources. In Chapter 6, I will describe several modifications to the synthesis strategy in Chapter 4 to incorporate energy and power, as well as provide a voltage scaling strategy for improving energy consumption as a post-scheduling step. Finally, I will present conclusions and directions for future work in Chapter 7.

Chapter 2

Background

In this chapter I will provide background on several important concepts that will be relevant for the remainder of the thesis. The following topics will be reviewed:

- Section 2.1 discusses asynchronous architectures, including pipelined and shared-resource implementations. I will also discuss the method of "slack-matching", in which buffers are inserted in order to improve the performance of a circuit.
- Section 2.2 provides background on silicon compilation, from source code to circuit. A discussion of source languages, graphical models for intermediate representations, and design flows is provided.
- Section 2.3 describes the analysis techniques we will use in this thesis, including canopy graphs, the cycle metric, and simulation-based methods. I will also define the terms we will use to describe the performance of a circuit, such as latency and throughput.

2.1 Asynchronous Architectures

Let us start by considering a set of basic architectures for asynchronous designs. I will begin by describing pipelined architectures, in which there is a high degree of

parallelism, possibly at the cost of high area consumption. Next, I will discuss shared-resource architectures, in which resources are shared to conserve area, possibly at the cost of performance. Finally, I will briefly introduce the concept of *slack-matching* via buffer insertion, which is often necessary to improve performance in pipelined systems.

2.1.1 Pipelined Architectures

Pipelining is a common technique used in both synchronous and asynchronous design to improve the throughput of a design. In pipelining, computation is fragmented into multiple portions that can be performed independently; each portion is given its own dedicated hardware for storage and computation. Because each stage in the pipeline has its own dedicated set of resources, it can operate on a different instance of a problem than its neighbor, much like an assembly line.

In this way, multiple instances of a problem are computed at once, improving performance as a whole, although the time associated with a specific problem instance may go up due to overheads in the pipelining process. Aggressive, fine-grained pipelining can result in very high performance circuits since many problems are being solved concurrently. However, pipelining comes at a cost of area, particularly increased storage to hold the data associated with each problem instance, as well as resource area associated with each dedicated function unit.

2.1.1.1 Pipeline Stages and Styles

In hardware, asynchronous pipelines consist of several pipeline stages that communicate via request-acknowledge handshaking signals (Figure 2.1). Typically, a stage initiates computation when it receives new data and a request from its left neighbor. Once data has been accepted (latched), the left neighbor is acknowledged. The stage may then perform operations on the data and forward the results along with a new request to its

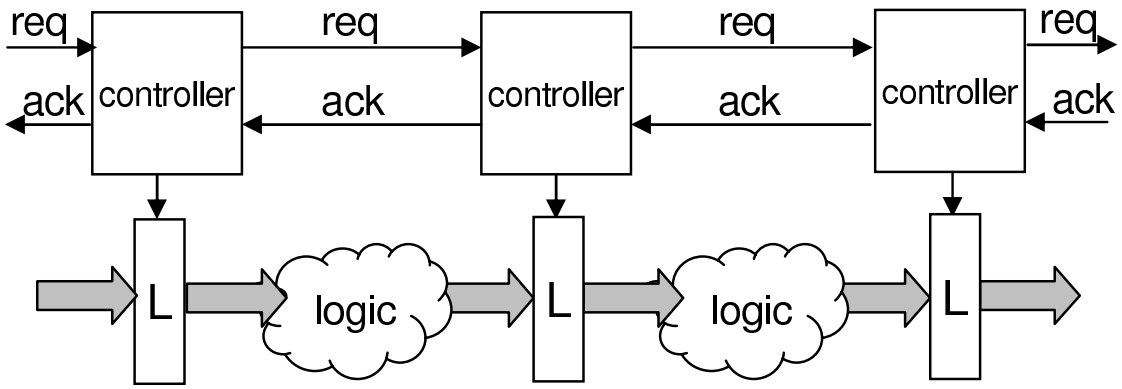

Figure 2.1: Simple asynchronous pipeline

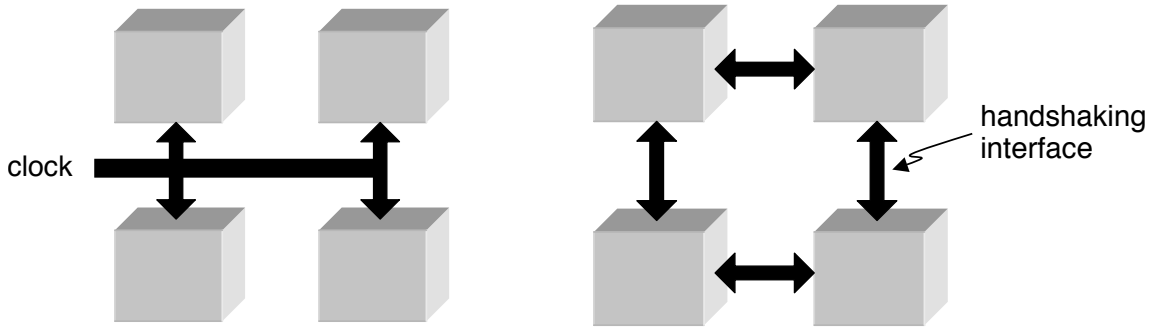

Figure 2.2: Synchronous vs. asynchronous communication

right neighbor. This behavior is unlike that of a synchronous approach (Figure 2.2), in which signals are received from a global clock to latch data.

Several techniques exist to transmit data between stages. Two-phase (Sutherland, 1989) and four-phase (Williams, 1991) protocols are used to signal arrival and acceptance of data. In two-phase handshaking, a transition on a request line indicates new data is available, a transition on an acknowledge line indicates the new data has been latched. A four-phase handshake is level-based rather than transition-based. A legal four-phase scenario is as follows: a request goes high indicating new data, the acknowledge line goes high to indicate the data has been latched, the request then resets to zero, and soon after the acknowledge resets to zero. Other variants are possible depending on the meaning attached to each event. For example, the acknowledge resetting may indicate that data has been latched. The exact implementation is up to the designer.

Aside from the variety in handshake protocols, data can also be encoded in multiple different ways. Bundled data (Sutherland, 1989) is a common approach in which a single wire is dedicated to each bit, and the data itself is combined with a control signal with a *matched delay* that corresponds to the computation time of logic between the stages. Dual-rail encoding (Williams, 1991) is a different paradigm where two wires are associated with each bit of data; some combination of signals on the two wires indicate that computation has completed. This type of encoding is more robust to timing variation, but will incur an additional area penalty due to completion detection.

Several different pipeline styles exist, from GasP (Sutherland and Fairbanks, 2001) to MOUSETRAP (Singh and Nowick, 2001) to Sutherland's micro-pipelines (Sutherland, 1989) to high-capacity (Singh and Nowick, 2007) pipelines. The work presented in this thesis targets two-phase, bundled-data pipelines, but is certainly amenable to other styles as well.

2.1.1.2 Data-flow Pipelines

In this thesis, I will refer to data-flow pipelines as pipelines in which each individual piece of data travels through the architecture without any unnecessary synchronization. That is, each stage will consist solely of data belonging to one "variable", and will travel along a channel until it synchronizes with another piece of data only as needed to perform a computation.

This type of pipeline can have a highly complex topology; rather than a flat, linear flow of data, there may be many paths forking and joining throughout the full pipeline. In order to achieve high performance with this type of pipeline, *slack-matching* is often needed in order to match the buffering on one path to that of another parallel path; this will be described in Section 2.1.3. Data-flow pipelines are the specific target of the synthesis approach proposed in Chapter 5.

2.1.1.3 Data-driven Pipelines

Data-driven pipelines are proposed in Chapter 3 as an alternative to data-flow pipelines. In these pipelines, slack-matching has been explicitly performed via construction; large blocks of data are synchronized at once and referred to as the "context" of an individual problem. As data accumulates, the size of each synchronized buffer increases; as data is consumed and is no longer needed in the pipeline, it is dropped from future synchronized buffers. This type of pipeline consists of large, linear blocks with minimal fork and join constructs, in contrast to data-flow pipelines that have lightweight buffers and complex topologies.

While data-flow pipelines are more efficient; data-driven pipelines are found in examples such as common pipelined processors, which may have several stages (*i.e.*, fetch, decode, execute). Data is not directly passed from source to destination, but typically goes through every stage even if a stage does not operate on the data.

2.1.2 Shared-Resource Architectures

Shared-resource architectures are an alternative to asynchronous pipelines. Unlike pipelines, in which control is distributed, shared-resource architectures generally rely on a global controller to transfer data between function units and registers. An example of such an architecture is illustrated in Figure 2.3.

In this figure, a large, monolithic control block is connected to a set of multiplexers that control the flow of data into function units. This control block is also connected to a set of registers to determine which register will latch the result of computation. Here, the complete schedule of data transfer is encoded in the controller, unlike in pipelines where data itself triggers computation.

A global controller is not a requirement for shared-resource architectures; instead, smaller individual controllers can be used, as in (Theobald and Nowick, 2001). While

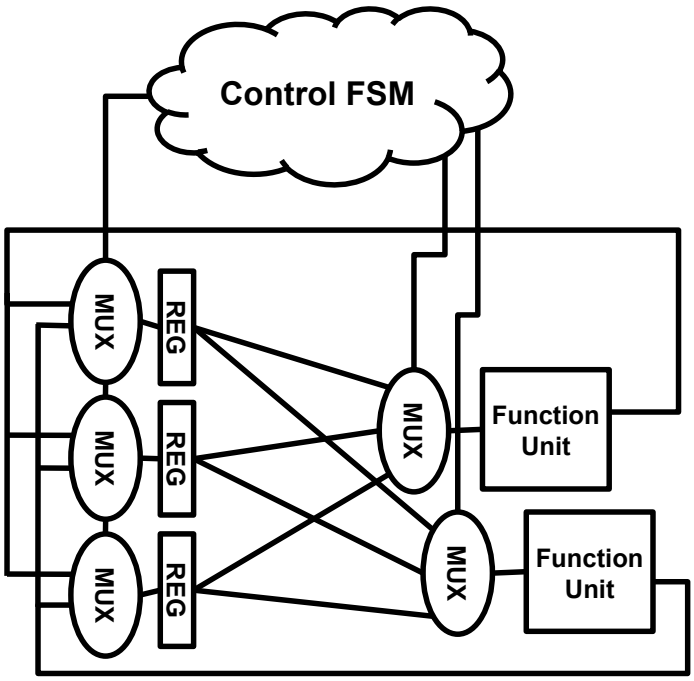

Figure 2.3: Shared-resource architecture

other flavors may exist, the most common element of a shared-resource architecture is control-directed transfer of data between shared registers and resources.

2.1.3 Buffering Requirements (Slack-Matching)

Slack-matching is a technique used by designers to reconcile two paths that have mismatched latencies or storage capacities. A slack-mismatch is a performance concern that results in reduced throughput, as data cannot enter a pair of forked pipelines if one of the paths is already full. By adding additional buffers on one of the paths, a slack-mismatch can be alleviated, and performance improved.

The problem of slack-mismatch is illustrated in Figure 2.4a. In this example, two paths fork off in the pipeline, one with a single stage, another with multiple sequential stages. Let us assume each stage has the same attributes, *i.e.*, forward and reverse latency (see Section 2.3). I will briefly illustrate how the performance will be limited

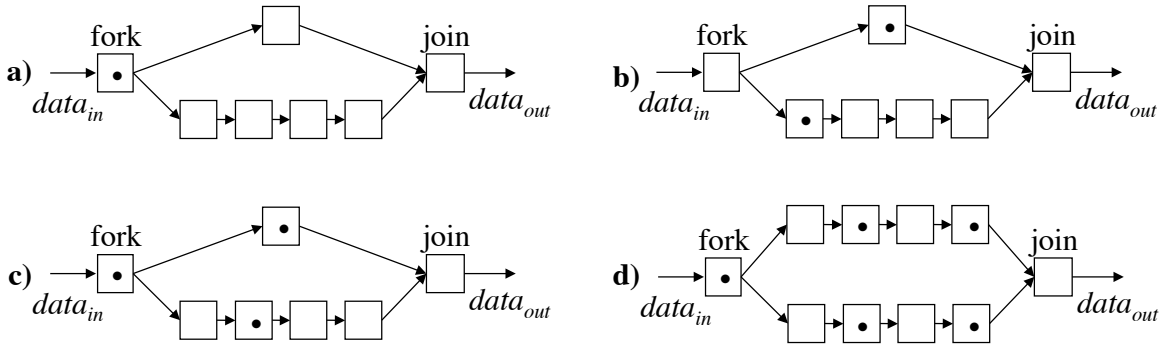

Figure 2.4: Slack mismatch example

due to a lack of buffer space in the shorter path.

Consider the operation of this pipeline. To begin, a piece of data enters the start of the pipeline on the left (Figure 2.4a). The data then splits and travels down each path, to be synchronized at the join node later in the pipeline (Figure 2.4b). A new piece of data can then enter, ready to start computation on both forks (Figure 2.4c). However, the data is stalled because there is no space for it on the shorter path.

By inserting additional buffers on the shorter path, additional room is available on the shorter path, allowing data to enter, thus alleviating the slack-mismatch (Figure 2.4d).

The problem of slack-mismatch is automatically avoided in the approach in Chapter 3 by using a data-driven pipeline style. However, in Chapter 5, I incorporate the slack-matching problem into the proposed synthesis method in order to slack-match the data-flow pipelines that are generated by that method.

2.2 Languages, Representations, and Compilation

In this section I will review the process of silicon compilation. I will begin by giving an overview of behavioral specification languages, particularly focusing on the Haste language that is used heavily in Chapter 2. Next, I will describe graphical representations,

such as abstract syntax trees, data-flow graphs, and Petri nets. Finally, I will give an overview of existing asynchronous design flows, such as the syntax-directed Haste and Balsa design tools, as well as review more general synthesis approaches, including those that attack the common constrained-optimization problem for scheduling, allocation, and binding of shared resources.

2.2.1 Behavioral Description Languages

2.2.1.1 Overview

The breadth of potential languages for performing asynchronous high-level synthesis is wide; some designers use software programming languages such as C for their high-level descriptions, while others use common hardware languages such as Verilog and VHDL, and yet others utilize highly-specialized languages for asynchronous design, such as Haste and Balsa. There are several factors that weigh into the selection of a language; two common desires are tool support and richness of the specification language.

Two syntactical features designers often require in a hardware description language are channel communication to transmit data between modules, and a simple, explicit means for representing parallelism in the specification. Unfortunately, these features are often lacking in software programming languages, hence specialized hardware languages are often a better match.

In Chapter 3 I will focus specifically on Haste, a specialized asynchronous design language that is a variant of CSP (Hoare, 1985), as an input specification. This language was selected because a complete design flow existed at the time, and because it provided the desirable features of concurrency and channel communication.

```
& byte = type [0..255]
& byteplus = type [-255..255]
& GCD: main proc(IN?chan <<byte,byte>> & OUT!chan byte).
   begin
   & ab: var <<byte, byte>> ff
   & a=alias ab.0
   & b=alias ab.1
   & s: var byteplus ff
   | forever
     do
      IN?ab;
      do a # 0 then
         s := a-b;
         <<a,b>> := if sign(s)
                      then <<b,a>>
                      else <<s,b>>
                    fi
      od;
      OUT!b
     od
   end
```

Figure 2.5: GCD example

2.2.1.2 Haste Language

Let us focus on the primary source language used by our approach: the Haste language.

Some key Haste constructs are as follows:

- channel reads (`IN?ab`)
- channel writes (`OUT!b`)
- assignments (`s:=a-b`)
- tuples (`<<a,b>>`)
- sequential composition (`b:=a+x ; c:=b+y`)
- parallel composition (`a:=b+x || c:=d+y`)
- loop control (`forever do ...od`)
- block definition (`begin | ...end`)
- type definition (`byte = type [0..255]`)

- procedure definition (`GCD: main proc(...).`)
- conditional assignment (`x:= if bool then y else z fi`)

Figure 2.5 shows the Haste specification of a very simple program that computes the GCD of two numbers. The program has one input channel, `IN`, through which it receives two data items from the environment in a tuple (pair of bytes). It also contains an output channel `OUT`, through which it transmits results to the environment. Each channel consists of a pair of request-acknowledge wires along with the data wires.

In the specification, `ab` and `s` are all storage variables, while `a` and `b` are merely aliases/pointers to variables in the tuple `ab`. The main construct in the body of the specification is a `forever do` loop. This loop reads from the input channel, then enters a second loop that computes the GCD of the input numbers. Once the GCD is computed, it transmitted to the environment on the output channel, `OUT`.

The conversion of these language constructs into a final hardware representation will be discussed in Section 2.2.3.3.

2.2.2 Graphical Representations

The first step in the compilation process is to transform the human-readable behavioral specification into a form that is amenable to processing and optimization by the compiler. Generally, this new form is an intermediate representation that often takes a tree-like or hierarchical structure, *e.g.*, an abstract syntax tree. Now, let us consider several common graphical representations that are used either by the compiler, or by the designer, in order to represent a specification or model its performance.

2.2.2.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a representation that is generated after the parser reads the input specification. This tree consists of individual nodes that correspond in

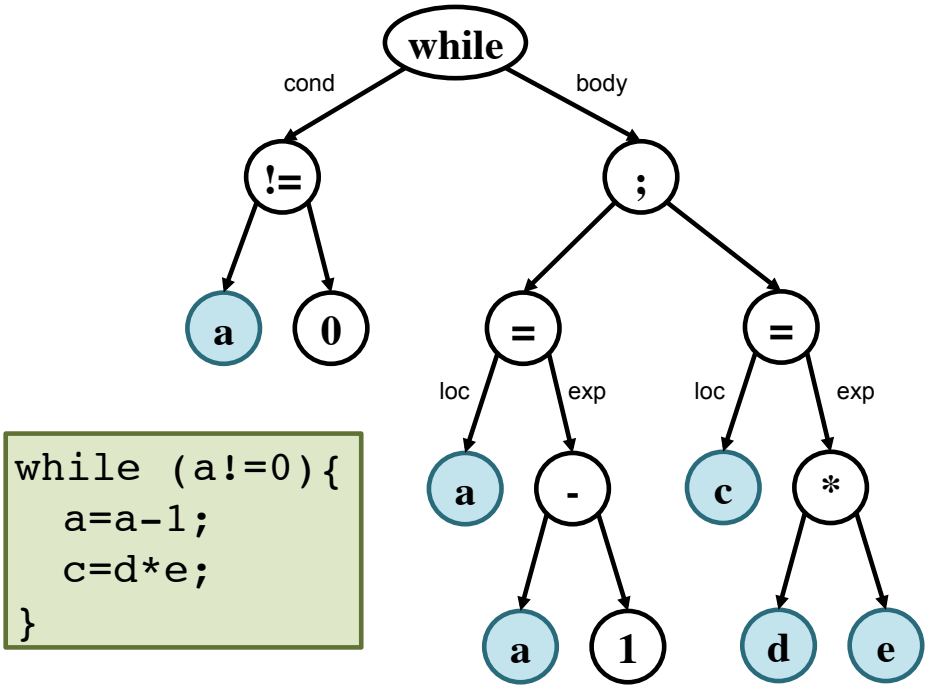

Figure 2.6: Abstract syntax tree example

a one-to-one fashion with source code constructs. Figure 2.6 illustrates a sample AST for a while loop. Here each construct is converted directly into a node; the while loop becomes a control construct with two children, a conditional and a loop body. The conditional is a binary not-equal operation that requires two children, in this case the variable a and the literal 0. The remainder of the graph is constructed in a similar fashion.

After the AST has been generated, the compiler performs several annotations, such as those linking a variable name to its declaration and type, determining bit-widths for operations, and so on. A compiler may perform optimizations here as well, such as conversion to single-static assignment form, dead-code removal, etc.

2.2.2.2 Control Data-Flow Graph

In a data-flow graph (DFG), unlike an AST, the focus is on the flow of data rather than the original control constructs. A basic DFG consists of several nodes that represent

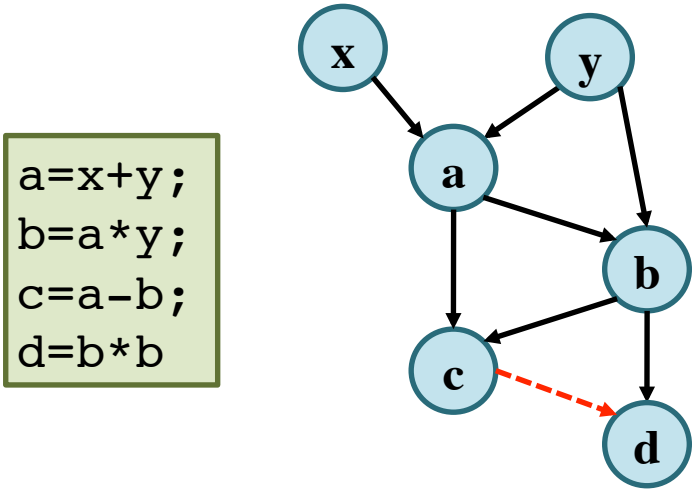

Figure 2.7: Control/data-flow graph example

operations, and arcs between these nodes that represent the flow of data.

A data-flow graph can be extended to incorporate control information, such as loops and conditionals. This extension is called a control/data-flow graph (CDFG). In Chapter 5, a similar construct is used, a folded-dependence graph. This type of graph incorporates data dependencies between operations, and then inserts additional control elements, including scheduling arcs, control constructs such as loops and conditionals, and write-after-read dependencies from which buffering is inferred.

A sample CDFG is shown in Figure 2.7. In this example, the dependence between c and d is purely control, rather than data, and is shown as a dashed red arc.

2.2.2.3 Petri Nets

A Petri net is a mathematical representation that is often used for modeling concurrency in systems. Petri nets have a wide range of uses; they can be used for simulation of operation, to determine correctness of an implementation, to test for deadlocks, etc. A formal definition of Petri nets is not required for this thesis; however, a basic introduction can provide some insight.

A Petri net consists a set of places, arcs, and transitions, through which *tokens* flow

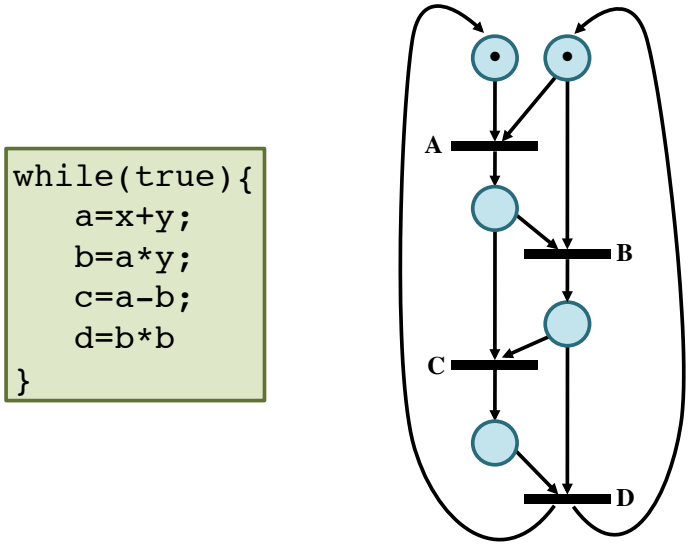

Figure 2.8: Petri net example

to model data or control transfer. A place is a storage location for tokens, an arc is a path on which a token can travel, and a transition is a guard that synchronizes data or control. A sample Petri net is shown in Figure 2.8.

The behavior of a Petri net is as follows. First, tokens are placed in the graph in an initial marking, which enables some set of transitions (otherwise the Petri net is not live). However, a specific transition cannot fire until all of its incoming arcs have a token available, at which point the transition becomes enabled. When a transition fires, a token is consumed from each input arc to the transition, and a token is produced on each output arc from the transition, to arrive at a new place. A new marking is produced, and the Petri net can continue operation by firing another transition.

There are several extensions to Petri nets, *e.g.*, timed Petri nets, and more restricted versions, *e.g.*, marked graphs. A timed Petri net associates a time penalty with either a place or a transition, *e.g.*, requiring some amount of time to elapse before a transition is enabled to fire. A marked graph is a Petri net that does not have the potential for choice or OR-causality; requiring that each place has only one input and one output arc. These two specific models are noted because of their ties to this thesis; timed Petri

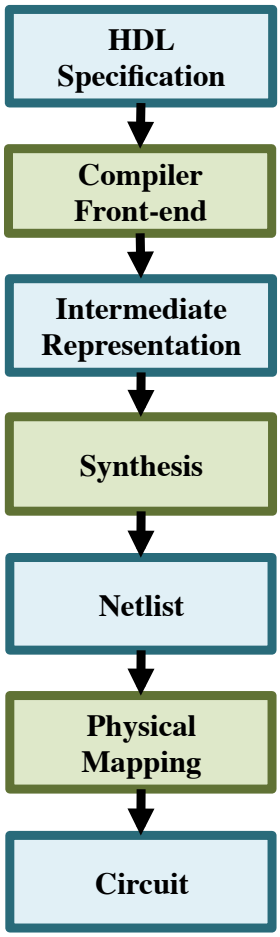

Figure 2.9: Common high-level synthesis flow

nets were used as an initial model in Chapter 4, while marked graphs were used as an initial model in Chapter 5.

2.2.3 Compiler Flow

Now that I have discussed languages and representations used in the compilation process, let me now describe the flow of compilation in high-level synthesis (HLS).

The flow of a common HLS compiler is shown in Figure 2.9. Here, we start with an initial behavioral specification (source code), then convert this specification into an intermediate representation. This representation can then be modified by performing

optimizations and synthesis tasks such as scheduling, allocation, and binding of shared resources. Finally, this modified representation is sent to a back-end for conversion to hardware.

2.2.3.1 Source to Intermediate Representation

The first step in the hardware compilation process is to convert the input specification into an intermediate representation; this is often described as the front-end of the compiler. As with software compilers, an HLS compiler will step through the common steps of lexical, syntactic, and semantic analysis in order to convert the source to an intermediate representation. The compiler implemented in Chapter 3 is a recursive-descent compiler that converts a Haste specification into an annotated AST. After the intermediate representation is produced, optimizations may be performed prior to being output by the back-end of the compiler.

2.2.3.2 Synthesis

Once an intermediate representation is created, the next step is to convert it to hardware. In this thesis I will consider two main forms of synthesis. The classic synthesis problem is one of performing resource sharing as part of a constrained-optimization problem. Chapters 4 and 5 will focus on this type of synthesis. An alternate method is syntax-directed translation, which is used by the Haste compiler

Let us describe first three main steps of synthesis:

- *Allocation* is the step where the designer or design tool determines the number of resources that will be used in the implementation. This task can include determining the number of function units, registers, etc. that an implementation will use. This step is vital as it trades off area for performance; more area generally means better performance, since there will be less contention for resources.

- *Scheduling* is the step where the designer or design tool determines the order of execution for a set of operations that share the same type of resources. The goal of scheduling is often to optimize a performance metric, such as maximizing throughput or minimizing latency. Alternatively, scheduling may target low-energy or low-power by scheduling execution appropriately, such as by scheduling operations on less power-hungry function units, or spreading execution out across the time domain.

- *Binding* is the step where schedules and operations are mapped onto specific pieces of hardware, *i.e.*, resource instances. As an example, an operation may be scheduled to execute on a type of function unit at a specific time, but the specific function unit instance may not yet have been determined. The binding step will finalize these mappings; this is important because different bindings may lead to different multiplexing costs (in terms of both area and delay).

Often a designer will use a synthesis approach to explore a full design space in order to balance or optimize for specific metrics, such as area, performance, energy, etc, as in Chapters 4 and 5. An alternate route is to generate a final implementation via construction; *i.e.*, applying a specific set of transforms and mappings to create an implementation without exploring a full design space. In Chapter 3, the proposed compiler creates a data-driven implementation by applying a series of transforms that can be manually enabled or disabled by the designer, then the syntax-directed Haste compiler converts the optimized specification in a one-to-one fashion into hardware.

2.2.3.3 Syntax-Directed Translation

In syntax-directed translation, an intermediate representation is generated from the source specification, then is directly mapped into hardware. The goal of a syntax-directed compiler is to produce implementations that have a one-to-one relationship

```
&fifo=proc(IN?chan byte &
           OUT!chan byte).
begin
  & x: var byte
  | forever do
      IN?x; OUT!x
    od
end
```

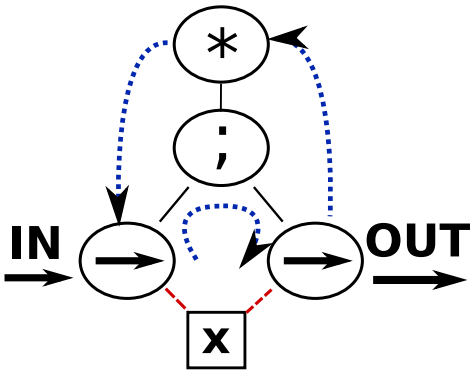

Figure 2.10: Haste example

with the source specification; each language construct typically translates to a specific hardware library component. This paradigm makes the process very transparent to the designer, but without manual optimization, syntax-directed translation may lead to lower performance. Two of the more mature syntax-directed translation approaches to synthesis are the Haste and Balsa design flows.

2.2.4 The Haste Design Flow

Let us focus on the Haste design flow, which we use heavily in Chapter 2. Given a specification, the Haste compiler parses the input into an intermediate handshake-component representation, then syntactically maps each construct onto a predefined library component to generate a hardware implementation, as shown in Figure 2.10. In particular, there is a predefined component that implements the `forever do` construct: it repeatedly initiates handshakes with its target. Similarly, there is a predefined component that implements sequencing, denoted by "`;`". The sequencer, upon receiving a handshake from its parent, performs a handshake with its left child followed by a handshake with its right child. The variable `x` maps to a storage element. Finally, the read and write operations, (*e.g.*, read from channel IN and write to `x`) map to redefined components called *transferrers*, denoted in the Figure by "$\longrightarrow$".

In summary, the compilation approach is quite simple but very powerful: fairly complex algorithms can be easily mapped to hardware. Gate-level implementations for complex designs, such as complete micro-controller, can be generated from a few hundred lines of high-level code.

In this work, I will use the syntax-directed paradigm in Chapter 3, but only after performing various automated optimizations (code rewritings) to improve performance.

2.3 Analysis Methods

In this section I will describe several common analysis methods for determining the performance of asynchronous pipelines. I will start by giving definitions of latency, throughput, and cycle time: key metrics used in this thesis. Next, I will describe canopy graphs, a useful tool for determining the throughput bound of a pipelined system. Then, I will discuss the cycle metric mean problem and its performance implications. Finally, I will discuss how simulation can be used as a tool for analyzing performance.

2.3.1 Performance Metrics

There are three key performance attributes we will be concerned with in this thesis: latency, cycle time, and throughput.

2.3.1.1 Latency

The term "latency" often refers to the time it takes for an action to complete from start to finish, such as the time it takes for data to propagate from one place to another (*e.g.*, in networking), or the time it takes for a program or operation to react to stimulus (*e.g.*, in user interfaces). This general definition is perfectly applicable for use in this thesis, but let us further define latency for scenarios that are common in the following

chapters.

For a schedule, latency will refer to the time it takes for a complete execution of every operation in a specification, from start to finish. In essence, this refers to the time from when the first action begins in the schedule to when the final action completes.

In the context of pipelining, the latency of a pipeline is similar; here, the forward latency of a pipeline is the time it takes for a token to travel from the start to the end of an empty pipeline (*i.e.*, from input to output). Similarly, the forward latency of a pipeline stage is the amount of time it takes for data to propagate from that stage to the next. If F_i represents the forward latency of a stage i in a pipeline, then the overall forward latency of a linear pipeline is:

$$F = \sum_{\forall i} F_i$$

Correspondingly, the reverse latency of a pipeline is the time it takes for a "hole" to propagate from the end of the pipeline to the start of the pipeline (*e.g.*, from output to input) if the pipeline is initially filled. Thus, the reverse latency is determined by the speed at which acknowledgments propagate backwards. If the reverse latency of a stage i in a pipeline is R_i, then the reverse latency of a linear pipeline is:

$$R = \sum_{\forall i} R_i$$

Often the designer will be tasked with minimizing the latency of a schedule or partial schedule in order to meet a set of constraints.

2.3.1.2 Cycle Time

Unlike latency, which is the measure of total time for a set of events to complete, the term "cycle time" refers to the amount of time that elapses between repeated actions.

As an example, in a linear pipeline, the cycle time can refer to the amount of time that elapses between outputs produced by its final stage, or, similarly, the amount of time between consumption of inputs by its first stage.

The cycle time of a linear pipeline is actually tied to the cycle time of its slowest stage. In a homogeneous pipeline, in which all forward and reverse latencies are the same for each stage, the cycle time of a stage is typically equal to:

$$CT_i = F_i + R_i$$

In the context of scheduling, the cycle time of a schedule is the time that elapses between two problem instances starting (or finishing) their schedules. As an example, a multi-token schedule may allow new problem instances at times 0, 20, 40..., and each problem instance may complete at times 100, 120, 140..., thus the schedule has a cycle time of 20 and a latency of 100.

2.3.1.3 Throughput

The term "throughput" refers to the inverse of cycle time. We often use throughput as an indicator of the performance of an implementation; the higher the throughput, the better the performance.

2.3.2 Canopy Graphs

Using canopy graphs for performance analysis was originally explored in the context of asynchronous pipelined rings (Williams, 1991; Williams et al., 1987). Since then, the work has been expand to linear and hierarchical pipelines (Lines, 1998), (Singh et al., 2002), and (Gill, 2010).

The basic concept is as follows: the performance of a pipeline in steady-state is a function of its occupancy, or, the number of data-items (tokens) that exist in the

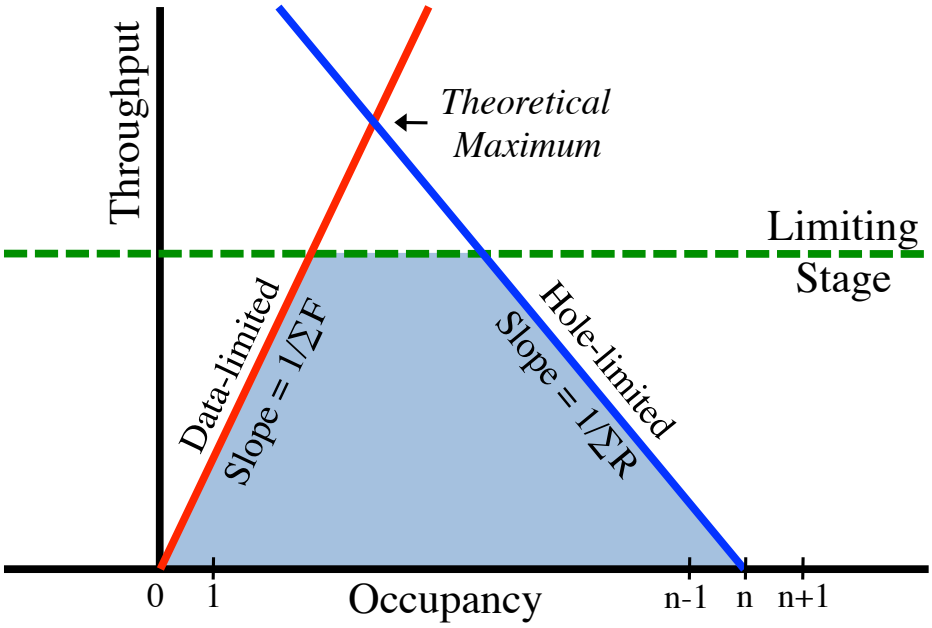

Figure 2.11: Basic canopy graph

pipeline. If the occupancy is too low, the pipeline is underutilized, and therefore it cannot achieve its maximum throughput. This is referred to as "data-limited" behavior. If the occupancy is too high, there is essentially contention for storage space; an item cannot move ahead to the next stage until that stage is vacated. In order to free up space, holes must travel *backwards* in the pipeline. We will refer to throughput degradation due to congestion as "hole-limited" behavior.

In analysis, data-limited and hole-limited behavior will place an upper bound on the throughput of a specification, separating the graph into two distinct regions. Figure 2.11 illustrates the achievable throughput versus the number of data items in a pipeline. In this figure, we see the region on the left-hand side of the graph is limited in the number of tokens, while the right-hand side is overly congested. The slope of the data-limited line is actually equal to the reciprocal of the forward latency of the pipeline, while the slope of the hole-limited line is equal to the negative reciprocal of the reverse latency of the pipeline (Williams, 1991). The data-limited line traditionally starts at the origin,

while the hole-limited line intersects the x-axis where the maximum occupancy of the pipeline is exceeded.

The graph is further limited by the throughput of the slowest stage; this stage's throughput introduces the bounding horizontal line at the top. This is what leads to the "canopy"-like structure of the graph. The operating region of the graph is the full region under this set of lines.

Several extensions and generalizations to the theory of canopy graphs have been introduced, including handling parallel and sequentially composed pipelines (Lines, 1998; Gill, 2010), as well as conditional operation and loops (Gill, 2010). A full review of this work is available in (Gill, 2010), but two key observations are that the joint canopy graph of a set of parallel pipelines will be the intersection of their individual pipelines, while the joint canopy graph of sequential pipelines will be their horizontal sum.

2.3.3 Maximum cycle mean

The cycle mean is an important property of a graph that can be used to help determine the performance of an implementation. In Chapter 5, I will use a graph-based model that is amenable to the classic *maximum cycle mean* problem (Dasdan and Gupta, 1997), which can be employed in order to bound the cycle time of a potential solution.

The computation of the cycle metric is rather simple. We begin with a cycle in a graph that has each of its arcs annotated with two values: a weight and a cost. In the scenario in Chapter 5, that cost will be a delay.

The cycle metric for cycle c in graph G is defined as follows:

$$\text{Mean}(c) = \frac{\sum_{e \in c} \text{delay}(e)}{\sum_{e \in c} \text{weight}(e)}$$

where e is an edge in the cycle c. The cycle mean for one cycle bounds its minimum cycle time; the specific cycle cannot work any faster. Since a typical graph may consist of many cycles (thousands or more), in order to determine the cycle time of the full graph, we must find the maximum of the cycle means for all cycles in the graph:

$$\text{Cycle Time}(G) = \max_{c \in G} (\text{Mean}(c))$$

This metric will be utilized heavily to determine performance in Chapter 5.

2.3.4 Simulation

Simulation provides an alternate means to measure the performance of an implementation. While analysis methods such as canopy graphs and the cycle metric can provide a model for performance under a certain set of conditions (*i.e.*, steady-state behavior), the stochastic nature of constructs such as conditionals and loops can make such analysis imperfect at best.

For verification of the work presented in Chapter 3, the built-in Haste simulator was used. As the Haste tools have since become unavailable, I created my own discrete-event simulator to verify the performance of the work presented in Chapter 5.

2.4 Summary

In this chapter I have provided background on various topics that will be relevant in the remainder of the thesis. For each chapter, it will be necessary to keep the target architecture in mind; therefore, the discussion in Section 2.1 will be particularly relevant. In addition, the background on compilation and the Haste design flow will be particularly useful for the upcoming chapter on data-driven design, Chapter 3. Chapter 4 will primarily rely on background from Section 2.1, particularly shared-resource architectures.

Finally, the analysis methods discussed in Section 2.3 will be used heavily in Chapter 5, where the performance of a pipelined data-flow architecture is analyzed.

Chapter 3

Data-Driven Design: Unlimited Resources

A fundamental desire in the process of design, whether in the realm of hardware, software, or even beyond computing, is to be able to explore a complete design space to find the best possible solution. Every designer must make some trade-offs; within the design space of our problem, several unique options exist, some implementations may focus on performance, while others may focus on area, energy, or power, or any combination in-between. One key problem, however, is that exploring every possible design manually is often infeasible within a reasonable time frame.

Therefore, the work presented in this chapter is meant as a stepping stone towards automated and, interestingly, transparent design-space exploration. In this work, the designer provides a high-level specification, and passes it off to an approach that performs multiple performance-enhancing optimizations in order to generate a high-throughput implementation in the original source language (Haste). While this approach is solely performance-oriented, future synthesis frameworks presented in Chapters 4 and 5 rely on the transformations performed in this work to aid in meeting performance constraints, but target a more diverse set of implementations by allowing

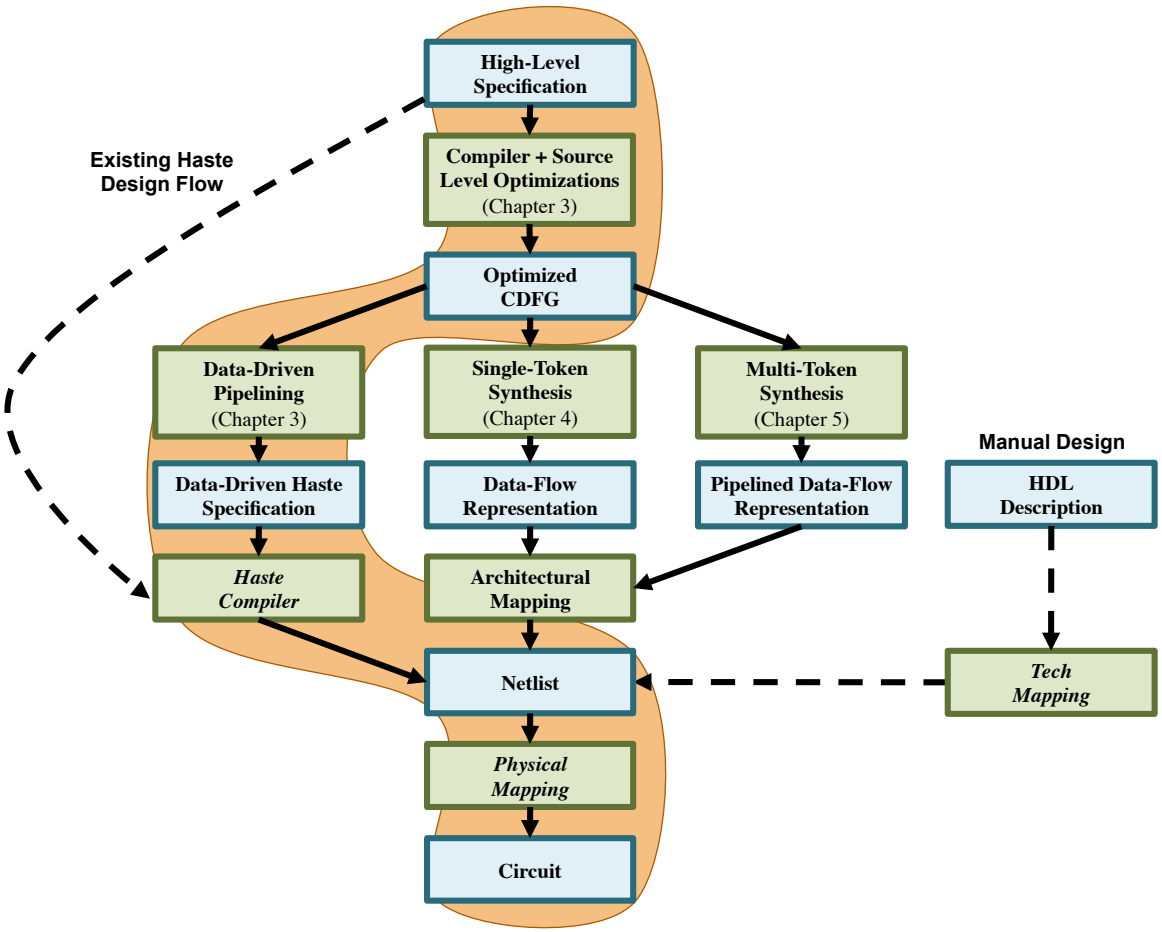

Figure 3.1: Data-driven design flow

for shared resources.

In the design flow from Chapter 1, repeated here in Figure 3.1, three main paths diverge after an intermediate representation is generated. This chapter focuses on two steps in the design flow: (i) converting the source specification to an intermediate representation and (ii) transforming this intermediate representation into a performance-oriented "data-driven" implementation. The synthesis path followed in this chapter is highlighted in Figure 3.1.

3.1 Introduction

Because of the syntax-directed nature of existing asynchronous design tools, generating high-speed implementations can be an arduous process. The best-known tools (*e.g.*, Haste/Tangram (Haste, 2008), and Balsa (Bardsley and Edwards, 2000)) use syntax-directed translation to compile behavioral specifications directly to circuits, with few high-level optimizations. In this type of design flow, each language construct directly maps to a specific hardware component. Therefore, the performance of a specification depends on the amount of optimization the designer manually performs. Straightforward specifications often have low performance due to unnecessary sequencing and unpipelined operation. As a result, designers must either contend with relatively slow implementations or bear the burden of writing highly optimized specifications themselves.

Burdening the designer with optimizing a specification has several drawbacks. First, writing highly concurrent code entails much effort and is error-prone. Second, such code often lacks readability and maintainability, and is therefore hard to modify and reuse. Finally, such a manual approach hinders automatic design-space exploration. In an ideal design flow, performance analysis tools are typically used to identify bottlenecks in the system, and then local modifications are applied to remove the bottleneck; this procedure is repeated until desired performance is achieved. Therefore, code rewriting should ideally be automated.

This chapter introduces an alternative to manual optimization: an automated *"source-to-source" compiler* that transforms one behavioral specification into another behavioral specification with significantly higher concurrency. The proposed approach introduces a suite of transformations:

- *parallelization* for increasing statement-level concurrency,

- *pipelining* for increasing concurrency within a statement group,
- *arithmetic optimization* for increasing concurrency at the sub-statement level, and
- *re-ordering of channel communication* for increasing concurrency across modules.

As a result, designers can write straightforward behavioral code, focusing mainly on its functional correctness rather than on concurrency and performance. The code is automatically transformed by the proposed source-to-source compiler to be highly concurrent, and then passed back through the original Haste design flow.

The two techniques of arithmetic optimization and communication reordering are core contributions of our approach. While basic parallelization and pipelining may help optimize a specification at the granularity of individual statements, there are often performance bottlenecks due to individual statements with long-latency arithmetic operations (*e.g.*, 64-bit adds or multiplications). Further, a single statement may have a complex expression involving multiple arithmetic operators. The proposed approach pushes concurrency enhancement down to a sub-statement level by introducing all of the following: expression re-factoring to introduce parallelism, expression pipelining, and pipelining of individual ('atomic') operators. As a result, bottlenecks due to complex arithmetic are alleviated.

The proposed approach to reordering of channel communication actions addresses a challenging problem. In particular, for a given module, changing the order of two communication actions is fundamentally different from reordering two computational actions. In the latter case, dependency analysis can easy help determine which reorderings or parallel groupings of those actions preserve the original semantics. However, channel communication inherently involves subtle synchronization issues, and naïvely reordering two communication actions may introduce a deadlock into the sys-

tem. A conservative approach is to always maintain the original order of channel actions; although safe, such an approach is suboptimal. The proposed strategy, instead, is to pursue a more optimal approach that includes a careful analysis to determine the space of legal code transformations. As a result, our approach provides greater opportunity for concurrency enhancement.

Previous approaches for improving the throughput of implementations produced by the Haste and Balsa tools have mostly focused at the circuit and intermediate (handshake) levels, including more optimized circuit-level designs of handshake components (*e.g.*, more concurrent sequencers (Plana et al., 2005)), and peephole optimization and re-synthesis at the intermediate level (Chelcea and Nowick, 2002). While some of these approaches have yielded significant speedup (1.54–2.06x), they are unable to take advantage of the significantly greater optimization opportunities at a higher level. As Section 3.5 shows, optimizing at the source level can provide an order of magnitude greater speedup. Moreover, the intermediate and circuit-level approaches are orthogonal to the proposed approach, therefore they are not excluded from being applied within the design flow.

The domain of specifications targeted by the proposed approach are *slack elastic* systems (Manohar and Martin, 1998a). A slack elastic system preserves correct operation even if extra pipeline buffer stages (*i.e.*, extra slack) are introduced on any communication channel. It was shown that a system is slack elastic if it is deadlock-free and it satisfies certain properties regarding channel probing and non-determinism (Manohar and Martin, 1998a). Since the approach introduces pipelining into a specification, the assumption of slack elasticity is a requirement.

The proposed approach has been implemented in an automated tool, and evaluated on a suite of design examples. The resulting concurrency-enhanced specifications were run through the commercial Haste tools from Philips/Handshake Solu-

tions (Haste, 2008), and synthesized to gate-level netlists and simulated. Experimental results demonstrate that the original specifications are correctly and efficiently rewritten into highly concurrent ones. If code length is used as an indicator of designer effort, the proposed approach reduces the required effort by a factor of 3.3x on average (up to 8.8x). Alternatively, the impact can be quantified by the throughput improvement achieved by optimizing the original specification: up to 59x speedup using the basic approach, and a further 5.2x using arithmetic pipelining.

The remainder of this chapter is organized as follows. Section 3.2 reviews the Haste flow and asynchronous pipelining, then discusses related previous work. Then, Section 3.3 presents the basic concurrency-enhancing transformations. Section 3.4 discusses advanced topics, including arithmetic optimization, handling of conditionals and loops, and reordering of channel communication actions. Section 3.5 presents results, and finally Section 3.6 gives conclusions and future work.

3.2 Background and Previous Work

This section first briefly reviews the relevant portions of the Haste design flow then discusses its limitations. Next, asynchronous pipelines are briefly reviewed, along with a discussion of the distinctions between control-driven, data-driven, and data-flow design paradigms. Finally, prior related work is presented.

3.2.1 The Haste Design Flow

The examples discussed in this chapter have been synthesized and simulated using the Haste design flow, which was described in Chapter 2. Recall that the Haste flow accepts specifications written in a high-level hardware description language, and compiles them, via syntax-driven translation, into a gate-level circuit. The high-level

```
&fifo=proc(IN?chan byte &
           OUT!chan byte).
begin
  & x: var byte
  | forever do
      IN?x; OUT!x
    od
end
```

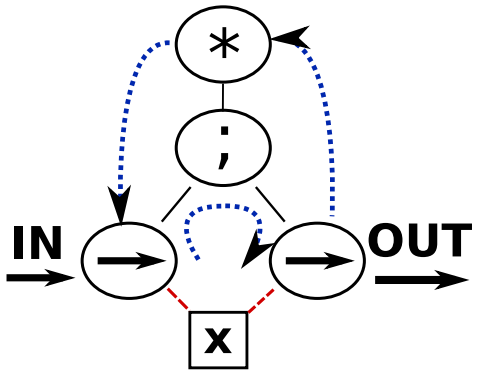

Figure 3.2: Haste example

language is a close variant of the CSP behavioral modeling language (Hoare, 1985).

The main Haste language constructs that are used in the presentation of this chapter are:

- channel reads (`IN?x`)
- channel writes (`OUT!x+y`)
- assignments (`a:=b+c`)
- sequential composition (`b:=a+x ; c:=b+y`)
- parallel composition (`a:=b+x || c:=d+y`)

Figure 3.2 shows the Haste specification of a simple program, a single stage FIFO. The program has an input channel IN, through which it receives data items from the environment, and an output channel OUT, through which it transmits results to the environment. Each channel consists of a pair of request-acknowledge wires along with the data wires. In the specification, `x` is a storage variable. The main construct in the body of the specification is a `forever do` loop that performs the following actions repeatedly: (i) read a value from channel IN and store it into variable `x`; then (ii) write the value stored in `x` to the output channel OUT.

Performance Limitations. As you may recall from Chapter 2, given a specification, the Haste compiler syntactically maps each construct onto a predefined library component to generate a hardware implementation, as shown in Figure 3.2. Therefore,

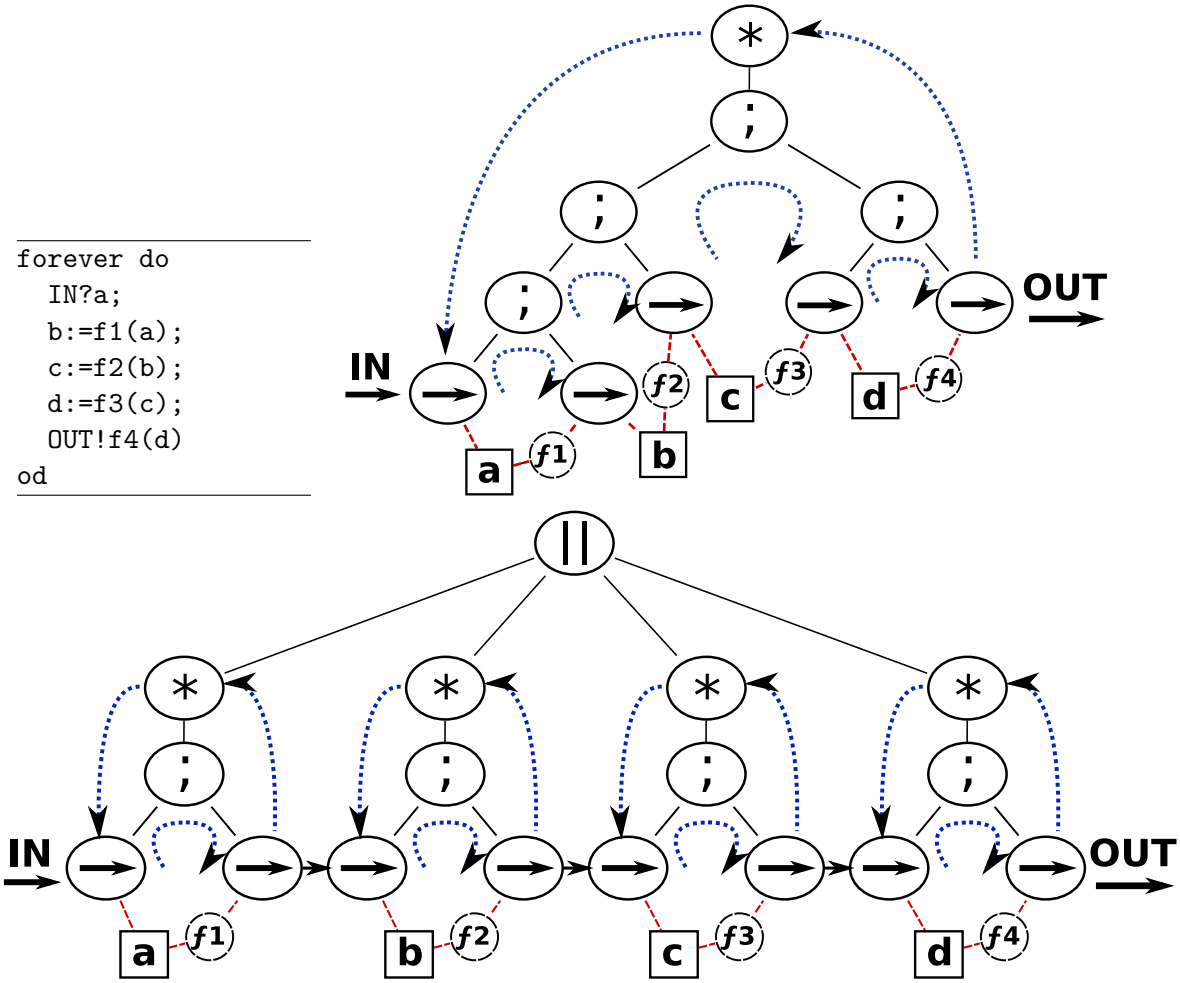

Figure 3.3: Control dominated (top) vs. data-driven (bottom)

as the number of statements increase in the code snippet in Figure 3.2, the size of the control cycle increases, resulting in a higher latency block. Several handshakes in the control tree may be required before an action can occur. As a result, the performance of the system suffers. We can describe this situation as "control-dominated."

3.2.2 Asynchronous Pipelining

To overcome the performance limitation of large control cycles, a designer can introduce pipelining to reduce the control overhead. Figure 3.3 illustrates how control overhead is reduced in this situation. Pipelining replaces a large control tree with a

forest of smaller trees governing the actions in the system. These actions are now initiated by channel communications directly, as opposed to sequenced by a complex controller. Thus, the single long control cycle in the original tree can be replaced by several relatively smaller control cycles local to individual computation blocks, thereby resulting in significantly better throughput.

In channel actions between stages, all of the variables that will be accessed in the remainder of the pipeline must be communicated. We will refer to this set of variables as the "context" of a stage.

In a *data-driven architecture,* the entire context is passed from one stage to the next, irrespective of whether an individual value is needed in the next stage as long as it is needed in some subsequent stage. Thus, once a result is produced, it may go through a number of intermediate stages before reaching the consumer. While this approach may seem somewhat expensive in terms of area and energy consumption, it is quite commonly used in practice due to its simplicity, such as in pipelined microprocessors.

In a *data-flow architecture,* by contrast, concurrency may be further increased by allowing data to propagate directly to stages in which it is used (Budiu, 2003). As a result, the pipeline is typically forked off into many branches, which often re-converge. However, such an approach introduces additional challenges: the performance can suffer if branches are not properly balanced (*i.e.*, not "slack-matched" (Beerel et al., 2006)) as discussed in Section 2.1.3, potentially resulting in throughput that may be *worse* than that of the slowest stage because of stalls caused by mismatched branching. In order to avoid slack-mismatch issues, this work presented in this chapter utilizes a data-driven paradigm instead of full data-flow (which will be considered in Chapter 5).

3.2.3 Previous Work

Much research has been done in the domain of performance optimization for high-level specifications. The most relevant approach to ours is *spatial computation*, introduced by Budiu et al. (Budiu, 2003). In spatial computation, ANSI C specifications are transformed directly into hardware, incorporating a number of optimizations that aim to enhance concurrency. However, their work fundamentally belongs to a different domain—ANSI C software specifications—which is less general than the behavioral specifications targeted in this work. In particular, C specifications, unlike Haste, do not allow explicit communication via channels between processes to be modeled, whereas such communication is key to modeling complex asynchronous systems. In addition, fork-join style of concurrency cannot be explicitly specified by a designer in C; such concurrency again is central to many asynchronous system specifications. Finally, their approach does not consider pipelining of atomic units, such as adders and multipliers, which is a key contribution of our approach.

Teifel et al. (Teifel and Manohar, 2004) and Wong et al. (Wong and Martin, 2001) have introduced approaches that translate specifications written in CHP (Martin et al., 1997) (a variant of CSP (Hoare, 1985)) into pipelined implementations. While these approaches allow channel communication, their communication models can be restrictive, *e.g.*, requiring that channel actions be unconditional or occur at most once in the body of a process. In contrast, this approach allows a more general framework for communication optimization.

Two recent approaches target conversion of behavioral specifications between CDFG and Haste/Balsa representations (Nielsen et al., 2004; Jensen and Nielsen, 2007). Their goal is to leverage mature synchronous tools that are capable of performing resource scheduling, allocation and binding (under physical constraints), thereby getting around the limitations of the Haste and Balsa tools, which lack such capability. These ap-

proaches also include some peephole optimizations, but do not aim to enhance system-level concurrency. In contrast, the data-driven approach specifically targets concurrency enhancement through pipelining, parallelization, and arithmetic and communication optimizations, with the goal of high system performance.

Many other approaches focus on low-level optimizations at the circuit and handshake level (Plana et al., 2005), (Chelcea and Nowick, 2002) to improve throughput. However, solely using lower-level optimizations fails to take advantage of concurrency that can be gained at a higher level. The proposed approach does not preclude these optimizations, and in most cases these can be performed in an orthogonal fashion.

3.3 Basic Approach

In this section I describe how the source-to-source compiler optimizes code through parallelization and pipelining. I will first discuss how performance optimizations change the hardware structure of the system and give an overview of the optimizations that are performed at a source level. I then discuss how parallelization and pipelining are performed in the proposed approach.

3.3.1 Method Overview

3.3.1.1 Hardware Level

Figure 3.4 shows an example of synthesized code and its representation in hardware. Each small block in the figure represents a basic datapath operation. Similar to the case in Figure 3.3, control delays dominate, and the throughput obtained is rather low. The only channel communications that occur are with the environment. In essence, the original code is synthesized into a single, unpipelined, high latency block. The throughput of the system is solely determined by the latency of this unpipelined block.

```
proc(IN?chan byte & OUT!chan byte).
forever do
    IN?a;
1:  b:=a*2;
2:  c:=b+5;
3:  d:=a+b;
4:  e:=c+d:
5:  f:=d*3;
6:  g:=f+e;
    OUT!g
od
```

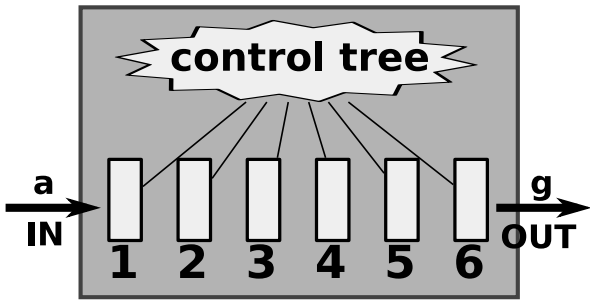

Figure 3.4: Original implementation

```
proc(IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  b:=a*2;
  (c:=b+5 ||
   d:=a+b);
  (e:=c+d ||
    f:=d*3);
  g:=f+e;
  OUT!g
od
```

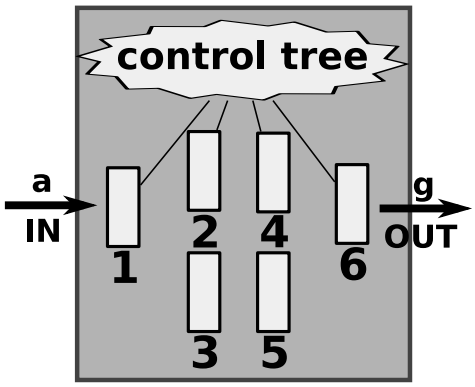

Figure 3.5: Parallelized implementation

Consider now the case where some operations in the original code are parallelized, as shown in Figure 3.5. The resulting circuit is still control driven, and again channel communication is only performed with the environment. The control tree is the same size, however, some parallel blocks replace sequential blocks in the tree. As a result, the latency of the full system is reduced, but the throughput is still determined by the latency of the whole system. The system still acts as a single stage, but with lower latency and higher throughput than that of the previous implementation.

The result of *pipelining* the original implementation is shown in Figure 3.6. Each operation now has its own individual latch to store data and channels to connect it with other stages. Note that the control cycle at each stage is considerably shortened. This data-driven pipeline has multiple, low-latency stages, yielding an increase

```
forever do (IN?a;
  OUT!<<a,a*2>>) od
...
forever do (IN?<<a,b>>;
  OUT!<<a,b,b+5>>) od
...
forever do (IN?<<a,b,c>>;
  OUT!<<c,a+b>>) od
...
forever do (IN?<<c,d>>;
  OUT!<<d,c+d>>) od
...
forever do (IN?<<d,e>>;
  OUT!<<e,d*3>>) od
...
forever do (IN?<<e,f>>;
  OUT!<<e+f>>) od
```

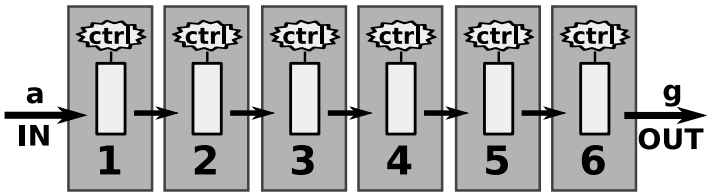

Figure 3.6: Pipelined implementation

```
forever do (IN?a;
  OUT!<<a,a*2>>) od
...
forever do (IN?<<a,b>>;
  OUT!<<b+5,a+b>>) od
...
forever do (IN?<<c,d>>;
  OUT!<<c+d,d*3>>) od
...
forever do (IN?<<e,f>>;
  OUT!<<e+f>>) od
```

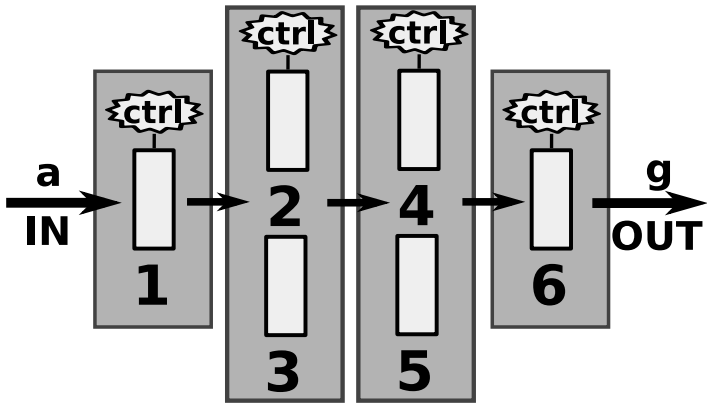

Figure 3.7: Parallelized and pipelined implementation

in system throughput. In this case, the throughput is limited by the cycle time of the slowest stage, rather than the latency of the whole system. Therefore, the throughput is increased, though possibly at the cost of some latency overhead.

By performing both optimizations, *parallelizing then pipelining,* the circuit of Figure 3.7 is produced. This circuit benefits from the reduced latency of parallelization, as well as the increased throughput of pipelining. Conversion to this design is the goal of our transformations.

3.3.1.2 Source Level

We now give an outline of how our algorithm is applied at the source level. Starting with a piece of straight-line, sequenced code, Figure 3.4, we transform it into the highly concurrent code of Figure 3.7.

The first step in the algorithm is to group the statements in a block of code that can be performed in parallel. In the code fragment in Figure 3.4, the assignments to variables `c` and `d` can be performed in parallel, and `e` and `f` can be performed in parallel, producing the circuit shown in Figure 3.5. This step performs simple instruction-level parallelization, reducing latency. However, we can further to increase performance by performing pipelining as well.

To pipeline, a channel is placed between every parallel grouping. This channel communicates the context for this dataset. A corresponding code fragment is shown for the pipeline stages for the assignments to `b`, `c`, and `d` in Figure 3.7 (note that the procedure headers have been removed for clarity).

3.3.2 Class of Specifications Handled

3.3.2.1 Handling Specifications with Cycles

Even though the example of Figures 3.4–3.7 shows a code snippet that is acyclic, the proposed optimization approach is fully capable of handling specifications with cycles. In particular, the approach is hierarchical: at each level of code hierarchy, a compound statement (*e.g.*, `if-then-else, while,` etc.) as well as any statement group with cyclic dependencies is treated as an atomic statement for the purpose of performing parallelization and pipelining. The compound statement or cyclic group can then be separately optimized when traversing the next lower level of hierarchy.

An example of how a cycle in a specification is handled is shown in Figure 3.8. In

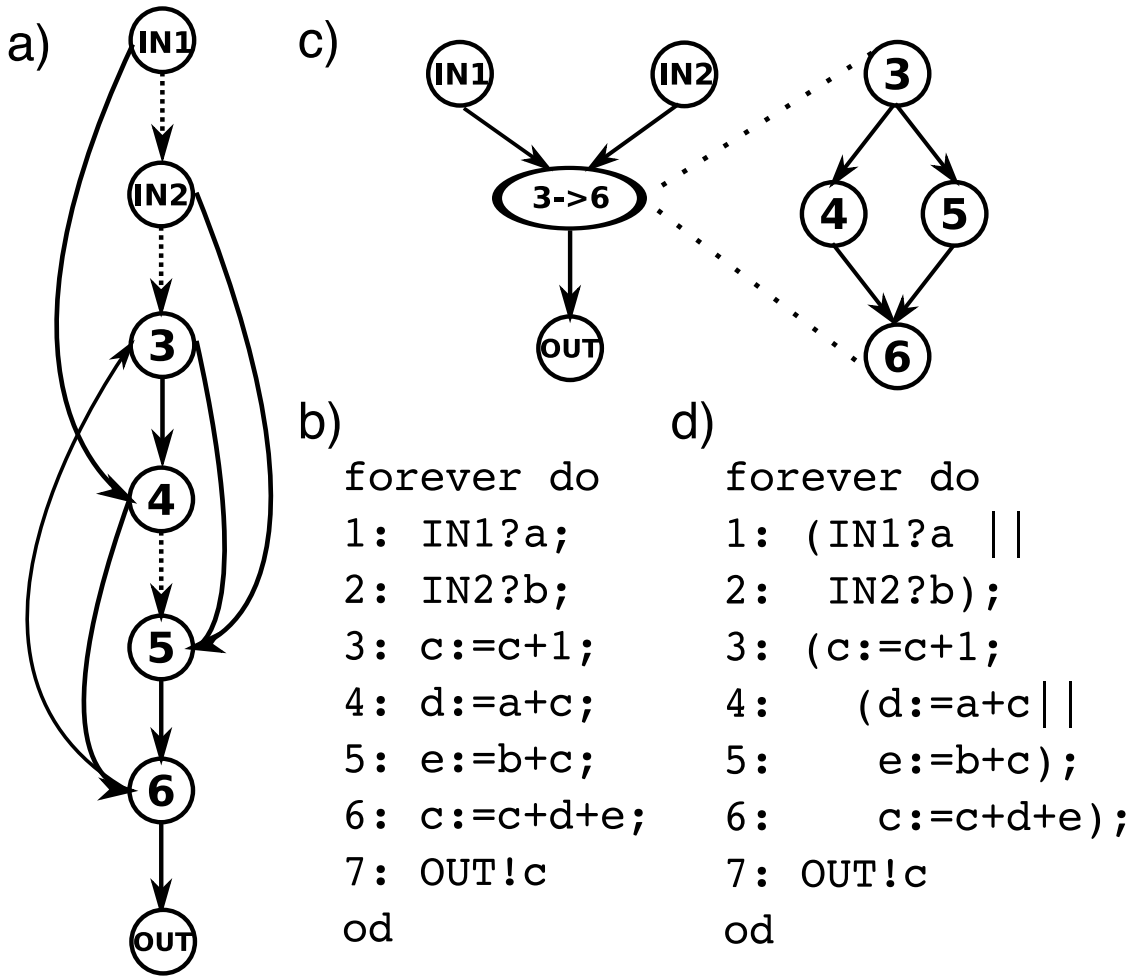

Figure 3.8: Handling cycles: a) cyclic dependency graph, b) corresponding source, c) treating cycle as atomic statement, and d) optimized source.

the code fragment of Figure 3.8b, the variable c is reused across iterations of the loop, creating a backwards dependency from statement 6 to statement 3. As a result, a cycle is introduced in the dependency graph (Figure 3.8a). Alternatively, cycles can also be caused by loop constructs such as `for` and `while`.

The optimization approach operates hierarchically, and starts at the top level where it treats the cycle temporarily as a single node (Figure 3.8c), and parallelizes the body of the outermost block. Subsequently, the code within the cycle is parallelized. The resulting code is shown in Figure 3.8d.

While parallelization is performed at all levels of hierarchy, the pipelining trans-

formation is typically not performed inside a cycle. Because loops allow only a single token to be present in them at a time without a more complex approach, pipelining will not provide a performance benefit. Therefore, in the proposed approach, pipelining is performed at the top level of the hierarchy, and further down the hierarchy into acyclic code blocks, until a cycle is encountered.

It is important to note that even if pipelining is not performed for a loop block, the loop's performance may still be improved by parallelization. In Figure 3.8d, parallelization can shorten the latency of the cycle, which translates to both shorter latency and shorter cycle time at the next higher level of the hierarchy. This topic is dealt with in detail in Section 3.4.

3.3.2.2 Set of Language Constructs

The full set of Haste constructs is permitted in the proposed approach; however, the transformed specification will be equivalent to the original one only if the original specification satisfies the conditions for slack elasticity (Manohar and Martin, 1998a). In particular, loops, conditionals, case statements, function calls, sequential, and parallel constructs are all supported. Similar to Figure 3.8, more complex constructs are collapsed into a single node and each of these are further hierarchically parallelized. Similarly, pipelining is hierarchically performed until blocks with cyclic dependencies are encountered. Details on the handling of some of these complex constructs (specifically, conditionals and loops) are presented in Section 3.4.

3.3.3 Parallelizing Transformation

At the core of the parallelization transformation is dependence analysis. This subsection briefly describes how this analysis is performed, then shows how the results allow the compiler to modify the program to increase concurrency.

```
expl=proc(IN?chan byte &
          OUT!chan byte).
begin
  & a,b,c,d,
    e,f,g: var byte
  | forever do
      IN?a;
1:    b:=a*2;
2:    c:=b+5;
3:    d:=a+b;
4:    e:=c+d:
5:    f:=d*3;
6:    g:=f+e;
      OUT!g
    od
end
```

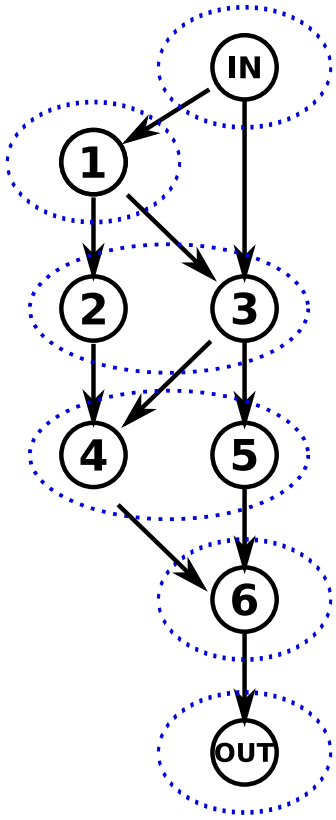

Figure 3.9: Precedence graph with parallel groupings

Figure 3.9 shows a sample precedence graph. After the graph is generated, a topological sort of the graph is performed. (As described in Section 3.3.2.1, any cycles encountered as treated as atomic statements for the purpose of this sorting.) Each statement that has no input edges (dependencies) is placed into the first grouping of parallel statements. These statements are then removed from graph, along with any edges they produce. Next, all statements that have no input edges are placed into the second grouping, then their edges are removed. The process repeats iteratively until all the statements are placed into a grouping.

The compiler then generates a new sub-tree in which parallel groupings are children of a parallel (||) construct. The parallel groupings are in turn combined using sequencers(;).

The compiler employs variable renaming to achieve greater concurrency enhance-

ment. Thus, if a second assignment to a variable occurs within a block of code, the target location is renamed, along with any future accesses. As a result, write-after-read and write-after-write dependencies are removed from the graph.

Theoretically, even further concurrency can be achieved by using a partial ordering of the statements, resulting in a full data-flow specification at the cost of greater forking and joining. However, with increased branching, challenging slack matching issues may arise, and without an effective pipeline-balancing approach, the resulting specifications can exhibit reduced throughput. This issue was the motivation behind adopting the simpler data-driven approach in this chapter, instead of a full data-flow approach, like that of Chapter 5.

3.3.4 Pipelining Transformation

Pipelining is an orthogonal process to parallelization; it can be performed on code that is sequential or has already been parallelized. This subsection discusses how pipelining is achieved for such a specification. In particular, we focus on the process of generating channel communications between stages using IN and OUT sets. In this section, let us assume an input specification follows the following basic pattern: read from a set of input channels, perform a computation, and write to a set of output channels.

To begin the pipelining transformation, the compiler first breaks every group of statements delimited by a sequencer (`;`) into its own pipeline stage. In source code, each stage will be represented by a statement block in which the initial statement is a channel read and the final statement is a channel write. The channel read accepts the context from a prior stage; the channel write transmits the updated context to a subsequent stage.

To complete the transformation, the correct context for each stage must be determined. First, the compiler visits each stage, building a list of the variables accessed

(VAR_x) by the group of statements in that stage. Next, the compiler generates the IN set for each stage, which consists of all the variables in use prior to or within the stage. IN sets are determined using the following productions, where x indicates the stage number:

$$\text{IN}_x = \text{IN}_{x-1} \cup \text{VAR}_x, \quad \text{IN}_1 = \emptyset$$

The compiler then determines the OUT set for the stage: the set of all variables accessed in subsequent stages. A similar production is used (n indicates the final stage in the pipeline):

$$\text{OUT}_x = \text{OUT}_{x+1} \cup \text{VAR}_{x+1}, \quad \text{OUT}_n = \emptyset$$

Two important observations are made by comparing the IN and OUT sets for each stage. First, if a variable is contained in a stage's IN set but not contained in its OUT set, that variable will be accessed in this stage, but will not be accessed in any future stages. Therefore, the variable does not need to propagate beyond this stage.

Second, a variable that exists in the OUT set of a stage but not in its IN set is being used for the first time in the next stage. If the variable is read in the next stage, the read can be replaced with the variable's initialization. In this case, the current stage sends the initial value of the variable, or zero if the variable is declared without an initialization. If the variable is only written in the next stage, the current stage does not need to communicate a value for the variable, since it will merely be overwritten.

Using the IN and OUT sets for each stage, the context for each stage is determined. For a stage x, the set of variables in the stage's context is the following:

$$\text{context}_x = \text{OUT}_{x-1} \cap \text{IN}_x$$

The variables that must be communicated on its output channel are:

$$\text{context}_{x+1} = \text{OUT}_x \cap \text{IN}_{x+1}$$

Once the contexts have been computed for each stage, channel reads are inserted for each stage after the first. Likewise, a channel write is inserted for all stages except

the last. In operation, each stage will read in the values of each variable needed in this stage or a future stage. The stage will then perform operations on these variables using the concurrent statement grouping associated with the stage. If a variable is modified, the output channel will transmit an expression containing the updated value. If unmodified, the output channel will merely transmit the original value of the variable. The channel read, variable modification, and channel write are then nested within a `forever do` loop, creating a pipeline stage. This process is followed for each stage to create a complete data-driven pipeline.

3.4 Advanced Techniques

This section describes several advanced approaches for improving the performance of a specification. I will first discuss several methods for optimizing arithmetic operations, then describe how conditionals and loops are handled. Finally, I present an approach for increasing concurrency in the presence of channel communication.

The two techniques of arithmetic optimization and communication reordering are key contributions for enhancing performance. The former pushes concurrency enhancement to the sub-statement level, whereas the latter technique enlarges the space of solutions by carefully allowing communication between distinct modules to be reordered without the introduction of deadlocks.

3.4.1 Arithmetic Optimization

While the basic approach can potentially obtain substantial speedup by optimizing code at the statement level, further improvement is possible by optimizing at the sub-statement (*i.e.*, expression and operator) level. I will now describe three methods for optimizing arithmetic computation: *expression tree balancing,* which can reduce the

latency of a series of arithmetic operations; and *expression pipelining* and *operator pipelining,* which can improve the throughput of a system.

3.4.1.1 Balancing Expression Trees

Many languages, including Haste, rely on both operator precedence and a left-right expression ordering to determine how sub-expressions are evaluated. Therefore, expression trees produced by the parser can be unbalanced, even linear in some cases. Re-factoring sections of the tree by taking advantage of operator associativity can lead to more balanced expression sub-trees, and introduce additional concurrency into the specification. This optimization reduces the overall depth of the tree, improving both latency and throughput for a statement.

For example, the expression `a+b+c+d` initially requires three sequential addition stages. Tree balancing converts the expression to `(a+b)+(c+d)`, which requires only two sequential addition stages since evaluating expressions `a+b` and `c+d` can be performed in parallel. As this optimization may change the meaning of a program in exception cases (*e.g.*, overflow), it is provided as an option to the user. In effect, this optimization can be regarded as parallelization pushed to the granularity of arithmetic expression evaluation.

3.4.1.2 Expression Pipelining

While balancing expression trees can provide some benefit to throughput if the latency of a statement is reduced, a statement with a large expression tree can still be a major bottleneck in the specification. By pipelining a computationally complex expression tree, significant gains in throughput can be attained.

To perform expression pipelining, we can divide the original statement's expression tree into several smaller assignments. Each atomic sub-expression becomes its own

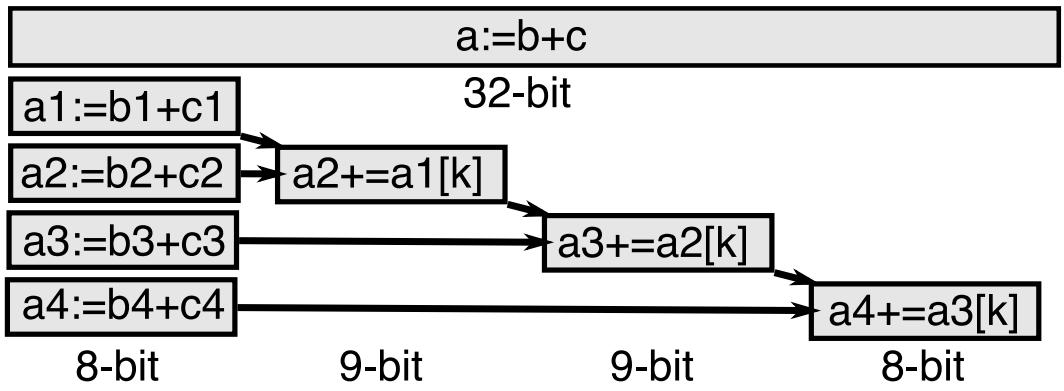

Figure 3.10: Operator pipelining via source code

individual assignment with a single arithmetic operation. For example, consider a statement with a complex expression tree: `a:=((b+c)*d)+e`. Through optimization, a series of simple statements are produced: `t1:=b+c; t2:=t1*d; a:=t2+e`. In essence, expression pipelining replaces one high-latency pipeline stage with multiple low-latency stages, improving throughput.

3.4.1.3 Operator Pipelining

Further gains in throughput are achieved by decomposing and pipelining *individual arithmetic operators*. Haste implementations, however, pose a special challenge to correctly pipelining an arithmetic function. In particular, naïvely replacing an arithmetic unit such as a combinational adder with a pipelined adder circuit does not yield any throughput improvement. This is because the controller associated with the stage that contains the combinational adder allows only one token in that stage at a time. Therefore, in order to pipeline the adder, that stage's controller itself must be modified; this modification is more easily performed at the source level.

In order to effectively pipeline arithmetic, control must be distributed to individual stages. This is achieved in source code by breaking down the arithmetic operation into several smaller assignments. Figure 3.10 illustrates how pipelining using finer

```
expl=proc(IN?chan byte & OUT!chan byte).
begin
  & a,b,x,y
  | forever do
      IN?a;
      IN?b;
      if a>b
         y:=y-1
      else
         x:=x+1;
         y:=y+1;
      fi;
      OUT!x+y
    od
end
```

```
forever do
  IN?a;
  IN?b;
  x:=if a>b then x
        else x+1 ||
  y:= if a>b then y-1
        else y+1;
  OUT!x+y
od
```

Figure 3.11: Replacing conditionals with conditional assignments

granularity operators is performed for a 32-bit addition. First, each operand is broken down into four 8-bit operands. These operands are then fed into four 8-bit adders in parallel to produce 9-bit results (including the carry out). The partial results are then combined sequentially to produce the final value.

By implementing operator pipelining at the source level, not only is throughput increased, but latency can be improved as well. In particular, the source-level decomposition of individual operators into several smaller operations affords new opportunities for parallelization (*i.e.*, exploiting parallelism among stages of distinct pipelined operators). Performing this task by hand is a time-consuming and error-prone operation, but is well-suited to a source-to-source compiler.

3.4.2 Conditional Optimization

Not all code the user wishes to synthesize is linear in nature, as conditionals (if-then-else) are often present. There are many options to handle these breaks in linearity.

Conditional Assignment. If both branches of a conditional consist solely of variable assignments, *i.e.*, no channel communications or loops exist in either branch, conditional assignment of variables is the preferred method. To perform a conditional

assignment, the assignments in either branch are removed and replaced with a tertiary assignment outside of the conditional. The form is as follows:

$$\texttt{var:=if bool then exp}_{then}\ \texttt{else exp}_{else}$$

Consider the code in Figure 3.11. In the `else` branch, the variable `x` is assigned `x+1`. In the `then` branch, no assignment is made. The assignment can be removed from the loop and replaced with a conditional assignment:

```
x:=if a>b then x else x+1
```

If assignments are made in both branches, such as for variable `y`, the same idea applies:

```
y:= if a>b then y-1 else y+1
```

If the boolean condition itself is a function of variables modified in either branch, the boolean must be computed and stored prior to performing the conditional assignments in order to preserve the semantics of the conditional. Finally, if several writes to the same variable occur in both branches, variable renaming is employed.

Early Decision. A second option for handling conditionals is early decision. Early decision (Figure 3.12) is used when either branch contains a channel communication or internal loop. It is necessary that the pipeline be split into two branches to handle this situation: one containing the 'then' branch, the other containing the 'else' branch. Two additional stages are introduced: one that forks the branches prior to execution, and one that merges them after execution.

In early decision, the value of the conditional's boolean is computed prior to entering either branch, just as it would in a normal system. After the computation, the fork stage decides the path to which the context should be sent. The context is then operated on by the proper branch, and then accepted by the merge stage to be sent out.

If the two paths are poorly matched in terms of slack and forward latency, early decision may result in out-of-order execution of consecutive datasets. In some cases,

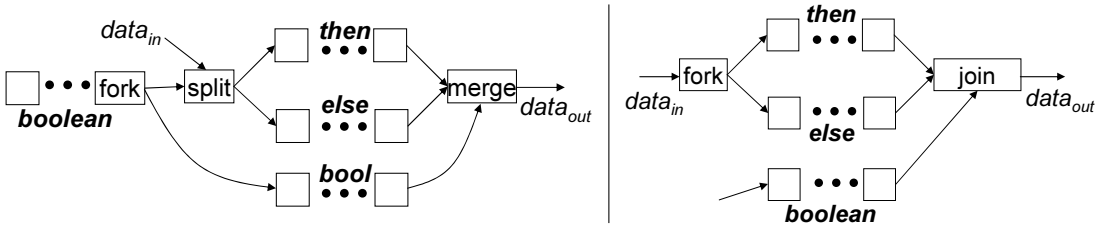

Figure 3.12: Early and late decision in conditionals

such as computer graphics and networking, out-of-order execution is allowable. If, however, correct order is required by the user, a third boolean path (with buffering) is introduced between the fork and join stages to indicate which branch the join stage should read from to preserve execution order.

Late Decision. A final alternative, late decision (Figure 3.12), can be applied in the case where either branch contains an internal loop, but cannot be applied when channel communication is performed by the branches. In late decision, both branches are executed concurrently, and the correct result is later chosen based on the boolean outcome. This is a form of speculation. Because both paths are taken regardless of the value of the boolean, channel communication inside the conditional is disallowed; otherwise unnecessary channel actions could potentially occur, thereby compromising the system's correctness.

In late decision, the pipeline must be split into three branches, two for `then` and `else` and one for the boolean value. Again, a fork and a join stage must be included in the pipeline. At the join stage, all three branches have completed computation. The join stage selects the context from the correct branch using the boolean value and forwards it, discarding the context from the incorrect branch.

Late decision suffers from poor energy consumption and can also limit throughput if the branches are not slack-matched. However, the latency of the conditional can be reduced if the boolean takes a long amount of time to compute. Early decision, in comparison, has the advantage of high throughput even if the paths are not slack

matched.

3.4.3 Optimization of Loops

Loops are a significant roadblock for designers aiming for high throughput specifications. In typical implementations, new data items cannot enter the loop until the current item exits, a loop effectively acts as a single, high-latency stage. Pipelining the internals of a loop provides no benefit if the loop contains a single token, and in fact decrease performance of a system due to latency overheads.

The designer does have three potential options for increasing performance: (i) statement parallelization, (ii) expression tree balancing, and (iii) loop unrolling. All three optimizations have the potential for reducing latency, and thus improving the throughput of the loop.

I have previously discussed how statement parallelization and expression tree balancing are performed in the compiler. The third option, loop-unrolling, is performed in the same manner in this domain as in software, and provides the potential for statement interleaving across loop iterations. Both full and partial unrolling can be performed. However, since loop unrolling essentially replicates hardware, this optimization comes at a cost of area. I have performed full unrolling for one of the benchmarks in Section 3.5.

While traditional design methods typically allow only a single token inside an algorithmic loop, my research collaboration with Gill et al. (Gill et al., 2006) introduced a novel approach to implementing loops that can operate on multiple tokens concurrently. This technique, called *loop pipelining,* correctly handles the flow of control and all data dependency challenges created by allowing multiple tokens inside a loop, thereby significantly increasing throughput.

3.4.4 Communication Optimization

The presence of channel communication introduces new challenges for code rewriting. In particular, simply relying on dependence analysis is not sufficient to determine the space of legal re-orderings and groupings of statements. The reason is that communication involves not only flow of data but also control synchronization. In fact, sometimes communication actions omit data altogether, and are simply used to synchronize two modules. Naïvely reordering communication actions can introduce a deadlock into the system.

A safe but suboptimal approach is to prevent re-ordering of channel actions altogether. Instead, I will introduce a more flexible approach for handling communication which includes a careful analysis of computation and communication actions within and across modules, to determine the space of legal code rewritings. As a result, the proposed approach allows more opportunities for concurrency enhancement.

In this subsection, I will first illustrate how channel communication makes code rewriting challenging, and then describe the proposed solution.

Avoiding Deadlock. Let us start with a simple example that illustrates the effects of re-ordering a pair of channel communications. Consider module 1 that consists of three channel communications: `A?a; B!a; C?c`. Here we see a data dependence only between the first two communications. Suppose the counterpart channel communications for A, B, and C are in three distinct modules with no other channel actions. The channel action on C is ready to be performed earlier in the module, and is only prohibited by the sequencing of this module. In this example, re-ordering can increase the concurrency and reduce the latency of the module. One approach would be to parallelize the channel actions: `(A?a||C?c); B!a`.

However, suppose that the counterpart channel actions for A, B, and C are all contained in a single module, in which they are all sequenced: `(A!a;B?b;C!b)`. Here

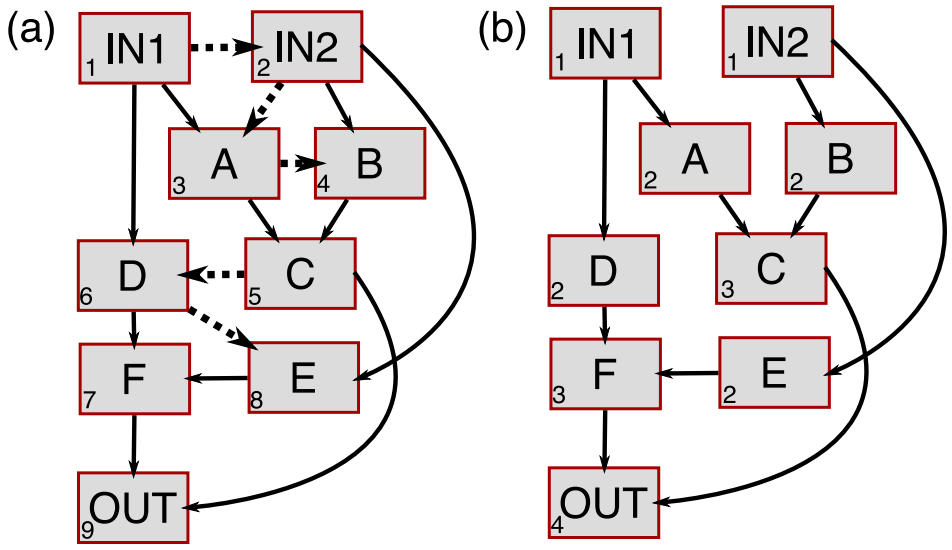

Figure 3.13: Communication optimization via directed graph

a data dependence exists between the channel communications on B and C. By re-ordering the channel actions in the original module, a deadlock has been introduced.

Solution Overview. To be able to safely optimize, the designer must first determine the flow of data across channels. This goal can be accomplished by building a directed graph, as shown in Figure 3.13a.

For each channel in the specification, a node is introduced in the directed graph. A dashed edge is drawn between two nodes if the two channels have sequenced actions in any module (`IN1?a; IN2?b`), while full edges are drawn between nodes that have a data dependence. Data dependencies can either occur directly: `F?f; OUT!f`, or due to statements sequenced between the two communications: `A?a; c:=a+b; C!c`.

By removing all of the sequencing arcs in the original specification, the directed graph in Figure 3.13b is produced. The compiler can use the optimized directed graph to generate a new behavioral specification by creating parallel groupings, similar to Figure 3.9. More generally, the compiler may re-order and parallelize communications as long as two criteria are met: (i) by sequencing channel actions in a module, new

edges must be inserted in the graph, and (ii) the resulting graph may not contain a cycle, as this indicates a deadlock has been introduced.

While several orderings are possible from the outlined rules, the proposed approach is to parallelize all channel communications in a grouping, and ensure that all groupings are performed in sequence. This precludes the opportunity for introducing deadlock in a system.

Performance. In general, communication optimization is focused on increasing the concurrency of a specification. By performing pipelining balancing, the opportunity exists for improvement in throughput and latency. One scenario where throughput can be improved is when stalling occurs in the original specification due to unnecessary sequencing of channel actions. A rough indication of latency can be determined by counting the number of nodes in the longest path in the directed graph; in the case of Figure 3.13, the longest path is reduced from 9 nodes to 4 nodes.

3.5 Results

The optimizing compiler presented in this chapter was written in Java and executed on a 2.16GHz Core Duo processor system with 2GB of RAM. Execution time for the compiler was 0.2 to 2 sec in all cases, with the exception of one example (ADD) which took 2 min. These run-times are quite short for a Java implementation, and even in the slowest case an order of magnitude shorter than the subsequent compilation step by the Haste compiler.

Experimental Setup. Each example was designed and simulated using the Haste design flow, described earlier in Section 3.2. All designs were synthesized to the gate level using a generic tech library supplied with the tools.

Eight different benchmarks were selected to illustrate the effects of our approach:

(i) FIR: a simple FIR filter that performs a weighted average of the previous eight inputs, (ii) ALU: an ALU based on the communication example of Section 3.4 that receives two inputs and produces their sum and difference, (iii) ADD: a 64-way adder tree derived from a real-world Boeing project, (iv) UTEA: an unrolled version of the Tiny Encryption Algorithm, (v) ODE: a differential equation solver, (vi) ROOT: an specification that performs the square root of an input, (vii) QUAD: a specification that determines if the quadratic roots of two input polynomials are interleaved, and (viii) LTEA: a second version of the Tiny Encryption Algorithm, without unrolling.

The first four specifications (FIR, ALU, ADD, UTEA) consist of straight line code, while the second four specifications (ODE, ROOT, QUAD, LTEA) include one or more loops. For each example the compiler created several transformed specifications: parallelized, pipelined, and both parallelized and pipelined. Furthermore, arithmetic pipelining at 32, 16, 8, and 4 bit granularities was performed using three specifications in addition parallelization and pipelining.

The ROOT, QUAD, and LTEA specifications contain very tight loops with little room for internal parallelization. ODE's loop, however, contains several sequenced operations that have the potential to be parallelized. QUAD contains two ROOT loops in sequence with no data-dependencies between the loops, as well as several unpipelined 64-bit multiplications in each version.

Performance. Table 3.1 shows the cycle time and latency for each specification with the arithmetic pipelining option disabled. Throughput generally increases (cycle time generally decreases) from left to right in the table, most notably for the four straight-line code examples. The four examples with loops also show throughput improvement, although in most cases only due to parallelization which tightens the loop body. As expected, pipelining alone had no benefit on the examples with loops except for QUAD, which consists of not just one large loop, but two sequential ROOT loops

along with several other complex sequential operations. As a result, QUAD was significantly sped up (8x) by pipelining. Overall the greatest throughput improvements were for the straight-line examples: from 2.2x for ALU, to 14x for FIR, 23x for UTEA, and 59x for the adder tree (ADD).

Latency is generally reduced (or remains unchanged) after performing parallelization, but is usually increased after performing pipelining. As a result, the latency of parallelized and pipelined specifications is increased in some cases and decreased in others: 1.2x longer for UTEA, but 8.4x shorter for ADD.

The table shows a few intriguing anomalies. The first is the reduced latency of FIR and ADD after performing parallelization and pipelining, when compared to the parallelized version. This reduction is due to expression tree balancing, which was only enabled when both parallelization and pipelining were performed. A second anomaly is the reduced latency of FIR after performing pipelining. FIR's original specification required flip-flop variables due to auto-assignment. The pipelining optimization was able to replace these flip-flops with lower latency latches, thus reducing the overall latency of the specification.

A second set of results shows the effect of arithmetic pipelining on the first three straight-line code examples. As shown in Table 3.2, *an additional 5.2x improvement* was achieved by pipelining arithmetic in addition to the previous optimizations, as long as the pipeline is not limited by other higher granularity stages such as those containing loops.

Area and Design Effort. Table 3.3 gives numbers for the total area of synthesized circuits, as well as the number of lines for each specification as an indication of design effort. By comparing the number of lines in the original specification to the number of lines in the parallelized and pipelined specification, we can get a sense of how much effort was saved by using our tool versus performing these optimizations by hand. This

amount averages around 3.3x, and is 8.8x in the best case (UTEA). In terms of area, the original and parallelized specifications have similar numbers, while the pipelined version generally has significantly more area. The parallelized and pipelined version falls somewhere between the original and pipelined versions in most cases.

3.6 Conclusion

This chapter proposed a source-to-source compiler to increase the performance of a specification while maintaining ease of design. The automated approach yielded improved throughput for a full suite of specifications, up to 59x in one case (293x with arithmetic pipelining). By performing these optimizations using an automated tool rather than by hand, design effort was reduced by up to 8.8x.

While the approach presented in this chapter targets high-performance data-driven pipelines, minimizing area is often a designer concern. Therefore, I will propose an entirely different approach in the next chapter: a resource-shared approach targeting reduced-area implementations.

		Optimization Technique							
		Original		Parallel		Pipelined		Parallel+Pipelined	
		Cycle Time (ns)	Latency (ns)	Cycle Time (ns)	Latency (ns)	Cycle Time (ns)	Latency (ns)	Cycle Time (ns)	Latency (ns)
Acyclic	FIR	190.6	381.2	99.85	199.7	13.37	227.2	13.37	93.59
	ALU	30.03	90.09	15.02	45.06	13.28	199.3	13.96	97.78
	ADD	790.3	790.3	790.3	790.3	13.37	1698	13.36	93.54
	UTEA	343.1	343.1	343.2	343.2	14.74	457.2	14.74	442.5
Cyclic	ODE	1218	1218	576.8	576.8	1246	4985	591.3	1774
	ROOT	132.4	132.4	118.6	118.6	137.3	824.3	125.7	503.1
	QUAD	1497	1497	748.0	748.0	187.0	4115	187.0	1870
	LTEA	341.6	341.6	341.6	341.6	342.4	1369	342.4	1369

Table 3.1: Performance of original and transformed specifications

	Optimization Technique / Granularity of Operator Pipelining											
	Original		Parallel+Pipelined		32-bit		16-bit		8-bit		4-bit	
	CT (ns)	Stages (N)	CT (ns)	Stages (N)	CT (ns)	Stages (N)	CT (ns)	Stages (N)	CT (ns)	Stages (N)	CT (ns)	Stages (N)
FIR	190.64	2	13.37	7	7.75	15	4.86	23	3.41	39	2.69	71
ALU	30.03	3	13.96	7	7.74	11	4.86	15	3.41	23	2.69	39
ADD	790.31	1	13.36	7	7.75	19	4.86	31	3.41	55	2.68	103

Table 3.2: Performance improvement through operator pipelining

		Optimization Technique							
		Original		Parallel		Pipelined		Parallel+Pipelined	
		Area (μm)^2	Lines (n)	Area (μm)^2	Lines (n)	Area (μm)^2	Lines (n)	Area (μm)^2	Lines (n)
Acyclic	FIR	12365	45	11225	47	25204	209	11869	88
	ALU	2441	46	2482	56	5913	227	3495	83
	ADD	44610	76	44610	76	658913	1661	53248	217
	UTEA	32731	61	32736	61	31225	551	30754	540
Cyclic	ODE	1750	28	1600	34	2292	91	2010	84
	ROOT	928	27	924	33	1245	94	1122	76
	QUAD	6213	54	5880	70	9412	294	6969	170
	LTEA	5268	36	5268	36	6475	86	6475	86

Table 3.3: Area and code length

Chapter 4

Resource-limited Design: Unpipelined

While the approach presented in Chapter 3 can lead to very high-performance circuits, the conversion to a data-driven implementation comes at the cost of area. Since each operation is explicitly mapped to its own individual function unit, no sharing of resources is performed. Therefore, we now turn to the other end of the spectrum: an unpipelined, shared-resource architecture.

In this chapter, conserving area will become the primary concern, either through a specified bound or via minimization. Yet, in the vein of true design-space exploration, we will provide a broad suite of options to the designer, providing techniques for both area-minimization and performance-maximization under a set of constraints. These constraints will be further expanded in Chapter 6 to incorporate energy and power. We initially target single-token (or single-threaded) implementations, but will move on to consider an alternate approach for multi-token scheduling in Chapter 5. The design flow for this chapter is shown in Figure 4.1.

The primary challenge faced in this chapter is common to most high-level synthesis approaches; we must determine how to allocate, bind, and schedule function units.

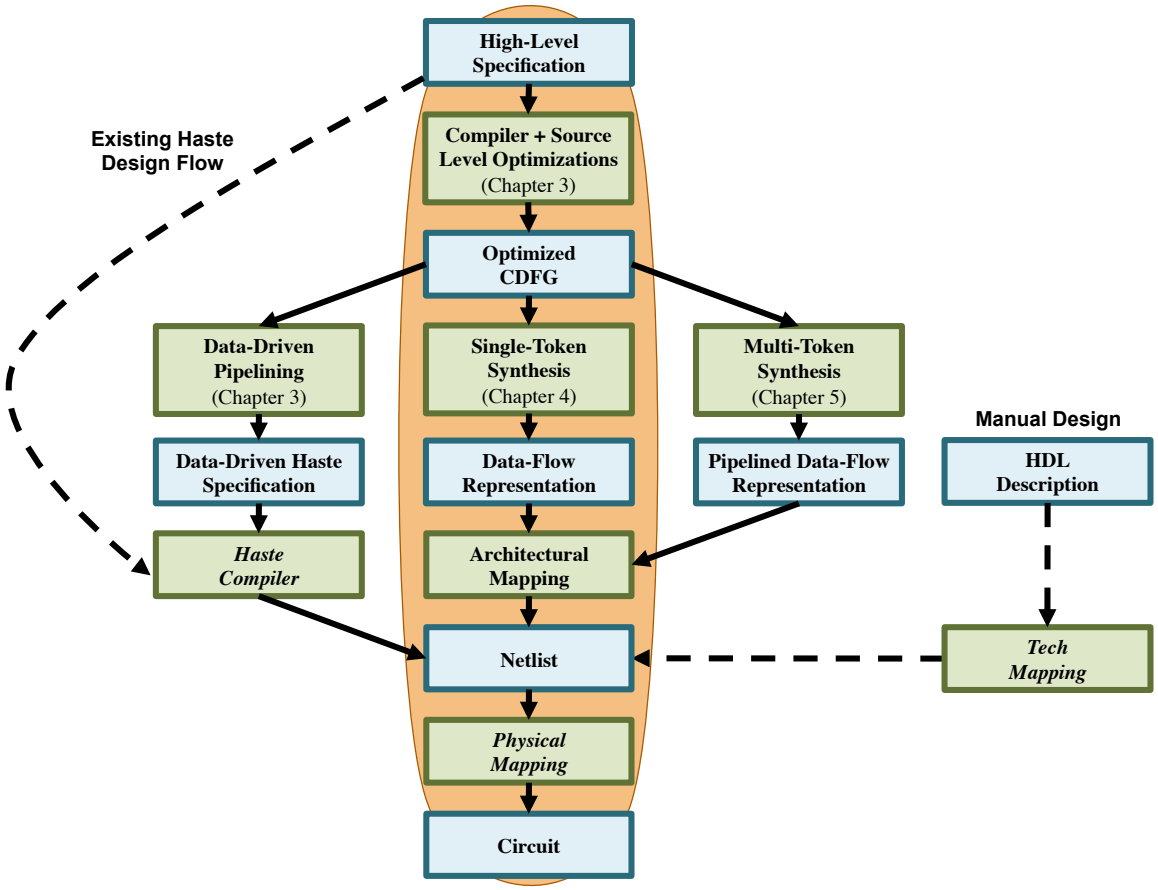

Figure 4.1: Single-token, shared-resource design flow

This problem, at its core, is intractable, particularly for large examples. However, I introduce efficient scheduling algorithms that, in practice, show excellent performance even for large examples.

4.1 Introduction

This chapter targets high-level synthesis of asynchronous systems, and introduces a fast exact approach to the scheduling and allocation problem for shared resources. Much of the recent work on this topic has been adapted from synchronous approaches, and therefore suffers from drawbacks associated with assuming discrete timing (or a discrete

approximation) for an asynchronous behavioral specification. In this chapter, I will present a fresh approach based on string permutations that does not require discrete timing, and is therefore capable of finding optimal schedules that other approaches may not be able to find.

Instead of the typical integer-linear-programming (ILP) based formulations used by prior approaches, my proposed method casts the problem directly as a string permutation problem. The string encodes events corresponding to starting and completion of operations, *i.e.*, a total ordering of start and end events. However, this simple representation directly encodes partial order among operations (due to dependency and/or resource constraints), and is powerful enough to encode all of the concurrency inherent in the specification. Since the string representation merely encodes an order, the associated search space is significantly smaller than that of an ILP because in the latter, each discrete time step, each operation, and each resource contributes to the dimensionality of the search space.

A significant contribution of the proposed approach is a set of powerful pruning strategies that drastically decreases the size of the search space, allowing for an efficient branch-and-bound solution. A key pruning strategy is to use heuristics to order the walk through the search space so as to find a quick, possibly non-optimal, solution, and use this solution to quickly bound the search space. In addition, several other techniques are provided to avoid redundant searches, and to quickly determine the infeasibility of certain solutions.

Several flavors of the high-level synthesis problem are addressed:

- time minimization under resource constraints,
- area minimization under time constraints,
- time minimization under area constraints, and

- multi-constrained scheduling.

The approach has been automated and applied to a set of examples. In most cases, an optimal solution was found in about 1 second or less (typically less than 50ms). Even for the largest example (1090 operations), all but one test case completed in under five seconds. For comparison, an ILP formulation based on previous work by Nielsen et al. (Nielsen, 2005; Nielsen et al., 2004; Nielsen et al., 2009) was also implemented, though a third-party ILP solver was used instead of simulated annealing or genetic algorithms used by Nielsen et al. Results show that my approach is faster by 1.9x-180x for small examples, and faster by *multiple orders of magnitude* on the larger examples.

The remainder of this chapter is as follows: in Section 4.2 I will describe previous work in the area of high-level synthesis and explain the drawbacks of using a time-step based ILP approach for the asynchronous scheduling problem. Then, in Section 4.3, I explain how the search space is formulated in my approach. Next, I introduce multiple search strategies for optimizing a specification in Section 4.4. In Section 4.5 I present a generalization to allow for a broader solution space by allowing *many-to-many* mappings of operations to function units. Finally, I present the results of experimentation in Section 4.6 and present conclusions in Section 4.7.

4.2 Background and Previous Work

Numerous approaches to high-level synthesis have been considered in the last several decades; a survey of synchronous techniques is available in (Micheli, 1994). The focus of this chapter is specifically on the problem of scheduling and allocation. Solutions to this problem fall into multiple categories; an approach may be exact or heuristic, target synchronous or asynchronous systems, and may be performed by an algorithm/heuristic or a guided random search (such as genetic algorithms/simulated annealing). In this

section I will discuss several techniques, including ILP-based approaches, the graph based approach of (Bachman et al., 1999), as well as other heuristic approaches.

4.2.1 ILP Approaches

ILP approaches are commonly discussed because they are capable of producing exact solutions, despite the intractability of the resource-constrained scheduling problem. The ILP approach fits naturally with a synchronous paradigm. Multiple formulations exist, one such (Nielsen, 2005) is described here:

1. Create a list of variables representing a range of possible scheduling times for each node in the data-flow graph (DFG). Exactly one variable in this list must have the value 1 (all others are 0), meaning the node begins execution at that time.

2. For each node, constraints are added such that the start time of each dependent node must occur after this node completes execution. These constraints are be added for each direct dependence.

3. Additional constraints must be added to the ILP such that no more resources are used than available. For ILP formulations in which the number of resources is not pre-allocated, a set of variables representing the number of functional units allocated for each type must be added.

4. An optimizing function is generally applied to complete the ILP. A function to minimize area would reduce a weighted sum of the variables in step 3. A latency minimization can be performed by adding a sink node and minimizing its start time.

While such a formulation can produce exact results in the synchronous realm due to a fixed time interval associated with the clock, it can miss optimal solutions when

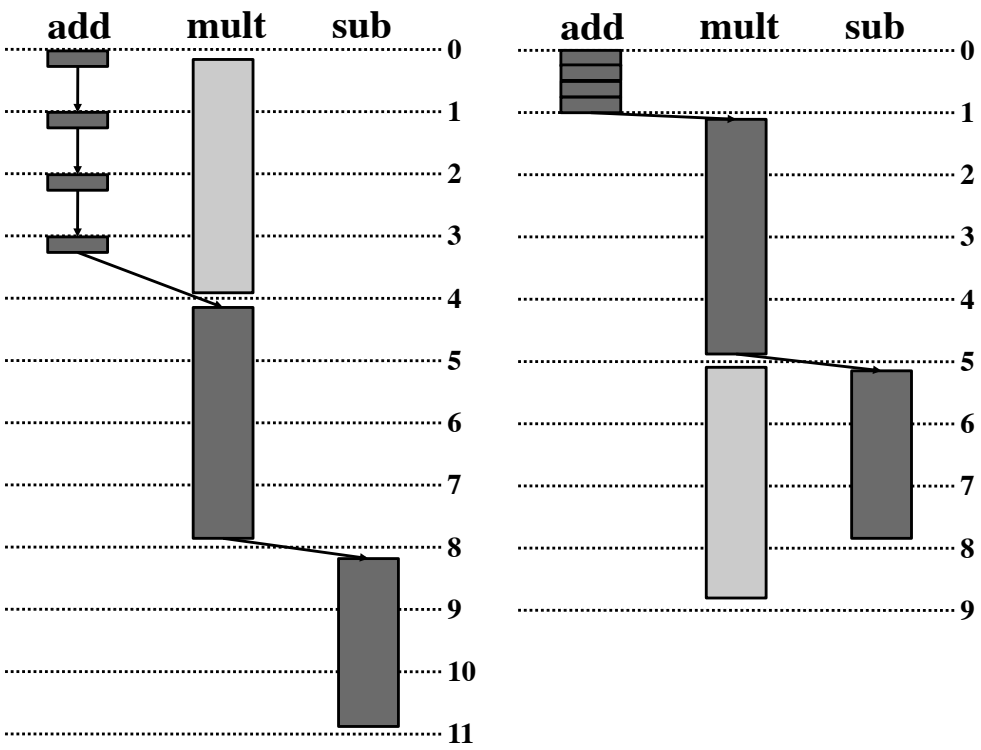

Figure 4.2: Illustration of differences between synchronous (left) and asynchronous (right) scheduling

applied to the asynchronous realm. This drawback is illustrated in the simple scenario in Figure 4.2. In this diagram, we consider a system with one functional unit each of three resource types: add, multiply, and subtract, with latencies 0.25, 4, and 3, respectively. Arrows indicate dependencies.

In Figure 4.2a, the optimal synchronous schedule is shown. Because the subsequent additions cannot be scheduled until the next full cycle, the multiplication on the critical path can start at earliest at time 4, yielding a best schedule of 11. In contrast, the adders in the asynchronous schedule do not need to be aligned on a clock boundary, and can produce the more optimal schedule shown in Figure 4.2b with a latency of 9.

To create a better schedule for the synchronous case, one could reduce the clock period to the size of the GCD of each operation (in this case, 0.25). However, this technique has significant drawbacks in terms of complexity and run-time of the ILP solver, particularly when the clock period becomes small in comparison to operation length. For this simple case, another technique, operator chaining (Wilson et al., 1995), can be performed by adding additional constraints to the ILP; but this technique is only applicable when the latency of a resource is half the clock period or less. Finally, one can consider pre-scheduling the four sequential adds as an atomic block within a single clock period; however, this approach may remove more optimal schedules from the search space.

In the asynchronous domain, the ILP approach has been adapted for use by (Nielsen, 2005) and (Saito et al., 2006). In work by Nielsen et al. (Nielsen, 2005; Nielsen et al., 2004; Nielsen et al., 2009), the ILP formulation discussed previously is used. Rather than focus on modification of the ILP to fit the asynchronous domain, their research describes heuristics to find a solution using guided search: simulated annealing and genetic algorithms. After scheduling is performed, the synchronous constraints can be relaxed, and may result in a lower latency asynchronous schedule. This work addi-

tionally describes techniques for power-constrained synthesis and other optimizations targeting reduction of power/energy use.

In (Saito et al., 2006; Saito et al., 2007), the ILP formulation is modified by performing approximation of start times. In essence, this approach can help limit the number of variables and constraints in a system by removing non-essential start times from a node's vector. This approach can feasibly reduce the run-time of a system and produce more optimal schedules by allowing a finer granularity clock to be considered, more effectively modeling an asynchronous approach. However, their approach does not guarantee optimality, as the number of approximated start times for each node must be limited for efficiency, particularly when the number of nodes in the DFG is large.

4.2.2 Graph-Based Approaches

The work most relevant to ours is described in (Bachman et al., 1999) and (Bachman, 1998), in which a graph-based approach is applied to produce an exact optimal schedule. Unlike an ILP approach, the complexity of the design space is independent of the discretization of time, and the approach accurately models an asynchronous paradigm. However, the approach is dismissed in later work by one of the authors (Saito et al., 2006) for having a high computational complexity.

In (Bachman et al., 1999), the input DFG is used as a baseline with data dependencies marked with a forward edge. The initial DFG corresponds to a schedule for a system with infinite resources and can be analyzed as such. The authors then present an algorithm to perform scheduling by adding resource edges between nodes. As each edge is added, a topological sort is performed on the graph, and the result can be reanalyzed to determine properties such as area and latency. Adding resource edges can at best reduce the area of the system, but may increase the latency of the schedule.

Like our approach, Bachman et al.'s method applies a branch-and-bound strategy to prune the search space, and applies multiple pruning techniques, such as removal of infeasible, redundant, and implied edges, and filtering by a minimal latency bound. Some of these filters have parallels to those in our approach.

4.2.3 Other Approaches and Heuristics

Several other unique approaches have been proposed, some of which target slightly different areas. The approach of (Tugsinavisut et al., 2006) proposes both an ILP and a list scheduling heuristic using Petri nets rather than DFGs as input. The heuristic presented in (Burns et al., 2004) presents an approach using "tight packing" and closeness tables for scheduling, particularly targeting low interconnect solutions.

Other heuristics have been developed with the goal of quickly producing quality solutions. Approaches such as force-directed scheduling (Paulin and Knight, 1987), ELS/ELAS (Badia and Cortadella, 1993), and list scheduling (Sllame and Drabek, 2002) have been proposed. While these approaches can be quite effective for producing results rapidly, these heuristics do not guarantee that the optimal solution will be found.

The SPARK (Gupta et al., 2003) framework is a well-known synthesis methodology that includes several high-level optimizations such as parallelization and loop transformation in the synthesis process. However, this framework primarily targets performance, and only considers one-to-one mappings of operations to resources.

4.3 Search Space Formulation

In this section I give a high-level overview of my proposed approach. I will start with a description of the input specification (a data-flow-graph) and describe a few properties and annotations that will be used in future sections. Next, I explain how the problem

is formulated as one of string permutation, describing validity restrictions for string acceptance. Finally, I describe how the search space is represented, explored, and pruned.

4.3.1 Preliminaries: Input Specification

4.3.1.1 DFG and CDFG representations

As input, my approach expects a DFG representing the behavior of the system. Each node in the DFG corresponds to an operation, such as addition or multiplication. Between nodes in the graph, directed edges are inserted that model the flow of data between operations, and therefore represent dependencies in the specification.

One aspect absent from the DFG representation is the control associated with the flow of data, *e.g.*, loops and conditionals, that are included in a control-data-flow-graph (CDFG) representation. We can apply the approach in a straightforward fashion to these constructs: loops can be targeted by scheduling the loop body and repeating this schedule at runtime until the loop terminates. Conditionals can be targeted by conversion to conditional assignments. However, in most cases an optimal static schedule for these problems cannot easily be found due to their dynamic runtime behavior. Yet, these constructs will be explored in the multi-token solution presented in Chapter 5.

For ease of analysis, two additional nodes are added to the DFG: a root node and a sink node. For all the nodes that do not have a dependence, a dependence is added on the root node. For all the nodes without any dependent nodes, the sink node becomes a dependent node. This allows for a single start node and finish node when annotating DFG nodes.

4.3.1.2 DFG node properties and annotations

One of the first steps in the approach is to annotate each DFG node with a few essential attributes, allowing us to calculate additional properties using the DFG structure. We assume the initial properties associated with each node include an operation type, a list of dependent nodes, and a list of nodes for which this node has a dependence. We will denote dependence by the following: $i \leftarrow j$ indicates that i is dependent on j.

First, each node in the DFG is assigned a unique string identifier to be used in scheduling. Each node is also linked with the appropriate "resource class" that contains properties of its functional unit, *i.e.*, area and latency. From this resource class, we can associate an execution time, $EXE(node)$, with each node in the DFG. This approach initially assumes an assignment of one resource type to each operation, a restriction that will be relaxed in Section 4.5.

With this information, we can calculate two properties: (i) $STTS$: the shortest time to start, and (ii) $STTF$: the shortest time to finish. The $STTS$ indicates the earliest that a node can start in the system given infinite resources. This computation is performed by analyzing the nodes upon which a node is dependent. The $STTF$ indicates the earliest that this node and all its children can complete execution, given infinite resources, and is determined by analyzing a node's dependencies. The formulation for these properties are as follows:

$$STTS(N_i) = \max_{j|i \leftarrow j}(STTS(N_j) + EXE(N_j)) \tag{4.1}$$

$$STTF(N_i) = \max_{j|j \leftarrow i}(STTF(N_j)) + EXE(N_i) \tag{4.2}$$

Note that the execution time of the root and sink nodes is zero. These properties are computed prior to scheduling. An example computation of these properties is shown in Figure 4.3.

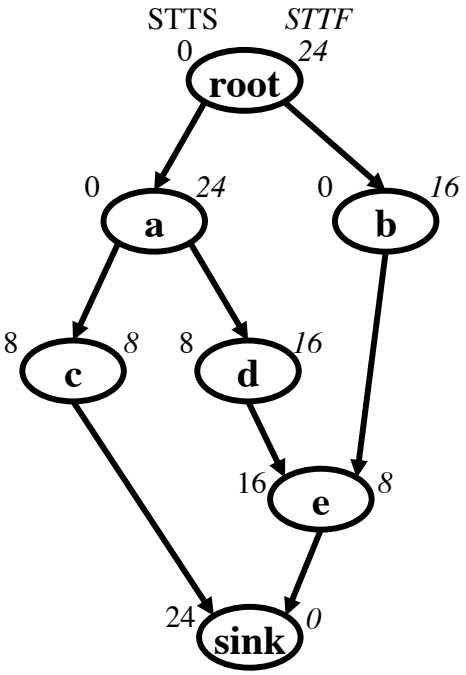

Figure 4.3: DFG example annotated with $STTS$ and $STTF$ properties (each operation executes for 8 time units)

4.3.2 Scheduling as a String Permutation Problem

The key idea of the proposed approach is to model the scheduling problem as a string permutation problem. In this approach, each event in the system is associated with a specific string, and multiple event strings are concatenated into one large scheduling string. Several restrictions must be placed on the formulation of these strings to ensure a valid schedule is generated.

4.3.2.1 Terminology

As noted in the previous section, each node is assigned a unique string identifier. For each node in the DFG, two unique event strings most be present in the scheduling string: *node*+ and *node*−. The node-start string, *node*+, corresponds to operation *node* commencing execution, while the node-finish string, *node*−, corresponds to its completion.

4.3.2.2 Basic restrictions on valid strings

Several basic restrictions must be placed on the set of valid scheduling strings:

- every unique event string must appear exactly once,
- for a specific *node*, *node*+ must always precede *node*−,
- $j-$ must always precede $i+$, for every $i \leftarrow j$.

From the first restriction, it follows that the length of the scheduling string must be equal to twice the number of nodes in the DFG.

4.3.2.3 String interpretation and temporal restrictions

Since strings do not include timing information, any valid string can be interpreted as an infinite number of schedules. Therefore, let us interpret the scheduling string as the most tightly packed schedule possible in an ASAP (as-soon-as-possible) fashion, for which time is monotonically increasing as the string is read from left to right. The primary constraint required in this interpretation is that each *node*− must complete $EXE(node)$ time units after its *node*+ commences. A secondary constraint is to require consecutive node-start strings to have zero time occurring between them.

For example, assume a and b are additions with a latency of 10 units, and the schedule produced is $a + b + a - b-$. In this approach, the interpretation is as follows (times associated with each event are given in parentheses): $a + (0)\ b + (0)\ a - (10)\ b - (10)$. Without the secondary constraint, other valid interpretations could exist, one such is: $a + (0)\ b + (5)\ a - (10)\ b - (15)$.

This choice of interpretation has been selected for two reasons: (i) it most closely matches an asynchronous data-driven paradigm, and (ii) this interpretation provides a schedule that is in the set of optimal schedules for a specified string. While other

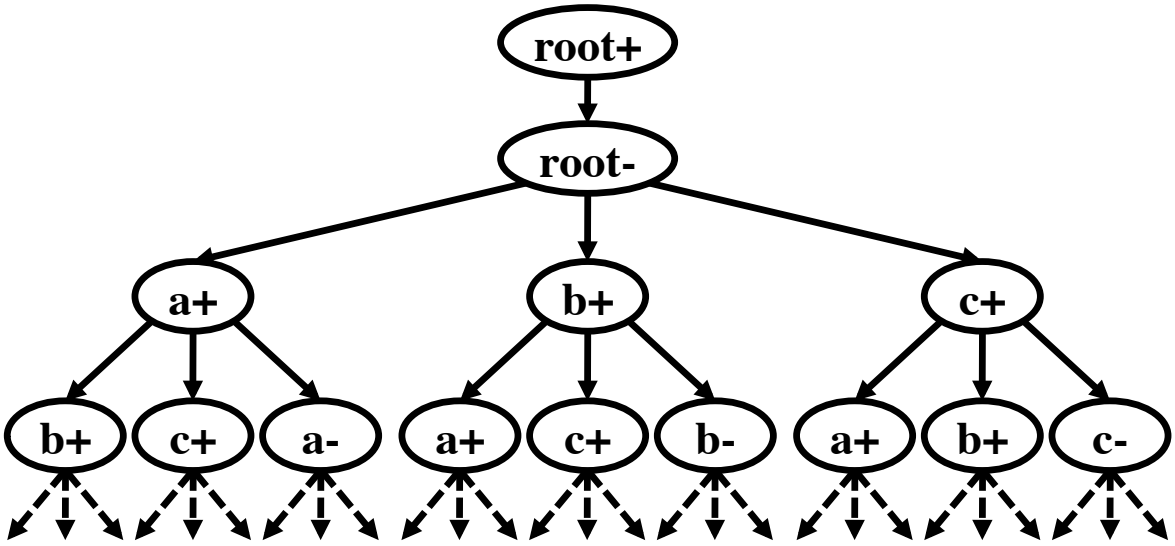

Figure 4.4: Partial expansion of search space for a three-statement DAG

schedule interpretations may be valid, any monotonically increasing interpretation can have a latency no better than that of the interpretation we have chosen.

Due to the monotonically increasing time requirement of our interpretation, only a subset of valid *node*− strings can be considered for selection at any point in the schedule. As an example, consider the case where a is an addition with a latency of 10 time units, and b is a multiplication with a latency 20 time units. The string $a+\ b+\ b-\ a-$ is not valid, as time is not monotonically increasing: $a + (0)\ b + (0)\ b - (20)\ a - (10)$. However, were both a and b additions of latency 10, the following schedule would be valid: $a + (0)\ b + (0)\ b - (10)\ a - (10)$.

4.3.3 Representing and Exploring the Search Space

In order to traverse the search space of valid scheduling strings, the scheduling problem is mapped onto a directed acyclic graph (DAG). In this subsection, I will describe how the DAG is generated and interpreted, how events are represented as nodes in the DAG, how children are selected for each node, and finally, how the search space is pruned via branch-and-bound and redundancy removal in order to significantly reduce runtime.

4.3.3.1 Building and interpreting the DAG

To start building the DAG, let us begin by inserting the node-start event of the root node, *root*+ as the first event in the system, as shown in Figure 4.4. From there, we can generate a list of possible other events that can be started in the system; these events become child nodes of *root*+. In order to be considered for selection as a child node, the requirements on string ordering in 4.3.2 must be met, including both dependence restrictions (for *node*+ nodes) and temporal restrictions (for *node*− nodes). Due to these restrictions, *root*− is the only node that can follow *root*+, as all other nodes have a dependence on *root* or another node.

After *root*− completes, we can again continue this procedure of selecting valid events that can be triggered at this time; in this case, this set includes any events that have no dependencies in the original DFG. These children are enumerated, and each can then be explored in the same fashion, until *sink*− is reached (a node with no possible children). If one were to naïvely expand every node in the graph, this would essentially generate a list of all possible schedules for this DFG. However, the approach *does not* perform a full enumeration in practice, as it would be an incredibly time-consuming process. Instead, the search space is pruned by applying several different bounds and optimizations; these will be described in later sections.

In the generated graph, the path from the root node to the current node gives the partial schedule of events occurring in the system up until that point. Each edge in the graph can be considered to have a weight corresponding to the time elapsed between two events, according to the interpretation described in Section 4.3.2. Any full path from *root*+ to *sink*− represents a complete and valid schedule.

4.3.3.2 Node properties

In addition to the unique string identifier associated with a node in the DAG, several other properties are associated with each node in order to reduce redundant computation when traversing the graph:

- a link to the corresponding annotated node in the DFG,
- the start or end time associated with this node,
- a list of computations that have been started and finished by the time this node is reached,
- the start times associated with computations that have started but not yet finished, and
- the number of free resources in the allocation at this node.

Each of these items could be generated simply by inspecting the path from the root to the current node. For performance reasons, however, we can pass as much of this information as possible from one node to the next, rather than recomputing it based on the full path each time. This choice is a storage/time trade-off that effectively reduces the run-time of the implementation.

The final item in the list above is determined at each node by counting the number of $node+$ nodes in the path for which a $node-$ node has not been encountered, for each resource type. This information is used to help prune the search space in resource-constrained scheduling.

4.3.3.3 Pruning the search space and branch-and-bound

As exhausting the full search space of the DAG would be highly inefficient, a key contribution of this work is the development of several methods for pruning the search

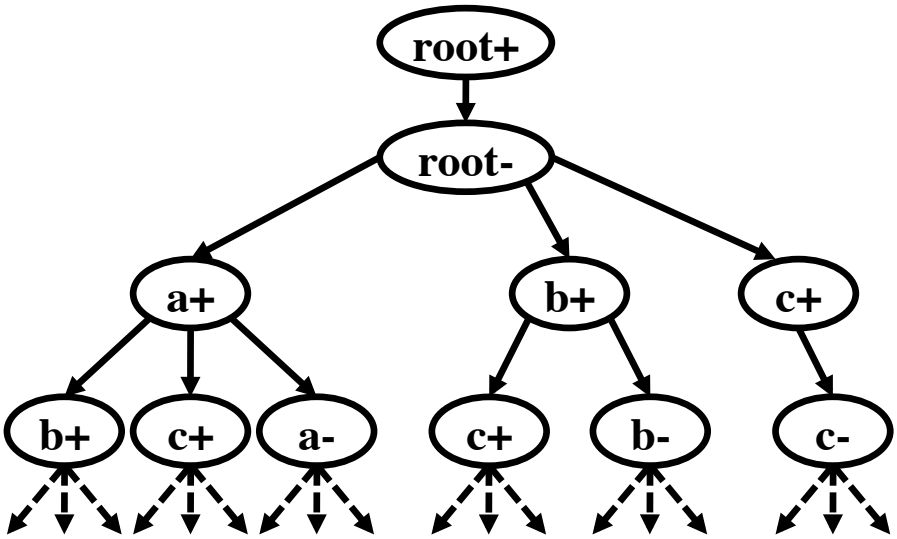

Figure 4.5: Pruning via lexicographical ordering

space. The cornerstone of our approach is a branch-and-bound technique to cut portions of the search space (DAG) entirely when constraints are not met, specifically time and resource usage.

There are several key components to effectively pruning the search space in the proposed approach:

1. developing several safe but tight bounds,
2. rapidly generating a quality schedule, and
3. removing redundancy in the search space.

Items 1 and 2 will be discussed in depth in Section 4.4 in relation to time bounds; item 3 is discussed below.

4.3.3.4 Redundancy in the search space

Due to the timing interpretation we selected, multiple redundancies in scheduling strings are removed by adding further restrictions on the validity of a string. Conceptually, the timing interpretation does not place any restrictions on the ordering of

events occurring within the same time frame. For example, the scheduling substrings $a + b + c+$ and $b + a + c+$ are both valid and will represent two different paths in the search space.

By enforcing a lexicographical ordering on node-start strings, we can reduce the overall search space without losing optimal solutions. For example, $c+b+a+$ no longer becomes a valid scheduling substring, but the same schedule will exist elsewhere in the graph as $a + b + c+$. These two strings have the same meaning: start operations a, b, and c; but this meaning can be represented in multiple redundant ways. By enforcing an ordering, we prune out that redundancy, as shown in Figure 4.5.

A similar ordering is enforced on node-finish strings. In the case where consecutive node-finish events occur at the same time in the schedule, multiple orderings can exist, as in the earlier example of $a + b + a - b-$ and $a + b + b - a-$. Again, we enforce an ordering on these node-finish events to remove redundancy. Note that this ordering is *not* enforced when time elapses between two node-finish events. As a result of the restrictions on node-finish ordering, at most a single *node*$-$ node may be selected as a child for any node.

We add one final constraint on the ordering of node-finish strings in relation to node-start strings: if both a *node*$+$ and *node*$-$ event occur within the same time instant, the *node*$-$ node must precede the *node*$+$ node.

The combination of these three restrictions on string ordering results in a single possible order for concurrent event firings: all *node*$-$ events are listed in lexicographical order, followed by all *node*$+$ events listed in lexicographical order. These ordering restrictions can have a very significant impact on performance, as shown in Section 4.6.

4.4 Search Strategies

Given the problem formulation just described, we can now move on to explore multiple synthesis techniques, each targeting different objective functions and restricted by a set of constraints. First, I will describe the core method, resource-constrained time-minimization, along with several optimizations to reduce the search space. Then, I will continue by describing time-constrained area-minimization, area-constrained time-minimization, and multi-constrained search, all of which are meta-level optimization algorithms for which resource-constrained time-minimization is an underlying step.

4.4.1 Resource-Constrained Time-Minimization

4.4.1.1 Objective Function and Constraints

In resource-constrained time-minimization, the goal is to produce the lowest-latency schedule for a DFG given a set of resource constraints. For this method, we assume allocation has already been performed, leaving us with a maximum number of function units that can be used concurrently when generating a schedule.

4.4.1.2 Basic Algorithm

The basic algorithm is shown in Figure 4.6. From the DFG, a worst-case maximum time can be computed by summing the latencies of every operation; this worst-case scenario corresponds to a completely serialized schedule, *i.e.*, one with no concurrent operations. This will serve as an upper bound for schedule latency, and the algorithm will replace this value as it finds better solutions.

The next step is to enter into a recursive procedure, *expand*(), that performs a depth-first search on the search space. When a sink node is reached, further expansion on this path terminates. If the start time associated with the sink node is less than the

```
int RCTM(DFG dfg, Allocation alloc){
   minTime = getWorstCaseTime(dfg)+1;
   expand(rootNode, alloc);
   return minTime;
}

void expand(Node node){
   if (isSinkNode(node)){
      if (node.startTime<minTime)
         minTime = node.startTime;
      return;
   }
   NodeList children = getChildren(node, alloc);
   for each child in children
      if (child.startTime>=minTime)
         return;
   for each child in children
      expand(child);
}
```

Figure 4.6: Basic algorithm for resource-constrained time-minimization

current minimum time, the new best time is logged, as well as the schedule (not shown in code).

One of the core procedures in the basic algorithm is the *getChildren*() procedure, shown in Figure 4.7. This method enforces the restrictions on strings given in Section 4.3 and lists only the nodes that can be executed by considering dependencies, availability of resources, and lexicographical ordering. As this method generates the child nodes, it also must set properties such as timing information and resource use on this path; this is represented as the *updateProperties*() procedure.

The only bound to prune the search space in the basic algorithm is a very loose bound: if a node is reached with start time at or exceeding the current best time, exploration further down this path is terminated.

```
NodeList getChildren(Node parent,Allocation alloc){
   NodeList list = new NodeList();
   for each child in unscheduled start-nodes:
      updateProperties(child,parent);
      if (dependenciesResolved(child)&&
         resourceAvailable(child, alloc)&&
         lexicographicallyOrdered(parent,child))
            list.add(child);

   Node finishNode;
   for each child in executing nodes:
      if (timeLeft(finishNode) > timeLeft(child))
          finishNode = child;
   if (timeLeft(finishNode)==0)
      return new NodeList(finishNode);
   if (finishNode!=null)
      list.add(finishNode);

   return list;
}
```

Figure 4.7: Algorithm for selecting child nodes in the DAG

4.4.1.3 Optimizations

In order to more effectively prune the search space, several optimizations are performed, as described below. These optimizations are applied to the basic algorithm given in Figure 4.6.

Sorting. The first optimization is a sorting of the child nodes selected by the $getChildren()$ method, according to their $STTF$. In the original algorithm, the order of selection of children is based on an arbitrary lexicographical order. However, this ordering does not consider any specific property of each node.

By assigning ascending lexicographical identifiers to each node in the order of decreasing $STTF$, we can ensure that the paths with the greatest $STTF$ are explored first for execution. This choice will allow the critical path in the DFG to be considered for scheduling first, leading a good quality initial solution for more effective pruning in later steps. As you may recall, in a step prior to exploring the search space, each node in the DFG was annotated with its STTF. Thus, this sorting does not incur any extra performance penalty.

Dominance check using hashing. There are several partial schedules generated by the graph for which the same set of operations have been scheduled to start and finish, albeit in different orders. So, as each node is generated, a hashing string is also created that corresponds to an unordered set of events that have occurred in the partial schedule. A global hash is then accessed to locate an array of other nodes that have also performed the same set of events.

The properties of the current node, particularly latency, can then be compared to that of other nodes with the same set of events. If this node is inferior to any of the nodes in that hash location, further exploration of this path is no longer considered. In order to be considered inferior, two conditions must hold:

1. the node must have a latency greater than or equal to the compared node, and

2. all active computations in a node must have end times greater than or equal to those in the compared node.

In most other cases, the two nodes cannot be accurately compared to determine superiority. In the case that the current node is found to be superior to a node in the array, the compared node is evicted from the array. If after comparison to all nodes in the array, this node cannot be found to be inferior, it is inserted into the array at this hash location.

Tightening time bounds. In the original algorithm, pruning was performed only when a node in the graph met or exceeded the current best time. To improve pruning, we can consider two tighter bounds on time: $STTF$ and $RCSTTF$.

The $STTF$ bound is implemented rather simply. The best possible finish time for a node and its children, given infinite resources, is equal to the current time in the partial schedule plus the $STTF$ for this node. Therefore, rather than checking that the current time is less than the current best time, we can instead check to see that the current time plus the $STTF$ for this node falls below the best time bound.

Using the same logic, a more complicated bound is generated, the resource constrained shortest-time-to-finish ($RCSTTF$). The aim of this bound is to consider the start-time overhead of scheduling the unscheduled nodes in remaining on a path while considering resource constraints.

For this bound, a list of *node+* events that have yet to execute is accumulated, sorted in descending order of $STTF$, for each resource. As a reminder, note that this sorting is performed *prior* to exploration of the DAG. We then compute a minimum time bound for this set of operations to finish in the best case. This is performed by adding together the current time, the $STTF$ for each node, and an additional wait-time delay for this operation to gain access to a resource. This sum is then compared to the best time, and if the result is greater than or equal to the best time, the current node

Table 4.1: Sample $RCSTTF$ bound for two adders

Node	Current Time	STTF	Wait Term	Estimated Finish
$q+$	50	22	0	72
$r+$	50	16	0	66
$s+$	50	16	8	74
$t+$	50	14	8	72
$u+$	50	10	16	**76**
$v+$	50	8	16	74

is dropped from the search space.

A sample table showing the intuition behind the $RCSTTF$ bound is shown in Table 4.1. In this scenario, we have six operations that have not yet been started at the current time of 50. These operations are sorted in descending order by $STTF$.

Two adders have been allocated in the system, each with an execution time of 8 time units. As a result, no more than two operations can have access to the adder resources at a time, and the other adds are forced to wait. If we assume the first two operations can execute immediately, the next two operations can commence at earliest at time 8, the next two at time 16, and so on. These wait delays are represented in the table as the "Wait Term."

In the end, the objective is to find the ordering of statements that will minimize the maximum value of the elements in the "Estimated Finish" column. The descending $STTF$ ordering as shown will give an earliest estimated finish of 76, which will be our RCSTTF bound for this node. Any other ordering than the one shown here will produce at least one element in the estimated finish column that meets or exceeds the calculated bound of 76.

Note that the $RCSTTF$ bound does *not* perform any scheduling; the sorted schedule is used only to produce a tighter bound for pruning. As the $STTF$ term on which $RCSTTF$ relies considers infinite resources, it may be that an alternate ordering will be what produces an optimal schedule. However, this alternate schedule will *not* be

discarded by the $RCSTTF$ bound.

4.4.2 Area-Constrained Time-Minimization

Now let consider a related, but more general problem, that of area-constrained time-minimization.

4.4.2.1 Constraints and targets

In area-constrained time-minimization, the goal is to produce the lowest latency schedule for a DFG given an area constraint. This method differs from resource-constrained area-minimization in that allocation has *not* been performed, so multiple allocations become part of the search space. Hence, our approach is a meta-level search for which the resource-constrained time-minimization method is run for each allocation.

4.4.2.2 Enumerating the search space

For area-constrained time-minimization, multiple unique allocations from the allocation search space must be enumerated and analyzed in order to determine a minimum value. Conceptually, one could build this space by generating a list of all possible allocations, removing any allocations with greater area than the constraint, and then removing any allocation in the list that is subsumed by another allocation.

If the search space is small enough, this list of possible allocations can be performed up-front for analysis. As the number of combinations increases, a more dynamic method of enumerating this space must be performed to reduce run-time and prevent unnecessary combinations from being considered.

Although the full search space under an area bound consists of a multi-dimensional "volume" of allocations (bounded by integer constraints), an allocation on the surface of this volume is guaranteed to provide the best possible solution. Because allocations

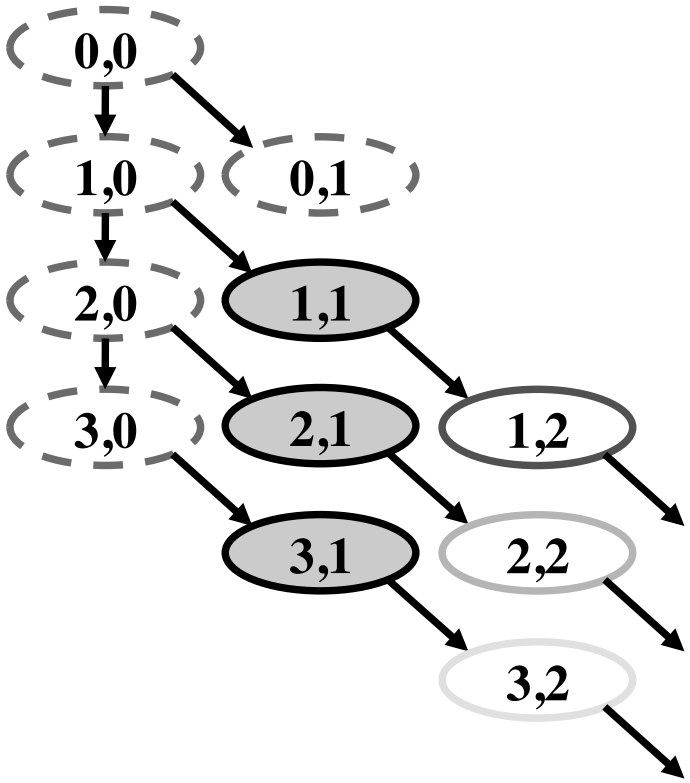

Figure 4.8: Allocation search space for two function unit types

below the surface will contain fewer function units than an allocation on the surface, they can at best match a surface allocation in terms of performance. Since we are only concerned with time-minimization, we can therefore ignore the subsumed allocations that exist below the surface.

The allocation surface can be generated in a variety of ways. In the proposed approach, a tree of allocations is created, starting with an empty allocation node as shown in Figure 4.8. The tree is then expanded at the root by creating a set of new nodes under a set of restrictions:

1. each child node can add only one additional allocated function unit,

2. each child node must be distinct from its siblings,

3. the generated node's allocation must not exceed the area bound,

```
void ACTM(DFG dfg, int areaBound){
   AllocationList allocations =
            getPossibleACTMAllocations(areaBound);
   int bestTime=getWorstCaseTime(dfg)+1;
   for (int x=0; x<allocations.length; x++)
      bestTime = min(RCTM(dfg,allocations[x]),
                     bestTime);
}
```

Figure 4.9: Algorithm for area-constrained time-minimization

4. the selection of children may not break a lexicographical ordering.

A detailed description of the final restriction is the following: each function unit type is assigned a string (*e.g.*, *mult*1), and the full set of function unit strings are given a lexicographical ordering. Each time a function unit is added to an allocation, that function unit's string is appended to the path's string. Paths generated in the allocation tree are legal as long as the path string is in lexicographical order. This restriction prevents redundancy in the search space, as illustrated in Figure 4.8: only one path exists to each node.

A simple example of the lexicographical restriction is the following: if a node's grandparent adds an ALU to the allocation, then the current node adds a multiplier to the allocation, its children are barred from adding ALUs to the allocation.

When a node can no longer be expanded and it meets the validity requirements (*i.e.*, each operation in the DFG can be mapped to a function unit), it is considered for selection in the final allocation pool. When fully enumerating the search space under an area constraint, the final step is to prune allocations that are subsumed by others; this step is performed by removing any leaf node for which a function unit can be added when ignoring lexicographical order.

4.4.2.3 Algorithm

A basic algorithm for area-constrained time-minimization is shown in Figure 4.9.

First, a list of all possible allocations is generated. A best time bound is set as in the resource-constrained time-minimization scenario. Then the resource-constrained time-minimization algorithm is run for each allocation, however, each subsequent call to $RCTM$ is bounded by the best time determined so far, rather than a worst case time for the DFG.

4.4.3 Time-Constrained Area-Minimization

4.4.3.1 Constraints and targets

In time-constrained area-minimization, the goal is to produce the lowest-area schedule for a DFG given a time constraint. For this method, allocation has *not* been performed, so each unique allocation becomes part of the search space.

4.4.3.2 Enumerating the search space

The search space for area-minimization now consists of the full multi-dimensional volume of possible allocations. However, the full search space need not always be considered; if a low-area solution is found, any higher-area allocations are discarded.

Unlike time-minimization where the full allocation search space must be expanded, the search space can be enumerated dynamically in area-minimization. A similar tree allocation approach to that of generalized time-minimization is performed, with the exception that nodes are not expanded indefinitely. Starting with the initial empty allocation, child nodes are expanded in each path in a depth first fashion, as in Figure 4.8. When a legal allocation is found, the path is prevented from further expansion temporarily, and added to a list of "leaf" nodes. The initial leaf nodes are shaded gray in the figure.

When this initial expansion pass is complete, a list of low-area legal allocations is now recorded. The list of allocations is then sorted from least to greatest area. The

```
void TCAM(DFG dfg, int timeBound){

   AllocationList allocs = new AllocationList();
   do{
      Allocation alloc = nextSortedAllocation(dfg);
      allocs.add(alloc);
   }while(RCTMHeuristic(alloc)>timeBound);

   int bestArea = getArea(allocs.lastElement());

   while (allocs.size()>0){
     Allocation last= allocs.lastElement();
      if(RCTM(dfg, last)<=timeBound){
         bestArea = getArea(last);
         removeAllocsWithGTEArea(allocs, last);
      }else
         removeAllocsWithLessFU(allocs, last);
   }
}
```

Figure 4.10: Algorithm for time-constrained area-minimization

lowest area allocation is then considered heuristically to see if it meets the constraints. If so (for the initial case) the lowest area allocation is found. If not, it is added to a list of heuristically failed allocations for later consideration. The node is then expanded and its legal children are added to the list of low-area allocations. This list is re-sorted and then the next lowest area allocation is considered.

When a heuristic solution is successful, any failed allocations have to be considered using an exact resource-constrained time-minimization approach. If successful, a new minimization result is found. If the allocation fails, then it is removed from the list of possible allocations, as well as any of the allocations it dominates. This process continues until the list of failed allocations is exhausted and the best area solution is found.

4.4.3.3 Algorithm

The algorithm for time-constrained area-minimization is shown in Figure 4.10. At

the top level, the goal is to explore the search space in ascending order of area, trying to find the first allocation that will meet the bound via a heuristic. Then, the nodes that failed the heuristic search are considered in an exact fashion to ensure that an optimal solution was not missed.

The first step in the algorithm is to generate an empty list of allocations that will expand as we explore the search space. The initial loop produces the next unconsidered allocation in the search space with the lowest area. This allocation is added to the list, and a heuristic version of the resource-constrained time-minimization method is then run on the allocation. This heuristic method consists of finding the latency of the first path to a sink node in the graph.

If this allocation does not meet the time bound heuristically, the next lowest area allocation is considered. If it does meet the time bound, the loop is exited, and execution begins in the next loop, which considers lower area allocations in an exact fashion. If an allocation does not meet the time bound, any allocations with fewer function units are removed from the list. If an allocation does meet the time bound, we have found a new allocation with a best area bound. Any allocation with greater or equal area is then removed from the list. This process repeats until the list of allocations is empty.

4.4.4 Multi-Constrained Search

One final scheduling problem is a multi-constrained search. For multi-constrained search with minimization functions, additional constraints can easily be added to the methods above. For example, one may perform time-constrained area-minimization under an area-constraint; this type of search results in a reduced search space.

For methods with no minimization functions, the objective function is to find any valid schedule that meets the constraints. As an example, for resource-constrained and time-constrained search, the method involves adding a time bound to the resource-

constrained time-minimization approach, and returning the first valid schedule. For area-constrained and time-constrained search, the area-constrained time-minimization approach is used, returning the first valid schedule meeting the time constraint.

4.4.5 Binding

The final step of the synthesis process in our approach is performing binding. A relatively simple binding technique is used in the proposed approach. Moving in ascending order of time in the schedule, a *node+* is assigned to the first available function unit in a round-robin fashion. As the binding step is performed in a post-processing fashion, the effects of binding (*e.g.*, number of multiplexers) on the overall time/area consumption of the scheduling string are not considered in the scheduling process. In other words, we assume the multiplexing time and area to be a fixed value for each function unit.

4.5 Generalized Mapping Extension

The basic approach described in Sections 4.3 and 4.4 is limited to a *many-to-one mapping* of operations to function units. Using this mapping, a function unit may accept multiple classes of operations, but each operation class can be bound to a single function unit type. As an example, an ALU may accept both multiply and add operations, but if so, multiply and add operations can *only* be scheduled on an ALU, and no other function unit type. As a result of this limitation, no choice exists when binding an operation to a resource.

Therefore, a more general strategy for binding is needed. In this section, I describe an extension to the proposed scheduling approach to allow for a *many-to-many* mapping of operations to function units. In practice, this extension allows for a wider variety of legal schedules and allocations, and therefore may produce better solutions.

4.5.1 Modified annotations

In Section 4.3, each node is immediately bound to a specific resource class containing function unit properties such as area and latency. This binding allows for one-time analysis of some computed DFG annotations but prevents an operation from executing on alternative function units, *e.g.*, an addition operating on either an ALU or a dedicated adder.

Instead, we would prefer to bind each DFG node simply to an operation class, such as addition or multiplication. The actual binding of node to resource (*i.e.*, function unit) is performed during the scheduling process. As a result, certain DFG annotations must be parametrized by allocation and recomputed multiple times during execution of the solver.

The main computed annotation used in the original approach is the $STTF$, or shortest time to finish. This property indicates the earliest that a node and its children can complete execution in the presence of infinite resources.

The computation of $STTF$ in Section 4.3 is as follows:

$$STTF(N_i) = \max_{j|N_i \rightarrow N_j} (STTF(N_j)) + EX(N_i) \tag{4.3}$$

where $N_i \rightarrow N_j$ indicates that N_j is dependent on N_i and $EX(N)$ represents the execution time of the function unit this node is bound to. However, as the modified approach does not initially bind a node to a function unit, the equation must be changed:

$$STTF(N_i) = \max_{j|N_i \rightarrow N_j} (STTF(N_j)) + \min_{k|R_k \Leftarrow N_i} (EX(R_k, N_i)) \tag{4.4}$$

where $EX(R, N)$ represents the execution time of a function unit R for N's operation, and $R_k \Leftarrow N_i$ indicates node N_i can operate on function unit R_k. This formulation is subject to the restriction that a resource R_k can only be considered if its allocation

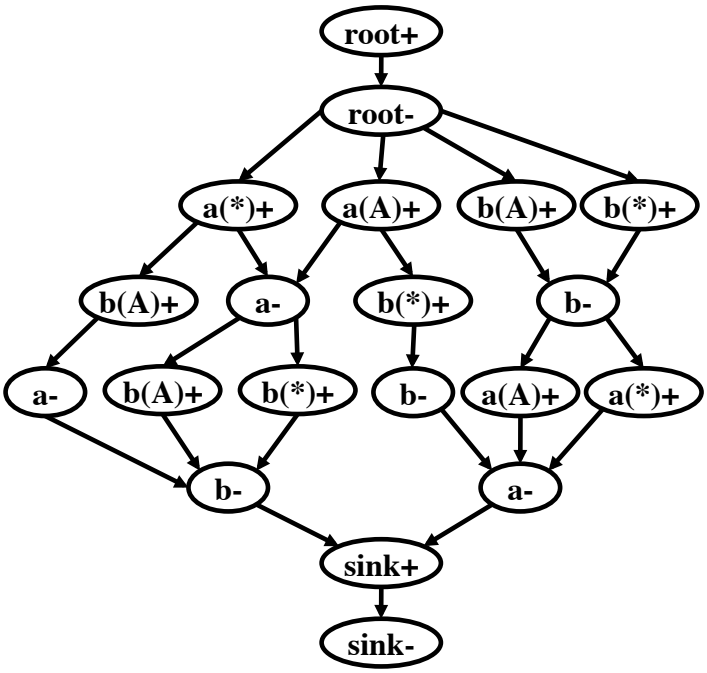

Figure 4.11: Full expansion of the DAG for a two-operation DFG with one ALU and one multiplier

count is greater than zero, hence the requirement that the DFG must be re-annotated with each new allocation.

This equation highlights several benefits of the modified approach: (*i*) an operation can be scheduled on multiple different types of function units, (*ii*) a function unit may accept multiple different types of operations, and (*iii*) a function unit may have parametrized latencies for each operation type it accepts.

4.5.2 Expanding the search space

The modified approach adds complexity to the search space by allowing multiple *node+* nodes corresponding to the same operation to exist as children of a single node. For example, $a(A)+$, which represents node a operating on an ALU, and $a(*)+$, which represents node a operating on a multiplier, can both be children of the same node, as shown in Figure 4.11. The selection of possible children nodes is allocation-dependent,

as some allocations may lack certain function unit types.

4.5.3 Modified time bound

One of the keys to the branch-and-bound approach is the selection of safe but effective bounds to reduce the overall search space. In Section 4.4, the resource-constrained shortest-time-to-finish bound ($RCSTTF$) is used to effectively prune the search space. Conceptually, this bound solves a simpler, less-constrained scheduling problem: find the best possible schedule in the absence of dependencies.

The bound is calculated by generating a list of nodes that have yet to start for a specific operation type and sorting them by their $STTF$ value. At this point, the maximum $STTF$ node can be summed with the latency of the current node, giving a loose bound on the earliest this path could finish. An additional RC term is added that takes into consideration the time that a node can be expected to wait to gain access to the resource it is bound to under the current allocation. The addition of this term forms an even tighter minimum bound on latency.

Because the original bound relied on having a many-to-one mapping of operations to resources, we must now modify the ($RCSTTF$) bound to incorporate the possibility of a many-to-many mapping. Since a node could be executed on a function unit that accepts multiple operation types, the best option is a conservative approach. In order to ensure that the bound is safe, when analyzing wait terms for a specific operation class (*e.g.*, addition) the wait terms associated with multi-operation function units (*e.g.*, ALUs) operate under the assumption that this operation class has exclusive access to the function unit. A sample calculation for this bound for a single class of operation is shown in Table 4.2, in which an add operation takes 6 time units on a dedicated adder and 10 time units on an ALU.

Table 4.2: Modified $RCSTTF$ bound for one adder (6 unit latency) and one ALU (10 unit latency)

Node	Current Time	STTF	Wait Term	Estimated Finish
$q+$	50	22	0	72
$r+$	50	16	0	66
$s+$	50	16	6	72
$t+$	50	14	10	**74**
$u+$	50	10	12	72
$v+$	50	8	18	**74**

4.5.3.1 Additional optimizations and considerations

When performing time-minimization in Section 4.4, each allocation generated is run through the resource-constrained time-minimization method and the best solution is maintained at each step to aid in rapidly pruning the search space. Because the number of legal allocations may increase significantly when multiple resource types are available, we can now extend this algorithm by performing two additional optimizations: (i) a heuristic pass is first performed for each allocation to provide a good initial time bound, and (ii) the allocation that provided the best heuristic solution is explored first.

The rationale behind these optimizations is that several allocations have a minimum latency which is much greater than true minimum provided by another allocation. Minimizing latency with respect to these allocations is generally a fruitless and time-consuming endeavor. Therefore, a heuristic pass can help order these allocations such that the best allocations are considered first.

One final optimization is to sort children of a node by considering the latency of the bound function unit type in addition to $STTF$. In this way, the algorithm will prioritize execution on lower-latency function units over higher-latency function units, if both are available.

4.6 Results

In this section, the runtime and optimality of our approach are compared to that of a traditional synchronous ILP formulation (as described in Section 4.2). The ILP formulation used for comparison is based on that of (Nielsen, 2005) with a further reduced variable space for comparison purposes.

4.6.1 Setup

To illustrate the benefit of our technique, the proposed branch-and-bound approach was compared to the ILP approach in three different scenarios:

1. time-constrained area minimization,
2. area-constrained time minimization, and
3. area-constrained and time-constrained search, selecting the first solution meeting both constraints.

For each case, both solvers were given a maximum run-time bound of 60 seconds, at which point they were terminated and their best results given. The decision to bound run-time was made because the ILP solver could take hours (or days) to complete; by comparison, the worst-case run-time for the branch-and-bound solver was under five seconds in all but one test case.

For the area-constrained time minimization scenario, results could not be produced within the run-time bound for ILP in many cases. As a result, an additional upper-bound schedule latency constraint was added to the ILP formulation to reduce its number of variables and constraints, effectively reducing run-time. This additional constraint was *not* used in the branch-and-bound approach, as it would only improve performance. For two test cases for the ILP solver, the run-time bound was relaxed

until the first solution was found. For a very large test case (1090 nodes), the ILP formulation itself was too complex to complete within the time bound, so no results were recorded.

The parameters corresponding to the functional units used in experimentation are shown in Table 4.3. These parameters were selected to be equivalent to those used in (Saito et al., 2006).

The proposed branch-and-bound approach was implemented in Java using standard packages. The ILP solver used for comparison results was lp_solve (lpsolve, 2009), a free open-source MILP solver with a Java interface. In order to produce results for both methods, a benchmark class file was coded for each DFG to be fed into the appropriate Java interface.

In an additional experiment, the branch-and-bound technique was tested with various optimizations disabled to illustrate the relative effectiveness of each optimization.

The benchmarks were tested using on a Macbook Pro with a 2.8 GHz Intel Core 2 Duo processor and 4GB of RAM and JVM 1.5. Run-times were measured with an accuracy of one millisecond.

4.6.2 Benchmark Description

Five different benchmarks were used in experimentation:

- **ODE**: solves ordinary differential equations using the Euler method. It receives as input the coefficients of a third-degree ordinary differential equation, along additional parameters such as step size.

- **DotProd8**: performs a dot product on two eight-element vectors. This example was used due to its high initial concurrency, resulting in a large number of unique schedules.

- **Cosine**: approximates the cosine of a number using the first nine terms of it's Taylor series.
- **Seventh**: runs a seventh order filter from the IMEC cathedral system. This benchmark was provided by (Nielsen, 2005).
- **Elliptic**: runs a fifth order elliptic wave filter. This benchmark was provided by (Nielsen, 2005).
- **TEA**: performs two unrolled implementations of the unrolled tiny-encryption algorithm in parallel (a very large example).

The node count of each benchmark, including root and sink nodes, is given in Table 4.4.

4.6.3 Discussion of Results

4.6.3.1 Table labels and interpretation

Tables 4.5-4.7 use several acronyms as labels:

- **BB**: results using branch-and-bound approach
- **ILP**: results using synchronous ILP approach
- **TC**: time constraint of the circuit (see Table 4.3)
- **AC**: area constraint of the circuit (see Table 4.3)

In Tables 4.5 and 4.6, items in **bold** indicate an optimal schedule was found with the specific area or latency minimized. In Tables 4.5 and 4.6, items **italicized* with an asterisk indicate that the solver did not complete execution within the 60 second limit, so the time the solver's best solution was found is shown. In Table 4.7, items *italicized* indicate that no solution is possible.

4.6.3.2 Discussion

Table 4.5 shows the results under time-constraint with the objective of minimizing area. Using the branch-and-bound algorithm, the run-time to exhaust the search space to find the minimal area is 50ms or less in 13 of 16 test cases. The ILP approach, however, failed to complete execution within the time bound in 9 of 16 test cases and missed the optimal solution half of the time. In cases where the ILP solver was able to complete execution, run-times were 1.9x-180x longer than that of the branch-and-bound approach. In cases where the ILP solver was unable to complete execution, the branch-and-bound solver outperforms the ILP approach by several orders of magnitude. One additional note is that for the TEA example, which contains 1090 operations, the ILP solver was unable to even generate the variables and constraints within the time bound, and in fact the number of constraints was so great that the JVM eventually exceeded its available heap space (2GB).

Table 4.6 shows the results under area-constraint with the objective of minimizing latency. As previously stated, a time-constraint was added to the ILP formulation after results for ILP were unable to be produced within the 60 second limit. In the case of Elliptic with an area constraint of 100 units, this 60 second time limit was relaxed until its first solution was found at 79 seconds.

Under the area constraint, the branch-and-bound approach generated optimal results within the time bound in 12 of 13 test cases and completed execution in under 40ms in 10 of 13 test cases. For the very large example, TEA, one difficult test case took 73 seconds to find the optimal solution, so the best solution found at 60 seconds is shown. The ILP approach, on the other hand, completed execution in only 2 of 13 test cases, and found the optimal solution in only 3 of 13 test cases. The run-time of the ILP solver exceeded 7.9 seconds in all cases. For the TEA example, again, generation of the constraints did not even complete within the time bound. An improvement of

Table 4.3: Functional unit parameters

Fn Unit	Area (units)	Delay (ns)
Add	8	8
Subtract	8	8
Multiply	48	9
XOR	8	8
Shift	8	8

Table 4.4: DFG nodes per benchmark

Benchmark	# of Nodes
ODE	11
DotProd8	17
Cosine	26
Seventh	31
Elliptic	36
TEA	1090

several orders of magnitude is seen over the ILP approach.

Table 4.7 shows the run-time of both approaches while under both time and area constraints for the benchmark DotProd8. For cases with a valid solution, the branch-and-bound approach produces its first legal result in 5-6ms in all cases. The ILP approach ranges from 121-366ms, 20-60x as long. Both approaches are generally faster when no solution can be found: 3ms for the branch-and-bound approach, and 69-92ms for 4 of 5 test cases for ILP. However, it took 71 seconds for the ILP to determine that no solution could be found in one test case.

Table 4.8 shows the effect of removing individual optimizations from the branch and bound approach. With no optimizations removed, each test case runs in 25ms or less. For the Cosine test case, removing the lexicographical ordering of start nodes (Node+) has the greatest impact on run-time, and removing hashing has the least impact. For Elliptic, removing the shortest-time-to-finish ($STTF$) sort has the greatest impact, while removing lexicographical ordering of finish nodes ($Node-$) and the resource-constrained shortest-time-to-finish ($RCSTTF$) have the least impact. The $RCSTTF$ bound relies on the $STTF$ sort, and includes the $STTF$ bound, so when removing the latter two optimizations, the $RCSTTF$ bound must also be removed.

Table 4.5: Run-time and results for time-constrained area minimization

Benchmark	TC	Run-time (ms) BB	Run-time (ms) ILP	Area (units) BB	Area (units) ILP	ILP Parameters #cons	ILP Parameters #vars
ODE	34	4	61	**160**	**160**	133	179
ODE	50	4	100	**112**	**112**	181	355
DotProd8	35	32	62	**416**	**416**	118	181
DotProd8	50	6	205	**208**	**208**	148	436
DotProd8	90	6	**3600*	**104**	112	228	1116
Cosine	75	9	**5600*	**208**	**208**	235	1047
Cosine	100	16	**1200*	**104**	160	285	1697
Cosine	160	8	**5000*	**56**	112	405	3257
Seventh	90	43	**300*	**168**	264	360	1296
Seventh	100	1044	**500*	**120**	168	390	1606
Seventh	120	11	**900*	**112**	**112**	450	2226
Elliptic	115	23	4151	**168**	**168**	353	2490
Elliptic	120	13	**800*	**120**	176	363	2670
Elliptic	160	9	**22900*	**64**	120	443	4110
TEA	2575	4648	-	**48**	-	11500	45571
TEA	2800	1332	-	**32**	-	12175	49846

4.7 Conclusion

This chapter presented an efficient technique to perform high-level synthesis – a branch-and-bound approach that out-performs the traditional synchronous ILP by orders of magnitude. By quickly finding a quality solution and utilizing safe and aggressive pruning, the proposed approach reduces the search space and solver run-time significantly. Experimentation illustrates its effectiveness for both area-constrained and time-constrained synthesis, showing run-times of 50ms or less in most test cases. Further, the approach presented in this chapter accurately models an asynchronous paradigm by removing the notion of integer time and relying on only actual events. As a result, this approach can find optimal solutions that cannot be feasibly reached in a synchronous ILP approach.

The work presented in this chapter focuses primarily on latency and area of an implementation, scheduling operations in a "single-token" fashion; that is, only one

Table 4.6: Run-time and results for area-constrained latency minimization

Benchmark	AC	Run-time (ms) BB	Run-time (ms) ILP	Latency (ns) BB	Latency (ns) ILP	ILP Parameters #cons	ILP Parameters #vars	ILP Parameters TC
ODE	100	4	9724	**53**	**53**	248	575	70
ODE	150	5	7920	**35**	**35**	248	575	70
DotProd8	150	8	**3100*	**60**	62	253	1286	100
DotProd8	280	9	**29300*	**42**	**42**	193	776	70
Cosine	150	15	**7100*	**97**	125	340	2347	125
Cosine	320	10	**15800*	**69**	84	290	1697	100
Seventh	150	30	**26700*	**95**	119	472	2381	125
Seventh	200	38	**1100*	**83**	100	397	1606	100
Elliptic	100	20	**79400*	**126**	150	428	3750	150
Elliptic	150	14	**55600*	**116**	150	428	3750	150
TEA	70	1478	-	**2560**	-	12776	53646	3000
TEA	50	**59000*	-	2584	-	12776	53646	3000
TEA	30	3736	-	**4608**	-	18776	91646	5000

Table 4.7: Run-time comparison for both time and area constrained synthesis for DotProd8

Time Constraint	Area Constraint 250 BB	250 ILP	200 BB	200 ILP	150 BB	150 ILP	100 BB	100 ILP
45 ns	5	273	*3*	*72*	*3*	*73*	*3*	*69*
60 ns	5	185	6	121	6	270	*3*	*92*
80 ns	6	158	6	145	5	252	*3*	*71041*
100 ns	6	173	6	193	5	261	6	366

instance of the problem is computed at a time. In the following chapter, we will consider a more general problem, *multi-token* scheduling, in order to allow multiple problem instances to be scheduled on the same set of resources concurrently. The search space of this problem is significantly more complex, but will allow us to target the throughput of an implementation rather than latency, improving performance.

Table 4.8: Effect of optimization removal on run-time and total nodes explored

Optimization(s) Removed	Cosine		Elliptic	
	#Nodes (1000s)	Runtime (ms)	#Nodes (1000s)	Runtime (ms)
All optimizations enabled	*0.72K*	*25*	*0.52K*	*22*
No Node+ Ordering	424K	3068	6.65K	91
No Node- Ordering	39.9K	377	1.97K	42
No Node+ or Node- Ordering	4033K	27220	38.1K	405
No RCSTTF Bound	38.7K	423	5.87K	85
No STTF/RCSTTF Bounds	317K	4007	173K	2211
No STTF Sort/RCSTTF Bound	255K	2735	353K	3592
No Node Hashing	2.74K	28	18.6K	139

Chapter 5

Resource-limited Design: Pipelined

In Chapters 3 and 4, I described two different synthesis approaches with opposing goals; Chapter 3 targeted high-performance specifications, while Chapter 4 introduced resource sharing to minimize area. In this chapter, I will propose a hybrid alternative: a novel synthesis approach that merges both pipelining *and* resource sharing for low-area, high-performance circuits.

5.1 Introduction

This chapter introduces a new approach for performing resource sharing in pipelined asynchronous systems. Since the pipelined paradigm is mainly meant for designing high-performance systems, conserving area is secondary to achieving high performance. Therefore, existing approaches to designing pipelined systems typically do not handle resource sharing (*e.g.*, (Budiu, 2003)). On the other hand, high-level synthesis approaches that handle allocation, scheduling and binding of shared resources in an automated fashion generally assume a control-driven architecture (*e.g.*, (Nielsen, 2005; Nielsen et al., 2004; Nielsen et al., 2009)). These latter approaches do not lend themselves easily to fast pipelined multi-token operation. This work attempts to bridge the gap between pipelined data-flow systems and control-driven shared-resource systems.

The key contribution in this chapter is a novel *multi-token* scheduling approach that targets throughput rather than latency. My proposed approach specifically targets resource sharing in a pipelined context, one where multiple instances of the problem are being computed at once. This domain is distinct from that of Chapter 4, which focused on *single-token* scheduling in order to minimize the overall latency.

In particular, it is assumed that a cycle time (or throughput) bound is provided, and the goal is to generate an implementation that minimizes area while meeting the cycle time constraint. The rationale is that typically a performance bound is specified to the designer, and their objective is to reduce area in order to improve yield, lower die costs, and reduce leakage power.

After minimizing function unit area, the designer may perform buffer insertion via slack-matching in order to help meet performance goals without allocating additional function units. My proposed approach, on the other hand, incorporates slack matching during the scheduling and area minimization process, and therefore the minimizes the sum of buffer area coupled with function unit area.

The work presented in this chapter consists of the following contributions: first, I introduce a graphical representation of a system that models both resource scheduling and buffer requirements. Next, I propose an architecture that combines the best of both worlds: a resource-shared, data-flow pipeline. Then, I introduce an approach that concurrently performs allocation, scheduling, and binding of resources along with slack-matching to meet performance targets. Finally, in order to handle large examples for which an exact method is too slow, I introduce a hierarchical method for scheduling on a per-block basis in order to heuristically minimize area for larger examples. This work follows the design flow illustrated in Figure 5.1.

A key feature of the multi-token scheduling approach is that it does not repeatedly perform "unfolding" of the data-flow graph, followed by scheduling, and finally

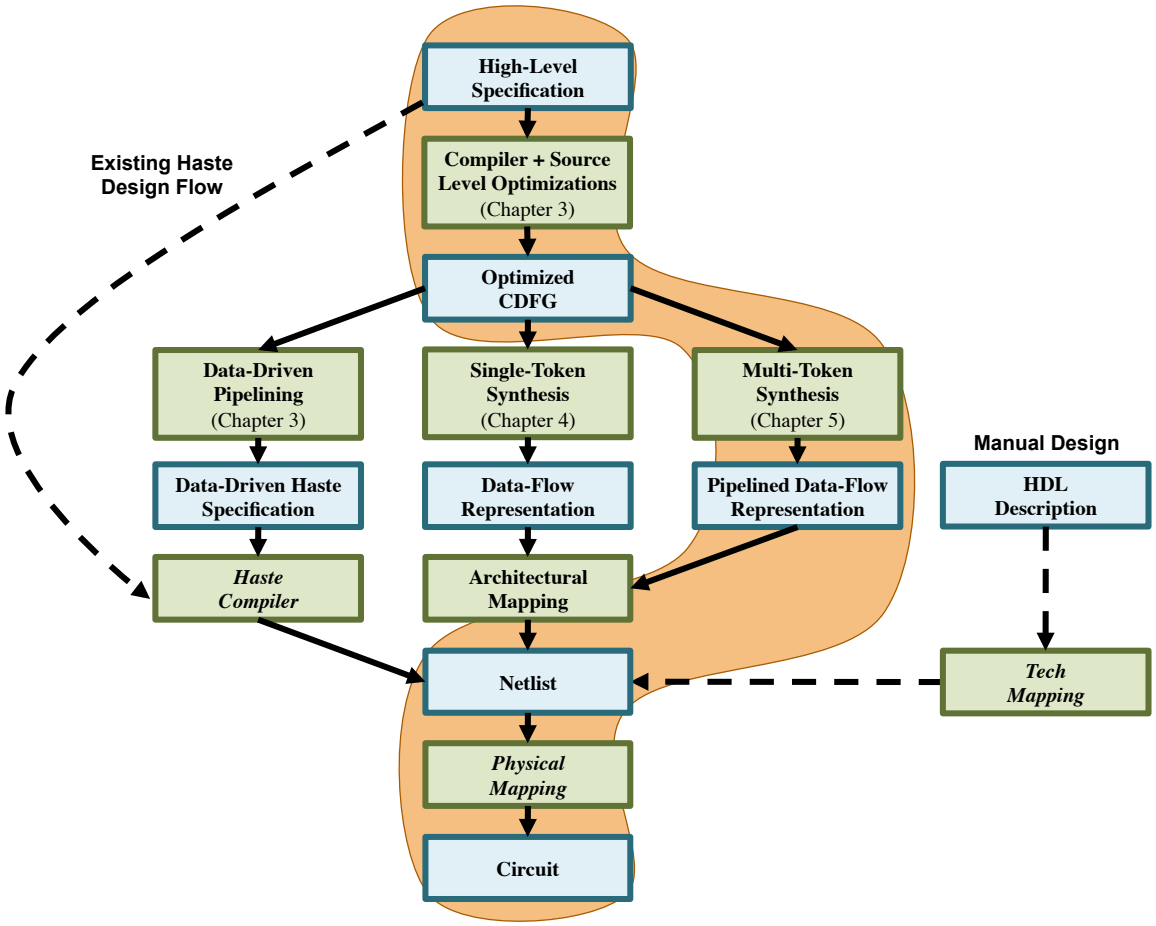

Figure 5.1: Multi-token, shared-resource design flow

compaction in order to determine the schedule for multi-token operation. Instead, it directly determines a compact, multi-token schedule in an optimal fashion for each block.

The class of schedules produced on each function unit are "single-stride cyclic" schedules. Each function unit has a single static schedule that repeats indefinitely. Each operation is mapped to only one function unit, and that operation occurs only once in that function unit's repeated schedule. The approach assumes a *slack-elastic* (Manohar and Martin, 1998b) model for correctness, that is, the order of outputs on a channel must remain the same (no out-of-order execution).

Experimental results are promising. Multiple different test cases were considered, each was synthesized using several different throughput constraints. In each case, our approach performed resource scheduling to meet the throughput constraints and reported the area of the implementation. As expected, as throughput constraints were relaxed, the area of the implementation improved.

The remainder of this chapter is organized as follows. Section 5.2 discusses previous work on synchronous and asynchronous scheduling. Sections 5.3-5.6 introduce my optimal method for scheduling resources in a multi-token fashion. In particular, Section 5.3 gives background on dependence graphs as a graphical model. Section 5.4 describes how the model presented in Section 5.3 is extended by incorporating buffering and resource schedules. Section 5.5 introduces the shared-resource pipelined architecture that implements by the multi-token scheduling approach, then Section 5.6 presents the multi-token scheduling and slack-matching algorithm itself. Next, in order to handle large examples for which an optimal method is too slow to solve, Section 5.7 describes a heuristic hierarchical scheduling method. Section 5.8 presents experimental results, and we conclude with Section 5.9.

5.2 Previous Work

Several techniques have been proposed for performing high-level synthesis of synchronous and asynchronous systems; a general survey of techniques is available in (Micheli, 1994). The majority of proposed techniques are heuristic, such as force-directed scheduling (Paulin and Knight, 1987), list scheduling (Sllame and Drabek, 2002), and others (Badia and Cortadella, 1993; Burns et al., 2004). In the asynchronous realm, synchronous ILP approaches have been adapted in order to approximate optimal schedules, but these approaches may end up being either slow or sub-optimal depending on the discretization of time. Such asynchronous ILP-based approaches have been reported in (Nielsen et al., 2009; Saito et al., 2007).

All of these approaches, however, allow only one problem instance to be computed at a time, limiting their performance substantially. Section 5.8 compares my proposed multi-token approach to the single-token approach described in Chapter 4 to illustrate how our multi-token method can produce higher-performance, lower area circuits that are infeasible in a single-token context.

Other approaches, such as (Tugsinavisut et al., 2006) can allow multiple threads of execution, but the designer must specify how many tokens will exist in the implementation.

In contrast to these approaches, the approach I present in the following sections creates a multi-token schedule, subject to a throughput constraint, that *optimally* minimizes the total function unit and buffer area of a pipeline. This approach searches the full space of multi-token schedules and concurrently performs slack-matching to meet a throughput constraint.

```
while(true){
   a=read();
   b=((3*b)+a)*0.25;
} //Loop A
```

```
while(true){
   a=read();
   b=3*d;
   c=a+b;
   d=c*0.25;
} //Loop B
```

Figure 5.2: Simple code example

5.3 Basic Graphical Model

This section reviews *folded dependence graphs* (Williams, 1991) as a convenient graphical model for representing repeated sets of dependent computations. The next section will introduce extensions to this model for incorporating resource sharing and buffering (*i.e.*, storage).

5.3.1 Dependence Graphs

Dependence graphs are used to model data dependencies between the individual operations in a specification. An example representation is shown in Figure 5.3a, corresponding to the specification in Figure 5.2. Here the graph has been expanded to show data and control dependence across iterations. Each node in the graph represents the an operation with its iteration number as a subscript; each arc represents a dependence between operations.

Because the dependence graph of Figure 5.3a becomes unwieldy as the iteration count increases, a more compact, but equally expressive version, is used: a folded dependence graph (Figure 5.3b). Here, a single node a represents the execution of the operation over all iterations (a_0, a_1, a_2, $\cdots$); the subscripts representing iteration numbers are dropped. A *weight* is associated with each arc to represent the difference in subscripts from the source node to the destination node. Thus, *intra-iteration arcs*, such as the one between operation b and c, will have a weight of 0. *Inter-iteration arcs*,

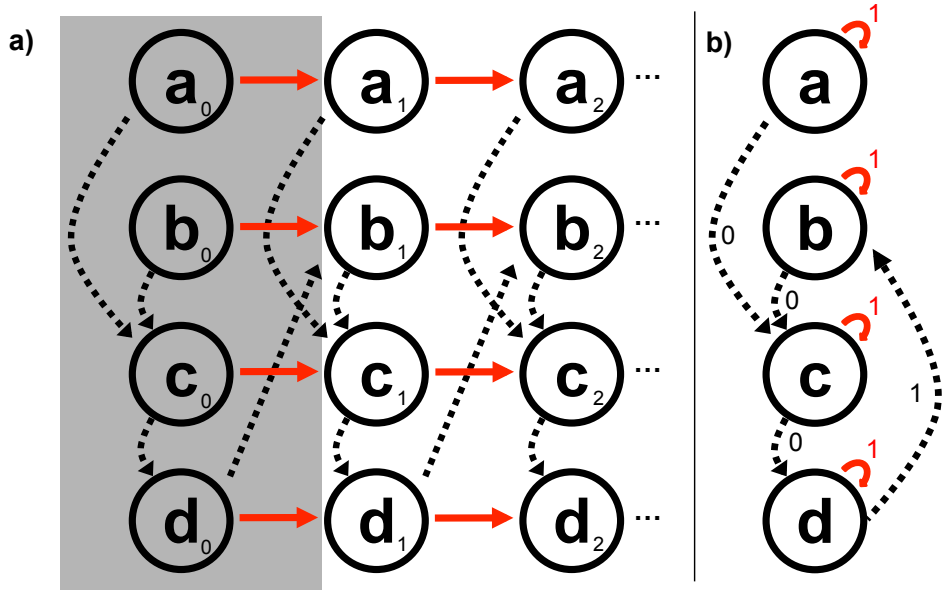

Figure 5.3: a) Unfolded and b) folded dependence graphs

such as the arc from d to b, have a non-zero weight, in this case 1. The self-ordering arcs become self-ordering cycles, each with weight 1.

To ensure liveness of the specification, the following property must be met:

Property 1. *(Liveness)*

$$\sum_{e \in c} \text{weight}(e) \geq 1 \qquad (\forall c \in G)$$

where c is a cycle in the graph G, and e is an edge in c. This property ensures that all cycles must have a weight greater than or equal to one. A cycle weight cannot be less than or equal to 0, because if it were, it would imply a deadlock.

5.3.2 Cycle Time Analysis

For performance analysis, let us now assume that the delays of each operation are given. Using this information, a dependence graph can be analyzed to determine its maximum throughput (or, equivalently, its minimum cycle time). The analysis approach belongs

to the classical category of *maximum cycle mean* computations (Dasdan and Gupta, 1997). This type of analysis has been used for folded dependence graphs (Williams, 1991) and recently in marked graphs (Tugsinavisut et al., 2006).

We briefly review the analysis approach here. Let there be a delay associated with each arc in the folded dependence graph (a "fixed-delay model"). The delay associated with an arc from node x to node y represents the length of time that must elapse from the instant that the x operation completes to the instant the y operation completes. No delays are associated with the nodes; instead all delays are represented on arcs. This delay is distinct from the weight associated with an arc.

The cycle mean for cycle c in graph G is defined as follows:

$$\text{Mean}(c) = \frac{\sum_{e \in c} \text{delay}(e)}{\sum_{e \in c} \text{weight}(e)}$$

where e is an edge in the cycle c. The cycle time is given by the maximum of the cycle means for all cycles in the graph:

$$\text{Cycle Time}(G) = \max_{c \in G} (\text{Mean}(c))$$

Intuitively, this analysis shows that the cycle time of an individual cycle is the total delay of the cycle divided by the number of tokens on that cycle. As described in Chapter 2, the cycle time of the full graph is limited by the worst cycle time of any cycle in the graph.

5.4 Extended Graphical Model

The basic model of Section 5.3 captures data-dependencies (RAW constraints). Now, let us extend that model to incorporate two additional types of dependencies: (i)

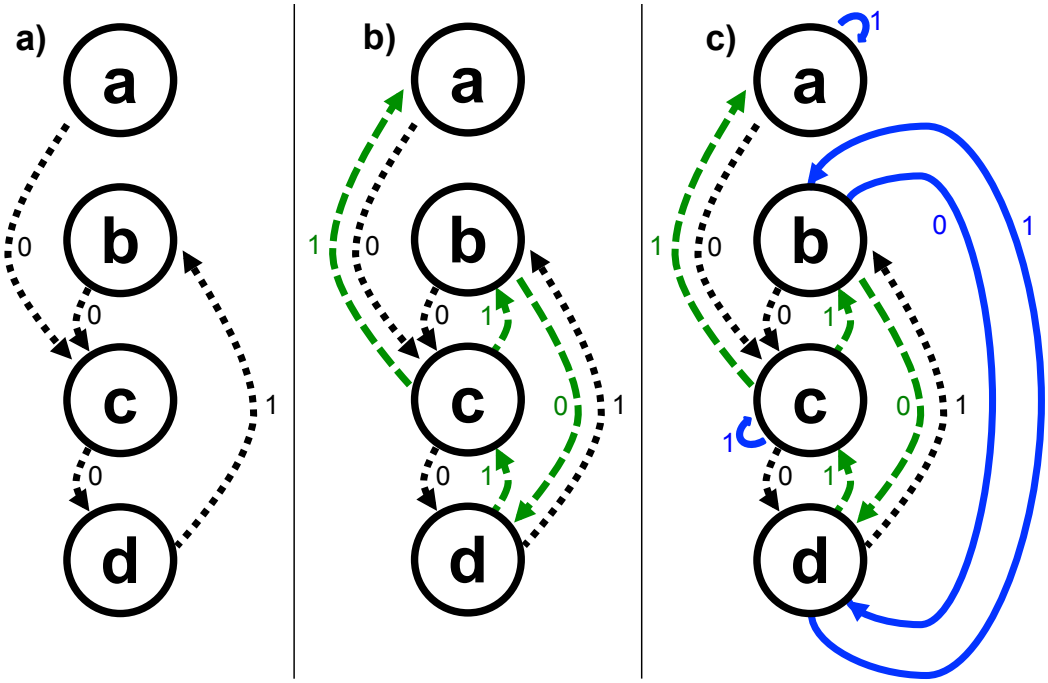

Figure 5.4: Adding a) data, b) buffering, and c) resource arcs to the graph

write-after-read (WAR) constraints, which prevent data from being overwritten until it has been consumed; and (ii) *resource scheduling constraints,* which are necessary when resources are shared. Both of these types of constraints are modeled by adding additional arcs to the dependency graph.

A key contribution of this section is illustrating how buffering (*i.e.*, storage) requirements can be directly inferred from the dependency graph. In addition, this section also describes how the delays of those buffers are modeled appropriately in the dependency graph.

5.4.1 Modeling Write-After-Read (WAR) Constraints

WAR constraints are necessary to ensure that a storage location is written only after its previous value has been read. Because there exists contention for storage, the extent of allowable concurrency in execution scenarios becomes limited.

To illustrate an example of the necessity of WAR constraints in the graph, let us begin by assuming that there is exactly one storage location to store the result of each operation (we will relax this constraint later). Therefore, the result of a new operation cannot be stored into a location that is holding a previously generated value, *i.e.*, one that is still waiting to be used by some other operation. For example, when a is read from the environment in Figure 5.4a, the value of a must remain in its storage element until it is consumed to produce c. Therefore, an execution scenario where a_{n+1} is produced before c_n is illegal because c_n needs the value of a_n (which will have been overwritten by a_{n+1}).

In order to model this restriction, we add WAR arcs to the dependency graph. For each data dependence arc between a pair of nodes, we add a WAR arc between the same nodes in the reverse direction, as shown in Figure 5.4b. Here, the dotted black arcs represent data dependence, and the dashed green arcs represent WAR constraints. In the remainder of this chapter, the terms *WAR arc*, *reverse arc*, and *acknowledgment arc* are used interchangeably.

To appropriately model a single storage location, the sum of the weights on the pair of forward and reverse arcs must equal 1. In the example in Figure 5.4, c_0 enables d_0 through a data dependence, and once d_0 computes, it enables c_1 via a WAR arc. Therefore, because of the difference between the subscripts, the WAR arc from c to d must have a weight of 1. Similarly, WAR arcs must be added between every other pair of nodes with a data dependence arc.

5.4.2 Inferring Buffering Requirements

We can now prove a more general result regarding buffering requirements: the number of buffers required for a data channel between two nodes is simply the sum of the weights of the data dependence arc (*i.e.*, forward arc) and the WAR arc (*i.e.*, reverse

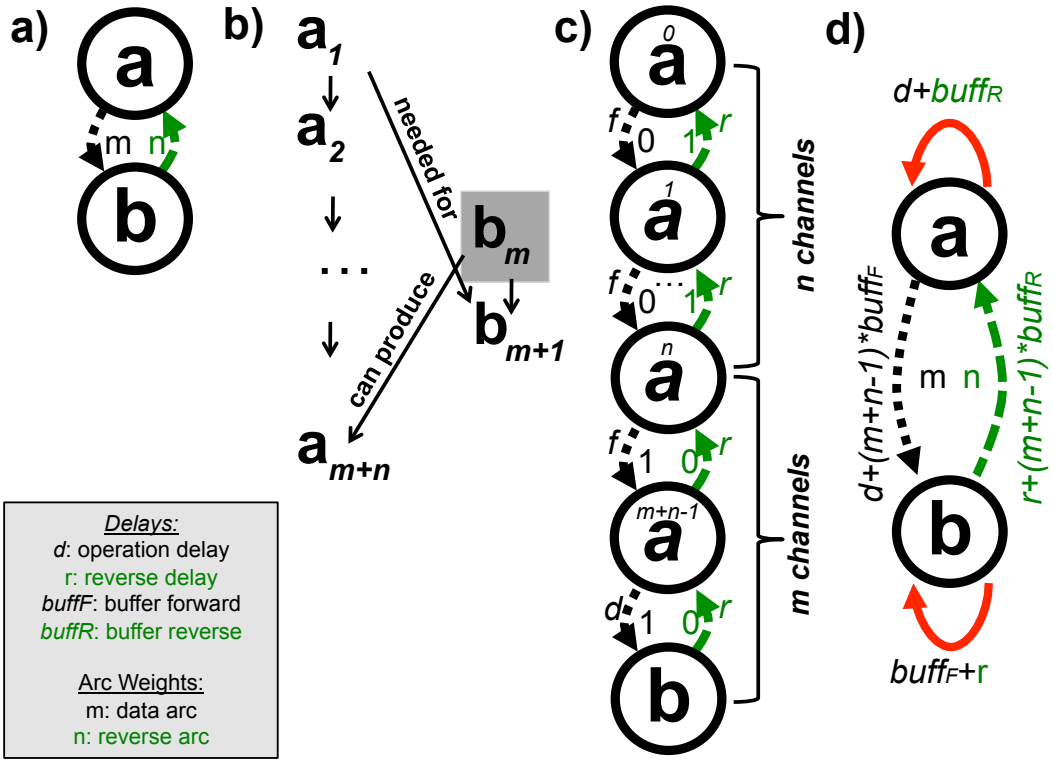

Figure 5.5: Inferring buffering requirements and modeling buffer delays

arc).

Theorem 1. *Given a data channel between two nodes a anqd b, with m the weight of the forward arc, and n the weight of the reverse arc (as shown in Figure 5.5a), the number of buffers required for correct operation is* $m + n$.

Proof. Assume an instant in time such that b_m has occurred (and therefore b_k for all $k < m$ have also already occurred), but b_{m+1} has *not* occurred. Then, by virtue of data dependence, a_0 must have occurred. Also, because of the reverse arc from b to a, a_{m+n+1} cannot have occurred yet since b_{m+1} has not occurred. At this point in time, the events $a_1 \cdots a_{m+n}$ may occur before any further events on b, and therefore the results from each of these must be stored and preserved as the future events $b_{m+1} \cdots b_{2m+n}$ will need them. As a result, up to $m + n$ buffers may be required to queue up the values $a_1 \cdots a_{m+n}$. Figure 5.5b graphically illustrates the proof. □

5.4.3 Modeling Buffer Delays

When buffers are present on a data channel, their delays must be correctly included during timing analysis. In particular, the forward latency through the buffers will add to the total delay from the source node to the destination node. In addition, each buffer also has a *reverse latency:* the time from the instant the buffer is emptied to the instant its predecessor is enabled to produce the next value.

The proposed approach to modeling the delays due to buffering is illustrated by Figure 5.5c. In the figure, the node a is replaced by $m + n$ new nodes, numbered $a^0 \cdots a^{m+n-1}$, each new nodes representing a distinct buffer. After buffering a, there are now a total of $m+n$ data channels strung end-to-end from node a^0 to node b. For m of these channels, chosen arbitrarily, set the weight on the forward arc to 1 and on the reverse arc to 0. For the remaining n of these channels, set the weight on the forward arc to 0 and on the reverse arc to 1. This selection can be done arbitrarily; it merely determines which of the intermediate channels are initialized full versus empty. The example in Figure 5.5c has chosen n channels near the top and the m channels toward the bottom.

Next, let us confirm that this new graph preserves all the constraints of the original graph, and then determine how to correctly assign the delays to the arcs for correct timing analysis.

Theorem 2. *The graph of Figure 5.5c preserves all the constraints of the graph of Figure 5.5a.*

Proof. The forward arc in original graph implies the constraint $a_k \rightarrow b_{k+m}$. In the new graph, there is a transitive dependence $a_k \rightarrow a_k^1 \rightarrow \cdots a_k^{m+n-1} \rightarrow b_{k+m}$ enforcing the same constraint. The reverse arc in the original graph implies the constraint $b_l \rightarrow a_{l+n}$. Similarly, in the new graph, there is a transitive dependence $b_l \rightarrow a_l^{m+n-1} \cdots a_{l+n}$. □

For correctly modeling the timing behavior, we can set the delays along the arcs are as follows:

- the weight of the forward arcs $a^0 \rightarrow a^1 \cdots a^{m+n-2} \rightarrow a^{m+n-1}$ is equal to the buffer forward latency, $buff_f$
- the weight of the forward arc $a_{m+n-1} \rightarrow b$ is equal to d
- the weight of the reverse arc $b \rightarrow a^{m+n-1}$ is equal to r
- the weight of the reverse arcs $a^{m+n-1} \rightarrow a^{m+n-2} \cdots a^1 \rightarrow a^0$ is equal to the buffer reverse latency $buff_r$

Therefore, for timing modeling and analysis, we can use the simplified graphical representation of the channel as shown in Figure 5.5d by setting the delays appropriately:

- set the forward arc delay from a to b equal to $d + (m + n - 1) * buff_f$
- set the reverse arc delay from b to a equal to $r + (m + n - 1) * buff_r$
- add a self loop on a with weight $buff_f + buff_r$
- add a self loop on b with weight $d + r$

With these delay assignments, the computation of the maximum cycle mean for any graph that contains the sub-graph of Figure 5.5c will be correctly computed by including instead the sub-graph of Figure 5.5d.

5.4.4 Modeling Resource Sharing

Scheduling of shared resources is modeled by adding new arcs to the dependence graph, called *resource arcs*. In particular, one cycle of resource arcs is created for each available

resource. The delay associated with each of these resource arcs is the latency of that resource.

The sum of the weights of the arcs in each such cycle is equal to 1 as the proposed multi-token approach only considers cyclic schedules with a unit stride. As an example, if a certain function unit executes the sequence of operations $a_i, b_j, c_k \cdots$ in one iteration, then the same function unit must execute the same sequence of operations in the next iteration, $a_{i+1}, b_{j+1}, c_{k+1} \cdots$. Therefore, the weight of each resource cycle will be equal to 1.

Property 2. *(Unit Stride Property) The cycle weight for a resource cycle with unit stride must be equal to* 1.

Example

Figure 5.4c illustrates the addition of resource scheduling arcs (solid blue). Here, only one multiplier is available, so it must be shared by the two operations, b and d. Therefore two resource arcs are added to the graph, one from b to d and one from d back to b, each with a delay equal to the multiplier latency. The sum of the weights of these arcs is equal to 1, which represents the execution sequence $b_k \rightarrow d_k \rightarrow b_{k+1} \cdots$. Further, assume one adder resource is available. Since only operation c uses an adder, the corresponding resource arc is a self-cycle on c with a weight of 1 and a delay equal to the adder latency. Operation a similarly uses a "channel-read" resource and has its own self cycle, with a weight of 1 and a delay equal to the input's cycle time. □

In practice, the delays on each arc consist of overheads beyond the operation latency. The controller delay associated with a function unit, multiplexing delay, and the forward delay of a buffer stage are also incorporated,

5.4.5 Converting the Graph to Architecture-Ready Form

Once a graph has been scheduled and slack-matched using the model above, one additional step is performed to prepare the graph for conversion into hardware. The method for mapping a graph to a hardware implementation (to be described in Section 5.5) is straightforward, provided there are no negative weights on any arcs in the graph. Therefore, we must remove these negative arcs, and do so in such a way that the schedule, circuit performance, and buffer requirements are preserved. This re-weighting transformation has some parallels to the problem of retiming (Leiserson and Saxe, 1991).

Here, we prove that any graph with negative arc weights can be converted to an equivalent non-negative graph which we call the *architecture-ready* form by following a series of transformations under the constraints above.

To begin with, let us define the method of *re-weighting*. In this method, we select a node that has all positive incoming arcs, reduce the weight of the smallest positive incoming arc to 0, and add that difference to the outgoing arcs. Since we are adding the same value to every outgoing arc that we are subtracting from the incoming arc, this method preserves the total weight on any cycle going through the node. One key aspect of re-weighting is that the weight of any non-negative arc can never become negative through re-weighting.

Theorem 3. *For a deadlock-free, strongly-connected graph, there must be at least one node in the graph that has positive weights on all its incoming arcs.*

Proof. We prove this by contradiction. First, we select an arbitrary node and begin to trace a path in the graph in reverse until a cycle is formed. At each visited node, we follow the path of the smallest weighted incoming arc. As we step through this path to each new node, each arc has a non-positive weight. Since there are a finite number of nodes that can be reached before a cycle is formed, and since the graph is

strongly-connected, a cycle must eventually be formed. This cycle must have a total weight of less than 1, implying deadlock. Hence, by contradiction, a node must exist that has positive weights on all its incoming arcs. □

Corollary 4. *For a deadlock-free, strongly-connected graph, we can perform an infinite number of re-weightings.*

Proof. This corollary is trivially true, as re-weighting can occur on any node as described in Theorem 3, and each graph is guaranteed to have such a node. □

Lemma 5. *For a deadlock-free, strongly-connected graph, all nodes in a graph will have been re-weighted after a finite number of re-weightings.*

Proof. We begin by considering a node that has not been re-weighted, X. Because our graph is strongly-connected, there is a path from X to every other node in the graph. For a path from X to any other node in the graph Y, we can sum up all the weights on this path to give a finite value. Because Y can only be re-weighted if its incoming arc is positive, this sum corresponds to the maximum number of times that Y could be re-weighted before X must be re-weighted.

Since there are a finite number of nodes in the graph, and since each can only be re-weighted a finite number of times before X is re-weighted, there are a finite number of re-weightings that can occur before X must be re-weighted. Since the number of legal re-weightings is infinite according to Theorem 3, X must eventually be re-weighted. By the same token, all nodes in the graph must be re-weighted within a finite number of re-weightings. □

Theorem 6. *Any deadlock-free, strongly-connected graph that contains arc(s) with negative edge weight(s) can be converted into an equivalent graph where no arcs have negative edge-weights.*

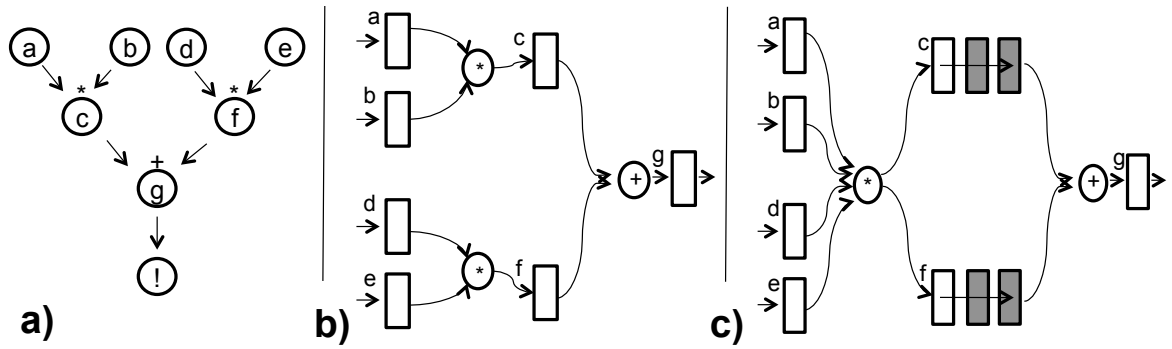

Figure 5.6: a) Sample DFG, b) unshared architecture, and c) shared architecture with buffering

Proof. According to Theorem 5, all nodes in the graph must be re-weighted after performing a finite number of re-weightings. Therefore, we can perform re-weighting in any legal order until each node has been re-weighted at least once. Since the re-weighting process cannot reduce any arc's weight below 0, and each arc has been re-weighted to have a weight of at least 0, the graph cannot contain any negative arcs after each node has been re-weighted. In addition, because re-weighting does not change the total weight of any cycle that contains the node (or the delays, for that matter), the minimum cycle time remains the same. Therefore, the graph is equivalent, and non-negative. □

Because the total weight on a cycle remains unchanged, each channel will have the same number of buffer stages after the conversion process. Additionally, since no arcs were added, removed, or redirected, the cyclic schedule on a resource remains the same.

5.5 Architectural Model

This section introduces a data-flow, shared-resource architecture that implements the extended graphical model of Section 5.4. I will begin with a general overview of the datapath, then discuss the different types of components used in the proposed architecture:

buffers, forking data latches, and resources (function units).

5.5.1 Overview

An diagram illustrating the basic architecture is illustrated in Figure 5.6. Figure 5.6a shows a simple DFG that performs a dot-product of two two-element vectors:

$$< a, c > \cdot < b, d >= a \cdot b + c \cdot d$$

Figure 5.6b shows a basic architecture for this DFG without resource sharing. Finally, Figure 5.6c shows our architecture with a shared multiplier and additional buffers on two data channels. This example features the three key components in the proposed architecture: (i) storage locations for variables ($a - g$) that come directly from the environment or function units, (ii) extra data buffers (in gray), and (iii) resources (shared or dedicated).

To generate an architecture from a given dependence graph, we begin by replacing each node in the graph with a data latch. This step ensures that we have at least one storage location for each variable in the original specification. Then, between nodes with data-dependencies, we build a channel that consists of zero or more additional buffers, necessary for slack matching and data synchronization. Multiple channels may be generated from the same data latch source, since a variable may be needed for different computations, but each channel from the same source variable may contain a different number buffers. At the end of a channel, the final buffer feeds into a function unit. The function unit will, in turn, feed into a new data latch.

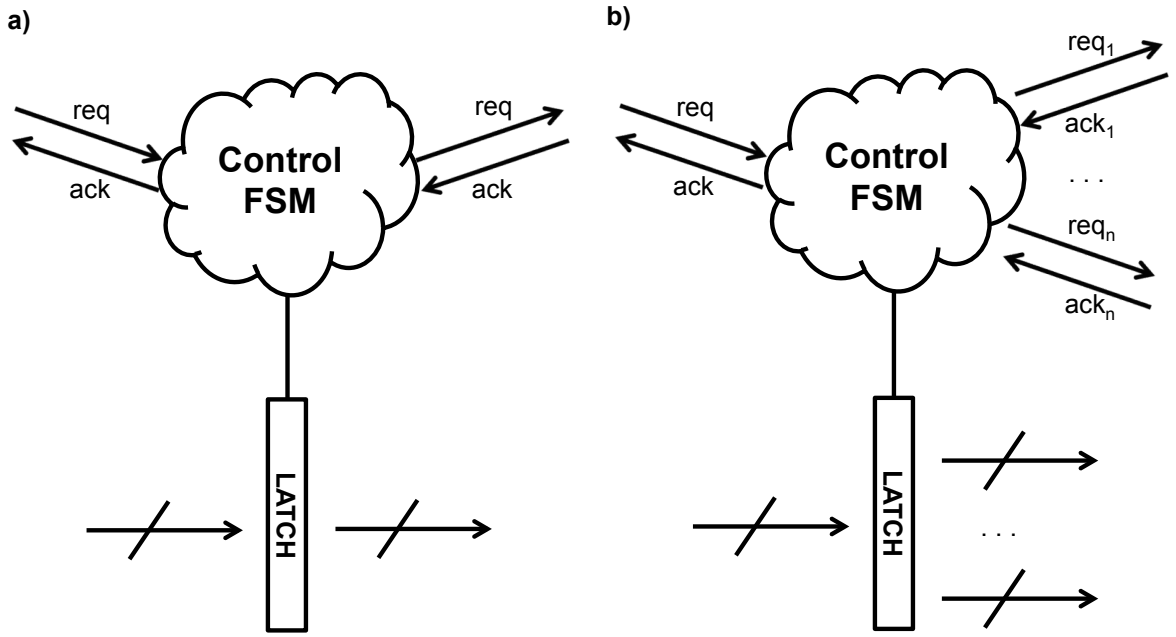

Figure 5.7: a) Buffer and b) forking data latch implementations

5.5.2 Components

5.5.2.1 Buffers

The purpose of a buffer in this architecture is (i) hold older data while new data is being computed, preventing old data from being overwritten, and (ii) to improve performance via slack-matching, as described in Section 5.3.2. A series of buffers may be placed on a channel between a data latch and the function unit it feeds into. The total count of all buffers on a channel (including a forking data latch) is described in Section 5.4.

The buffer stage consists of a basic storage element manipulated by a simple controller, as shown in Figure 5.7a. The behavior of a single buffer stage repeats as follows: (i) wait for an incoming request, (ii) latch data, acknowledge, send an outgoing request, (iii) wait for acknowledgement. While we have selected this specific pipeline style, other pipeline styles can certainly be used (refer to Chapter 2.1 for alternatives).

Based on the architecture-ready graph produced by our algorithm, a buffer stage will either be initialized as full (a 1 on a forward data arc) or empty (a 1 on the reverse

data arc).

5.5.2.2 Forking Data Latch

The purpose of a forking data latch in the proposed architecture is to capture the output of a function unit and then forward the data down one or more buffered channels. The data latch is similar to a buffer in terms of behavior and design, with the exception that it may fork its data to multiple channels. Therefore, it sends multiple outgoing requests concurrently, and must wait for all of them to be received before accepting new data. The diagram for a storage unit is shown in Figure 5.7b.

Like a standard buffer, a forking data latch will either be initialized as full or empty, although this initialization occurs on a *per-channel* basis.

Because a forking data latch is shared across channels, the number of additional standard buffers on a channel is one less than the sum of the forward and reverse arcs that constitute the channel.

Note that a *joining* latch is not necessary since synchronization of data occurs at a function unit.

5.5.2.3 Function Unit and Control

Function units may be dedicated or shared. If dedicated, no complex control is necessary. If a function unit is shared, there will be multiple inputs to be multiplexed and outputs that need to be routed. The controller for each function unit has N input channels and M output channels; for a binary function unit, $N = 2M$, for a unary function unit $N = M$. The diagram for a function unit is shown in Figure 5.8.

All of the handshake channels feed into a state machine that controls the schedule of operations on the function unit. This state machine is not global, but is instead a local controller, one per function unit. The state machines repeats the following steps indef-

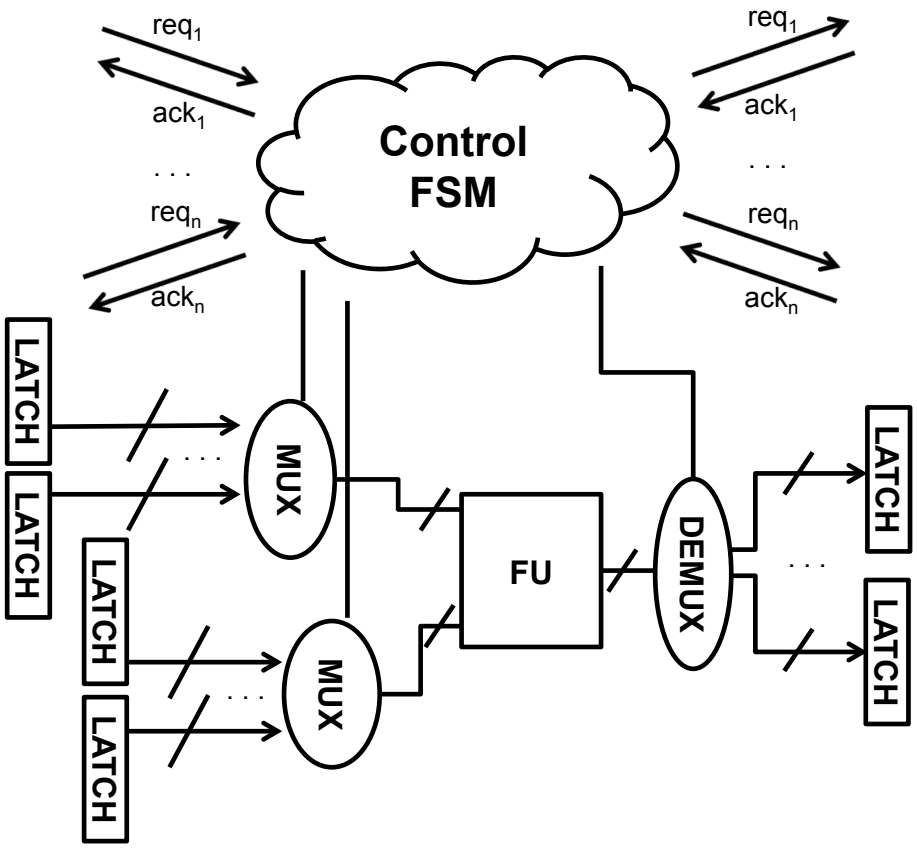

Figure 5.8: Shared resource implementation

initely: (i) consult schedule to set input and output multiplexers, (ii) forward incoming request to data latch with appropriate matched delay, (iii) forward acknowledgement from data latch to inputs.

5.6 Optimal Problem Formulation

In this section, I describe an optimal approach for synthesis. I first give a top-level overview of the approach, then describe a branch and bound strategy for scheduling, allocation, and binding of resources. Next, I describe an ILP-based approach for verifying the throughput constraint by performing slack-matching.

5.6.1 Overview of Approach

The multi-token scheduling problem can be broken down into two specific sub-problems: (i) scheduling and allocating function units, and (ii) verifying that the schedule meets the throughput constraint after optimal buffering. Therefore, the proposed solution has been broken down into two phases: a branch-and-bound scheduling phase, and an ILP-based technique for optimal buffer insertion and ensuring satisfaction of the throughput constraint.

At the top level, the proposed approach steps through the scheduling process for each function unit, allocating additional units as necessary. This branch-and-bound algorithm fully schedules each resource one by one. As each function unit is scheduled, an ILP instance is run to ensure it meets the throughput constraint given the opportunity for buffer insertion. Once a legal schedule is found with all operations scheduled that meets the cycle constraint, this schedule compared with the best solution so far, and searching continues until no better schedules can be found.

5.6.2 Scheduling, Binding, and Allocation: Branch and Bound

The branch and bound portion of the proposed approach begins with the original graph with all the data-dependencies in place. Next, a reverse arc is added between each data-dependent node in order to produce a complete channel.

A basic version of the recursive scheduling algorithm is given in pseudocode in Figure 5.9, the steps are as follows:

1. Generated a list of unscheduled items, sorted lexicographically. Select the first unscheduled item from the ordered list.

2. Create a list of resources on which this item could execute, subject to an area constraint.

3. Explore scheduling the operation on each one of these resources in a depth-first fashion.

4. After an operation has been scheduled, create a list of unscheduled nodes remaining that could execute on the same resource. Also include in this list the first node on this resource's schedule in order to complete the resource's cycle.

5. Explore scheduling each child operation on this resource in a depth-first fashion, adding a resource arc from the previously scheduled node to the current node.

6. If the resource cycle has been closed, compute the buffering needed to achieve the throughput constraint by running ILP described in 5.6.3. If buffering cannot meet the throughput constraint, or if the total area exceeds the best area, we stop exploring this partial schedule.

7. If there are unscheduled nodes remaining, return to Step 1. Otherwise, a new best area solution has been found. This value is recorded and scheduling continues.

Beyond the basic bounding performed in Step 6, we can improve run-time by adding a few additional optimizations. The first optimization is to estimate the minimum area for unscheduled operations by using utilization analysis, and use this value to help prune.

Second, after calculating the minimum amount of buffers needed for a partially scheduled implementation, we know that this amount cannot decrease as we continue to schedule more items. The justification for this pruning is that adding an additional scheduling arc to the graph can only reduce performance, therefore the number of buffers needed to improve performance must be monotonically increasing as more arcs are added to the graph.

Third, we sort child nodes in the tree by choosing dependent nodes first. Since these arcs already exist in the graph, the resource arcs may end up becoming redundant, and

```
procedure scheduleOptimally(Block){
   if no resource selected or resource cycle closed {
      slack_match_ILP()
      if slack match failed
         exit with failure
      if all operations scheduled
         update bestArea
      nextOp = select next unscheduled operation
      for each legal resource for nextOp {
         allocate resource
         bind nextOp to resource
         scheduleOptimally(resource, block)
         unbind nextOp
         deallocate resource
      }
   }else{
      for each unscheduled operation eligible for curResource {
         bind operation to resource
         scheduleOptimally(resource, block)
         unbind operation
      }
   }
}
```

Figure 5.9: Basic optimal area-minimization algorithm

therefore may not limit performance. These pruning techniques are safe, and thus do not affect the optimality of the results.

A final optimization is to run the ILP on *partial* function unit schedules, *i.e.*, those that do not have their cyclic schedule closed. This method is employed via backtracking at any point when additional buffers have been inserted to meet the throughput constraint. This optimization allows the scheduler to determine at what specific scheduling step additional buffers became necessary, rather than relying on the full schedule to be enumerated.

5.6.3 Buffering and Cycle Time Constraints: ILP

After the step of scheduling each specific resource, the result must be confirmed to meet the performance constraint specified by the designer. In order to meet the throughput constraint, additional buffers may be inserted automatically by the algorithm. Because the designer's goal is area minimization, we aim to minimize the count of these additional buffers.

The steps of buffer insertion and confirming that a schedule meets the throughput constraint are performed in tandem using an ILP approach. In this process, we will insert the performance constraints as linear constraints in the ILP, and allow the solver to vary the number of buffers used. The sum of buffers in the implementation will be the minimization target.

The following notation is used below:

- $\mathcal{F}$: the set of *forward* arcs (data dependencies)
- $\mathcal{R}$: the set of *reverse* arcs (WAR constraints)
- $\mathcal{S}$: the set of resource *scheduling* arcs
- $\mathcal{C}$: the set of *cycles* in the dependence graph
- $\mathcal{CS}$: the set of cycles consisting solely of scheduling arcs

5.6.3.1 ILP Variables and Constants

The set of variables to be determined is:

- weight(e) for each $e \in \mathcal{R} \cup \mathcal{S}$

The set of known values/constants in the ILP are:

- weight(e) for each $e \in \mathcal{F}$

- T: the *target cycle time* specified by the designer

- $ch_{\#}$: the number of channels in the graph

- $n_{\#}$: the number of nodes in the graph

5.6.3.2 Cost Function

The cost function to minimize is simply the total number of buffers required. As described in 5.4.2, the total number of buffers required on a channel is given by the sum of the weights on the forward and reverse arcs that constitute that channel. However, if a node has more than one output channels (*i.e.*, it represents a fork), then the first latch is common to all channels; any additional buffers added are disjoint. Therefore the cost function is:

$$\sum_{e \in (\mathcal{F} \cup \mathcal{R})} \text{weight}(e) - ch_{\#} + n_{\#}$$

5.6.3.3 Constraints

For each cycle in the graph, we need to enumerate three sets of constraints to ensure that (i) the liveness property is met; (ii) only schedules with stride of 1 are allowed; and (iii) the performance target is met.

Liveness constraint According to Property 1, the sum of the weights on a cycle must be greater than or equal to 1:

$$\sum_{e \in c} \text{weight}(e) \quad \geq \quad 1 \quad \text{for all } c \in \mathcal{C}$$

Unity stride of schedules According to Property 2, the cycle weight for a cycle consisting solely of resource scheduling arcs must be equal to 1:

$$\sum_{e \in c} \text{weight}(e) \quad = \quad 1 \quad \text{for all } c \in \mathcal{CS}$$

Performance constraint Section 5.3.2 explains how the minimum cycle time is computed for the graph. Therefore, the cycle mean for each cycle in the graph must be less than or equal to the target cycle time specified, T:

$$\text{Mean}(c) \quad \leq \quad T \quad \text{for all } c \in \mathcal{C}$$

Since $\text{Mean}(c) = \frac{\sum_{e \in c} \text{delay}_{(e)}}{\sum_{e \in c} \text{weight}_{(e)}}$, this constraint is rewritten as:

$$\sum_{e \in c} \text{delay}(e) \quad \leq \quad T \cdot \sum_{e \in c} \text{weight}(e) \quad \text{for all } c \in \mathcal{C}$$

Note that the the expression for delay(e) will, in general, include delay terms for forward and reverse buffer latencies, which is in turn dependent on the number of buffers required on the corresponding data channel. As discussed in Section 5.4.3, the number of buffers is determined by the sum of the weights of the forward arc (known constant) and the weight of the reverse arc (a variable in ILP). Therefore, the cycle mean constraints are linear in the variables.

5.6.3.4 Implementing the circuit

After the ILP determines the fewest buffers needed for a completely scheduled implementation, we can then extract the values on each arc in order to produce our final schedule. The first step in this process is to convert any negative-weighted arcs to positive arcs, as described in Section 5.4.5, to generate an architecture-ready represen-

tation.

After converting to this representation, each resource cycle will have exactly one arc with a positive weight. This location of this arc in the resource cycle indicates where the cyclic schedule for the resource starts. In particular, the destination node of this positively-weighted arc will be the first operation that will execute on the node, followed by the next node on the resource cycle, and so on until the cycle is closed.

The state of the buffers is determined by the value on the forward and reverse arcs. For a specific channel, if there is a weight of 5 on the forward arc, and a weight of 1 on the reverse arc, 5 buffers on the channel will initialize full while 1 initializes empty.

5.7 Hierarchical Extension: Block-based Modeling

5.7.1 Overview

While the approach described in Section 5.6 can quickly provide optimal solutions for a specific set of examples, an exact approach can become too complex for significantly larger specifications. Therefore, as an alternative to the scheduling approach described in Section 5.6, I now propose a hierarchical method specifically for dealing with large, real-world examples by hierarchically scheduling portions of the graph (*blocks*) individually and replacing them with a simplified model. In addition to the performance benefits of a hierarchical approach, my proposed method also accepts a more general class of specifications, allowing both loops and conditionals to be scheduled.

Because the complexity of the optimal multi-token scheduling problem may grow exponentially with the size of the benchmark, an exact approach is not tractable for large problems. Therefore, instead of scheduling the full graph (Figure 5.10a) at once, the graph can be scheduled in multiple sections, or *blocks*, as shown in Figure 5.10b. Each block in the graph can be scheduled separately with its own disjoint set of resources,

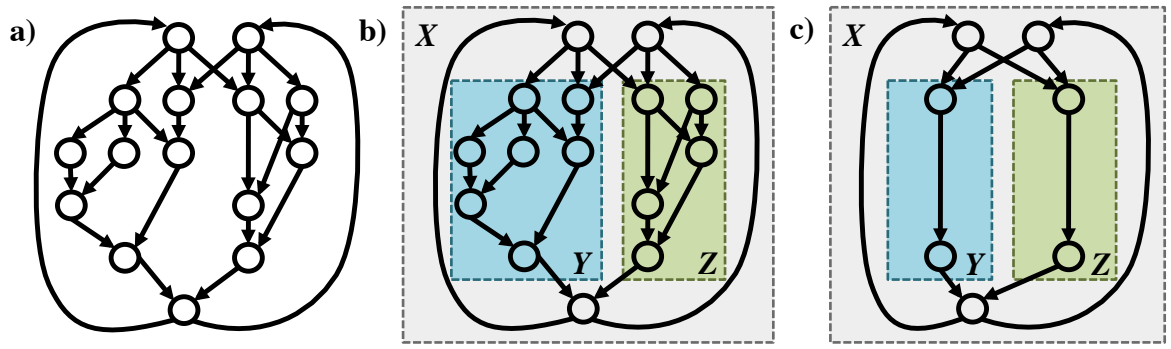

Figure 5.10: a) Original DFG, b) block-partitioned DFG, and c) block-partitioned DFG after blocks Y and Z are scheduled and simplified

minimizing area while meeting the target throughput constraint. After scheduling, a block can then be abstracted into a simpler model, to be easily incorporated into larger blocks that have yet to be scheduled, as shown in Figure 5.10c. This procedure repeats for each nested layer of blocks until the full graph is scheduled.

In comparison to the optimal algorithm of Section 5.6, this algorithm may not produce a globally optimal solution, but it will take significantly less time to compute. Because resources are no longer shared across blocks, this method will essentially trade off optimality for efficiency, dependent on the size and partitioning of blocks. While the final solution may not be the exact optimal solution in terms of area, it is guaranteed to meet the throughput constraint.

The remainder of this section is organized as follows. Subsection 5.7.2 will describe the type of input specifications allowed by this approach. Next, Subsection 5.7.3 defines a *block*, and explains how blocks are abstracted into a simpler model. Subsection 5.7.4 then illustrates how blocks are combined, using parallel and serial composition as examples. Finally, Subsection 5.7.5 describes a heuristic algorithm for area minimization based on the hierarchical constructs and transformations described.

5.7.2 Input Specifications

Unlike the approach described in Section 5.6, the proposed hierarchical approach is more general, allowing for loops, conditionals, and nested blocks. However, there are some restrictions placed on input specifications:

1. Channel communication within the body of a loop is disallowed, since multiple instances of the problem are occurring within the same loop concurrently, and therefore the channel transactions will necessarily occur out of order. If the environment is capable of producing and consuming these values out of order, this restriction can be relaxed.

2. Cross-problem feedback in loops is disallowed. The cost of synchronizing data across problem instances in loops (in terms of both area and performance) becomes prohibitive as the iteration count increases. However, cross-iteration dependence in loops *is allowed* (*i.e.*, data synchronization across loop iterations for the same token/problem), as well as cross-problem feedback outside of loop structures.

3. Variable iteration-count loops are disallowed. The primary reason is to prevent out-of-order execution. Additionally, modeling the performance of stochastic loops can become tricky, particularly when trying to model the performance of a combination of loops. One can conceivably place a re-order buffer at the end of each loop to solve the problem of out-of-order execution, but the proposed approach cannot accurately model how performance would be impacted.

5.7.3 Modeling Blocks

5.7.3.1 Block Definition

In the source CDFG provided by a specification, two types of blocks can be defined: a basic block (one that contains only DFG nodes), and a more general block, which may contain blocks itself.

A basic block is a selection of multiple DFG nodes that are usually connected to each other via dependence. The block's boundaries may be defined by the original source specification (*e.g.*, the body of a function, procedure, loop, conditional, etc.), or may be partitioned automatically by the tool if the source block is larger than a designer-specified limit.

A general block, on the other hand, may contain not only DFG nodes but one or more lower-level blocks as well. In the following subsections, use of the term *block* will refer to the more encompassing class of *general blocks* (rather than basic blocks).

As an illustration of a block in the graph, refer again to Figure 5.10b. Here we see two nested basic blocks, Y and Z inside a general block X.

At the edge of each block there exists an interface to external nodes and blocks. This block interface consists of the channels in and out of items at the edge of a block. These channels exist to carry data between pairs of dependent nodes that straddle a block's boundary.

Figure 5.11 illustrates the interface to Block Y from the earlier example in Figure 5.10b. Here only the data arcs (solid black arcs) and buffering arcs (green dashed arcs) are shown. An internal interface node for a block is one that exists inside the block but has channels exiting the block; these are highlighted in blue and labeled A, B, and C. Similarly, an external interface node exists outside the block, but has channels connecting to an internal interface node for a block; these are highlighted in orange.

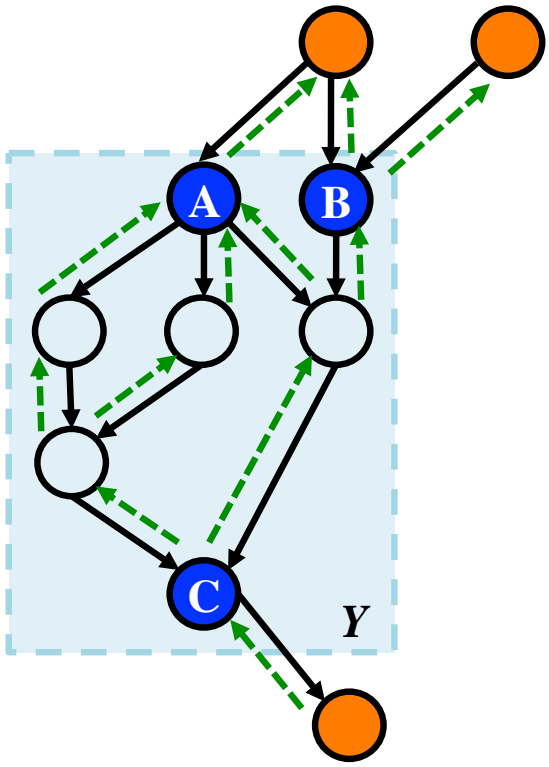

Figure 5.11: Block Y and its associated internal interface nodes, A, B, and C

5.7.3.2 Interface-Level Abstraction

In the proposed hierarchical approach, each individual block will be scheduled independently from other blocks in the graph (the full procedure will be described in Subsection 5.7.5). However, in order to easily combine blocks at higher levels of hierarchy without a prohibitive amount of computational complexity, each block must be abstracted into a simpler representation. One can successively simplify the block model and reduce the search space by performing the following: (i) ignoring internal cycles in the scheduled block and considering only paths between nodes on the interface of the block, (ii) reducing the number of arcs between interface pairs by approximation, and (iii) reducing the number of nodes on the interface by synchronizing inputs and outputs to a single node each.

In the following discussion, let us consider Block Y, as shown in Figure 5.12a. Here

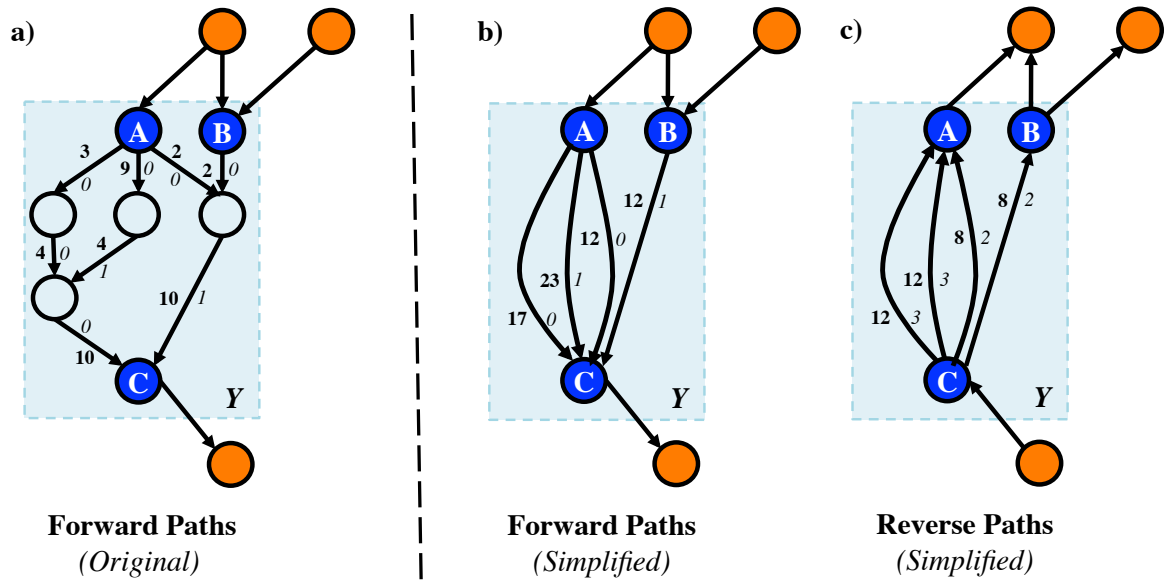

Figure 5.12: Simplifying the internals for Block Y (*reverse path not shown for original graph*)

we have re-introduced the notion of weights and delays on arcs; recall that the delay is the time associated with the computation of a node, while the weight is a difference in iteration count between two operations. For simplicity, only the forward arcs are illustrated in this figure; let us assume the reverse arcs all have a weight of 1 and a delay of 4. We will also assume the remainder of the graph consists of straight-line pieces of code (no loops, conditionals, or feedback). Now, let us begin simplifying the representation of Block Y.

In order to schedule the whole graph concurrently, we can follow the method outlined in Section 5.6 to achieve the optimal solution. Block Y will share a set of resources with the rest of the graph, since scheduling of resources can occur across blocks. However, as noted, this approach can become unwieldy as the number of nodes increases. Instead, let us assume that Block Y has been scheduled individually with its own set of resources, independent from the rest of the graph. Turning again to Figure 5.10b, we can now attempt to schedule Block X containing Y (assume Block Z has also been scheduled).

In this case, we can use the approach described in Section 5.6, but modify it to take advantage of the fact that Block Y has already been scheduled. The first change is that the branch-and-bound algorithm now no longer needs to explore resource schedules for any node in Block Y. Additionally, the weights of all the variable arcs in the ILP for Block Y can now be statically fixed (both resource and reverse arcs). Therefore, the search space of both the branch-and-bound and the ILP (in terms of number of variables) have been reduced.

We can prune the ILP even further by reducing the number of total constraints. For example, there is no need to enumerate the internal cycles of Block Y, because they have already been completely scheduled and are therefore guaranteed to meet the throughput constraint. To shrink the ILP, we can prune these unnecessary cycles out before entering them in the ILP, but it would preferable in terms of runtime to not enumerate them in the first place.

Instead, we want to ignore the structure of the internals of Block Y, and consider only the paths between interface nodes of the block, effectively ignoring any internal cycles in Block Y, since they have already met the throughput constraint. To do so, we enumerate every path between each pair of interface nodes, summing up their delay and weight terms. We then introduce a new arc between the interface nodes with these attributes for every such path between the nodes. Finally, we remove any other internal nodes within the block that are not on the interface, along with any arcs connected to them, leaving a simplified graph.

The result of performing this abstraction of Block Y is illustrated in Figure 5.12, in which we have separated the forward and reverse paths for readability. The simplified forward path is shown in Figure 5.12b, while the simplified reverse path is shown in Figure 5.12c.

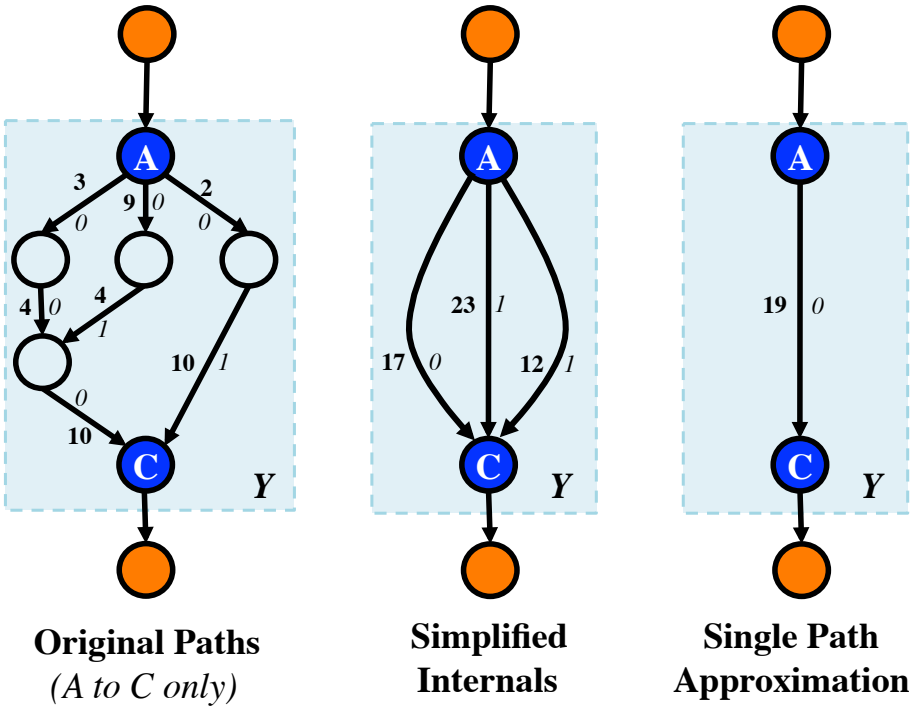

Figure 5.13: Performing a single-path approximation on Block Y (*reverse path not shown*)

5.7.3.3 Single-path Approximation

After abstracting out the internals of the block, the block's abstract model consists only of the nodes on the block's interface and their connections to each other. Because there may be a significant number of unique paths from one interface node to another, maintaining every one of these paths as an arc in the block's model at the next level of abstraction may still severely impact the runtime of the solver. Therefore, reducing the number of arcs between each pair of nodes is the next step in simplification, and is performed by *single-path approximation*.

An example of single-path approximation is shown in Figure 5.13. In Figure 5.13a we focus on the path between two interface ports A and C in Block Y from the previous example. In Figure 5.13b, we have replaced the internals of the block by arcs representing the paths from A to C. The block has been simplified to contain only

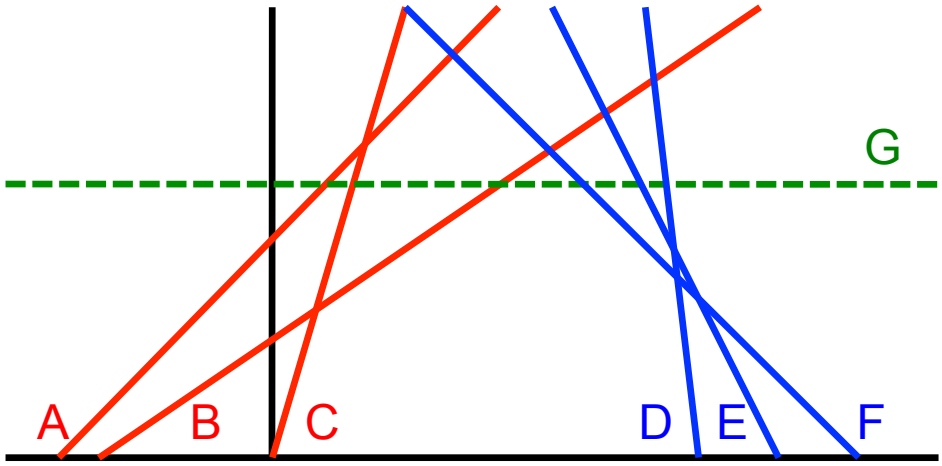

Figure 5.14: Modeling a block interface as a canopy graph

three arcs and two nodes, rather than seven arcs and three nodes. Now, we reduce the block even further: in Figure 5.13c we have replaced three arcs with a single arc that safely bounds their throughput.

To perform single path approximation, we make use of *canopy graphs* as an analysis tool (previously described in Section 2.3.2), in order to help us visualize the bounds on attainable throughput.

While our analysis approach applies to any arbitrarily large interface, for simplicity, we will now consider a block consisting of only two interface nodes (*e.g.*, one input and one output), between which there are multiple arcs with different weights and delays. Remember that each arc represents a path through the internals of the block. We will label the set of directed arcs from the input node to the output node as set $\mathcal{A}$, and the set of arcs traveling in the opposite direction $\mathcal{B}$.

Recall that in the basic canopy model, the forward slope of the "data-limited" line is determined by the inverse of the total forward latency between a begin and end stage in a pipeline. However, in this case we have *multiple* paths between the pair of interface nodes, these are defined in the set $\mathcal{A}$. As a result we will have several lines bounding the maximum achievable throughput, as shown in Figure 5.14. In the figure, these lines

are labeled $A - C$.

Furthermore, unlike the basic canopy model, in which the "data-limited" line starts at the origin, the lines corresponding to the arcs in our abstract block may have different x-intercepts in the canopy, because each path may necessarily have a different occupancy at runtime.

Modeling each arc as a throughput-limiting line in the canopy graph is straightforward: the slope and intercept of each line can be directly determined by the delay and weight of the arcs they correspond to. For the arcs in set $\mathcal{A}$, the slope of the line corresponding to each arc will merely be the reciprocal of the delay associated with the arc. The intercept of the line on the x-axis will be equal to the weight of the arc with its sign inverted. This leaves us with the set of inequalities:

$$TPUT \leq \frac{1}{\text{delay}(e)} * (occ + \text{weight}(e)) \qquad \forall e \in \mathcal{A}$$

As an example, a path with a delay of 100 and a weight of 2 has its throughput bounded by the inequality:

$$TPUT \leq \frac{1}{100} * (occ + 2)$$

Each of the arcs in set $\mathcal{A}$ will produce inequalities that coalesce to form the left-hand side of the canopy graph. The other side of the canopy graph will be bounded by the arcs traveling in the reverse direction, those in set $\mathcal{B}$. Those inequalities can be represented in a similar fashion:

$$TPUT \leq -\frac{1}{\text{delay}(e)} * (occ - \text{weight}(e)) \qquad \forall e \in \mathcal{B}$$

where the slope is the negative reciprocal of the delay associated with the arc and the x-axis intercept is equal to the arc's weight. These lines are labeled $D - F$ in

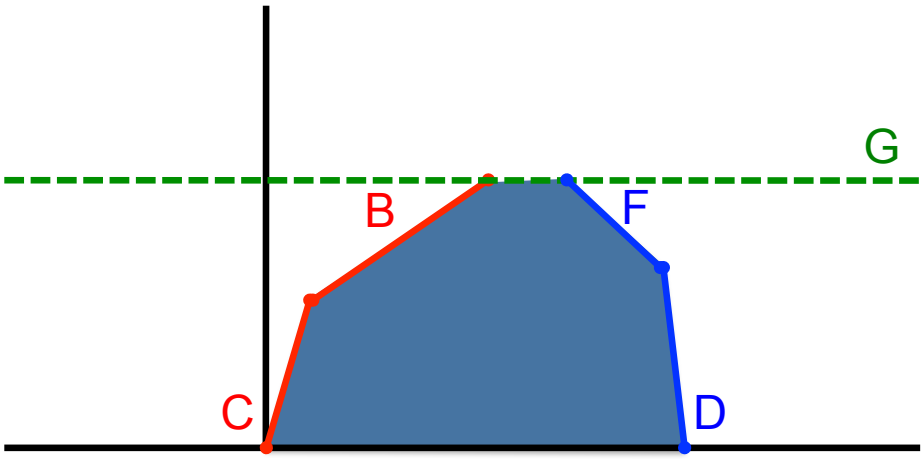

Figure 5.15: Removing redundant arcs from the canopy graph

Figure 5.14.

Finally, in basic canopy graph analysis, the graph is bounded from above by the throughput of the slowest stage. In our case, we instead bound the canopy graph by the throughput constraint set by the designer, shown as line G in Figure 5.14. While the circuit may be capable of achieving higher throughput, we can safely remove the region above it from our search space to reduce the solver's run-time. Bear in mind that G must be no higher than the throughput of any cycle in the block, because if the internals of the block could not originally meet the throughput constraint, scheduling would necessarily have failed.

After constructing the full canopy graph, we can now easily determine which arcs are limiting and which arcs are redundant in order to help simplify the graph, as shown in Figure 5.15. Here, each side of the canopy needs only two lines, rather than three, to bound its throughput; the inequalities A and E are dominated by the other constraints. Therefore, their associated arcs in our abstract block model can be removed, simplifying our block.

To simplify our model even further, we can replace the set of inequalities that limit one side of the canopy with a *single* inequality that dominates the full set. Going back

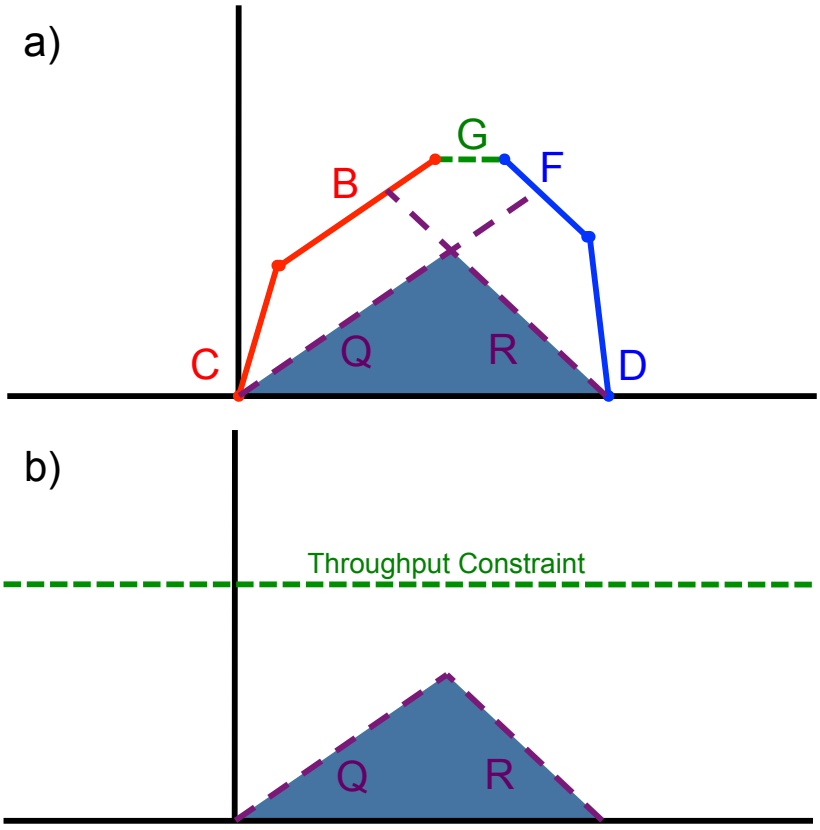

Figure 5.16: Naïve approximation of throughput constraints

to the cycle metric formulation, we want to create an arc that satisfies the following:

$$\frac{D+d}{W+w} \geq \frac{D+\text{delay}(e)}{W+\text{weight}(e)} \qquad \forall e \in \mathcal{A}$$

where d and w are the delay and weight of the dominating arc we would like to create to replace the set of arcs in $\mathcal{A}$ (the constraint is equivalent for $\mathcal{B}$). Here, D and W represent the sum of the remaining delays and weights of a cycle on which this arc is contained (these values are unknown).

In a naïve fashion, we can simply satisfy this constraint by setting d and w to the following values:

$$d = max(\text{delay}(e)) \qquad \forall e \in \mathcal{A}$$

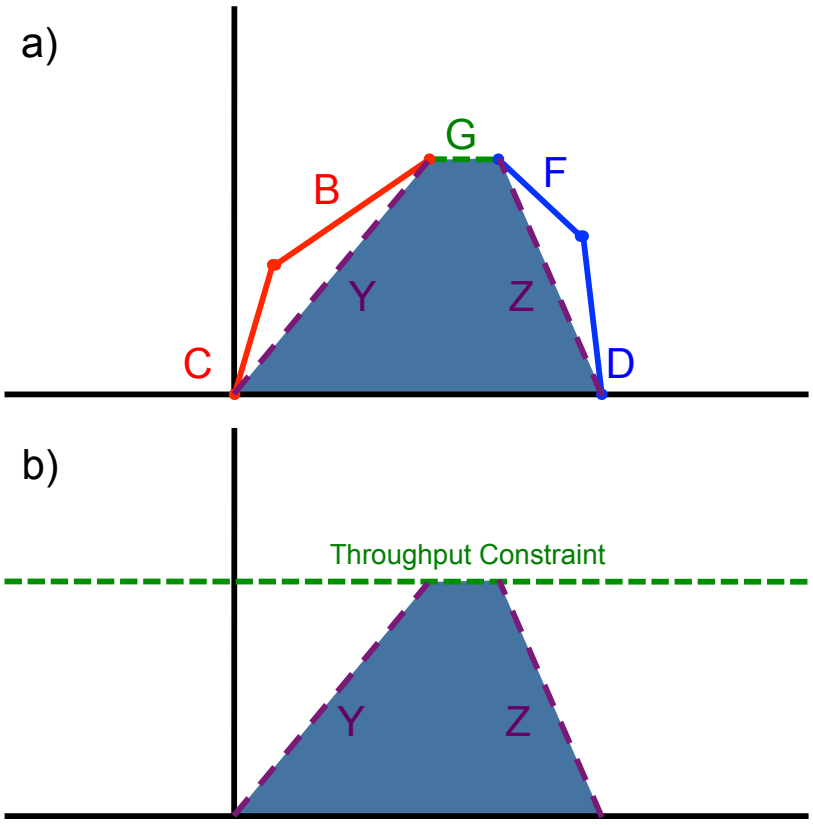

Figure 5.17: Our method's approximation of throughput constraints

$$w = min(\text{weight}(e)) \qquad \forall e \in \mathcal{A}$$

This produces the canopy graph shown in Figure 5.16a. Here, the dominating arc for $\mathcal{A}$ is represented by Q, and the dominating arc for $\mathcal{B}$ is represented by R. As you can see from Figure 5.16b, the canopy is so limited by this pair of dominating arcs that the block does not appear to meet the throughput constraint, thus scheduling will fail.

Instead, we aim replace the arcs with a better approximation, as illustrated in Figure 5.17. Conceptually, the goal is to replace this convex region with the largest trapezoid that can fit in the region while meeting the throughput constraint.

Let us begin by considering the set $\mathcal{A}$ that bounds the left-hand portion of the canopy. Here, we approximate this side of the canopy by drawing a new line that starts on the x-axis where the largest x-intercept of all the arcs in $\mathcal{A}$ is located. This

corresponds to setting the dominating arc's weight to the lowest weight of all the arcs in $\mathcal{A}$. Then, we connect this point to the line modeling the throughput constraint. The intersection at this line occurs at the greatest x-value of all the intersections of this line with constraints specified by $\mathcal{A}$. We can determine these values mathematically; we know that for arc e:

$$slope_e = 1/\text{delay}(e)$$

$$tput_e = 1/\text{delay}(e) * (x + \text{weight}(e))$$

and we want to determine maximum x-value where each arc intercepts the throughput bound:

$$x_{max} = \max\left(\frac{\text{delay}(e)}{CT} - \text{weight}(e)\right) \qquad \forall e \in \mathcal{A}$$

where $1/CT$ is the throughput constraint set by the designer. Since we know the x-intercept of the line by determining w:

$$w = min(\text{weight}(e)) \qquad \forall e \in \mathcal{A}$$

We can determine d to be:

$$d = CT * (w + x_{max})$$

This pair of values (w, d) will determine the attributes of the dominating arc for the set $\mathcal{A}$. This procedure can be performed equivalently for the set $\mathcal{B}$.

Unlike the naïve approach, this approach ensures that the throughput constraint can still be met at some occupancy, as shown in Figure 5.17b.

Thus, the full set of arcs in $\mathcal{A}$ and the full set of arcs in $\mathcal{B}$ can be approximated by a single dominating arc; all other arcs in the set can be removed. Therefore, each pair of nodes will have only two arcs between them, as in Figure 5.13c, significantly simplifying the block interface.

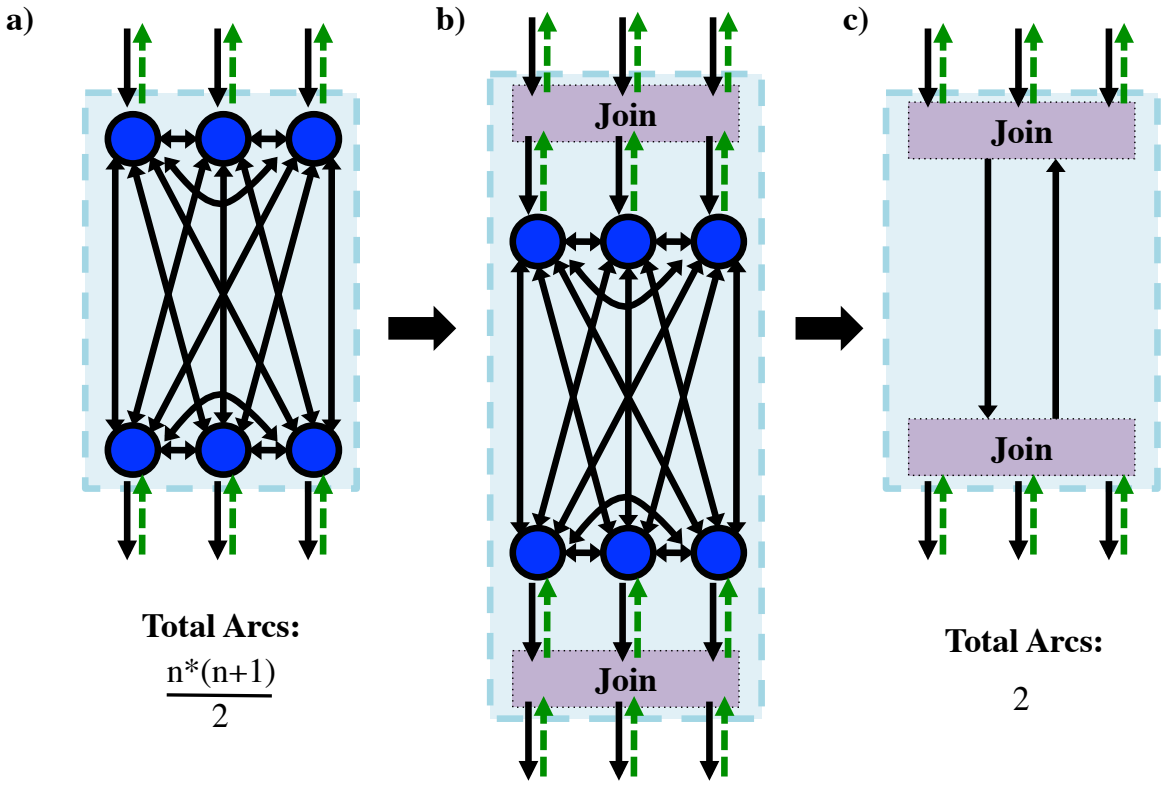

Figure 5.18: Converting a block to a two-port representation

5.7.3.4 Two-port Specifications

Even after performing a single-path approximation for each pair of nodes on the interface, the number of arcs going through the interface may still be significant. The total number of arcs in the block grows with the number of interface nodes on the order of $\Theta(n^2)$ when every interface node is connected, as shown in Figure 5.18a.

Therefore, an additional transformation can be performed on a block to further simplify its abstract model: we can reduce the interface to only two ports. This transformation can be done up-front by the designer modifying the input specification to match the two-port restriction, but can be automated. By modifying the original specification such that a single input and output node will exist on the interface, one can further simplify the internals of the block down to a single pair of arcs: one forward

and one reverse arc.

The two-port transformation merely requires a synchronization of inputs to the block into a single node, and a synchronization of outputs to a single node, as illustrated in Figure 5.18b. In practice, these synchronizations are performed by join operations on the input and output sides of a block, in which a full tuple of values are combined at once into one large buffer. When the data is transferred to another block, it is then split back into individual buffers allowing the appropriate data to flow through the internals of the next block in a decoupled fashion. Once the synchronization has occurred on the input and output of a block, the block's abstract model can be simplified into a single pair of nodes and a single pair of arcs, as in Figure 5.18c.

While the two-port specification style can significantly reduce the complexity of the hierarchical approach, the downside is increased area cost. Additional buffering will generally become necessary in a block due to the introduced synchronization blocks.

It is interesting to note that the two-port specification transformation and scheduling procedure begins to resemble the data-driven style of Chapter 3, particularly as the block size begins to decrease and there is less opportunity for resource sharing.

5.7.3.5 Handling Conditionals and Loops

The proposed hierarchical block-abstraction method not only improves run-time over the optimal approach, it also allows for a straightforward modeling of conditionals and loops as individual blocks in the hierarchy. Here I will describe how conditionals and loops are handled hierarchically by our approach.

Conditionals. Two methods are employed for handling conditionals: conditional assignment and early evaluation. The former method executes both branches of the conditional in parallel and selects the appropriate value at the output. The latter conditionally selects a branch to forward data (based on a boolean value), and thus

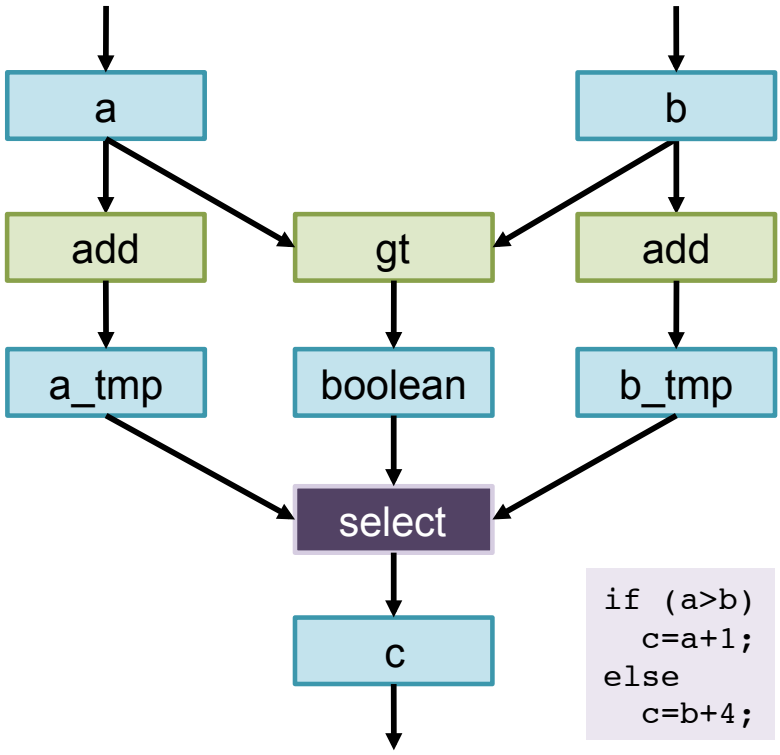

Figure 5.19: Conditional assignment

does not perform unnecessary computation.

Figure 5.19 illustrates conditional assignment. Here the notion of control-level choice has been replaced by computation. In this example, the expressions $a+1$ and $b+4$ are computed in parallel (or sequentially if the adder is shared), then the proper output is chosen by the *select* element. The select element operates repeatedly as follows:

1. Wait for all three inputs to be available: $input_1$, $input_2$ and sel.

2. When all three values are available, select the appropriate value based on the value of sel and feed it through its output port, along with a request.

3. Wait for an acknowledgement from the output channel to ensure that the data was received.

4. Acknowledge all inputs.

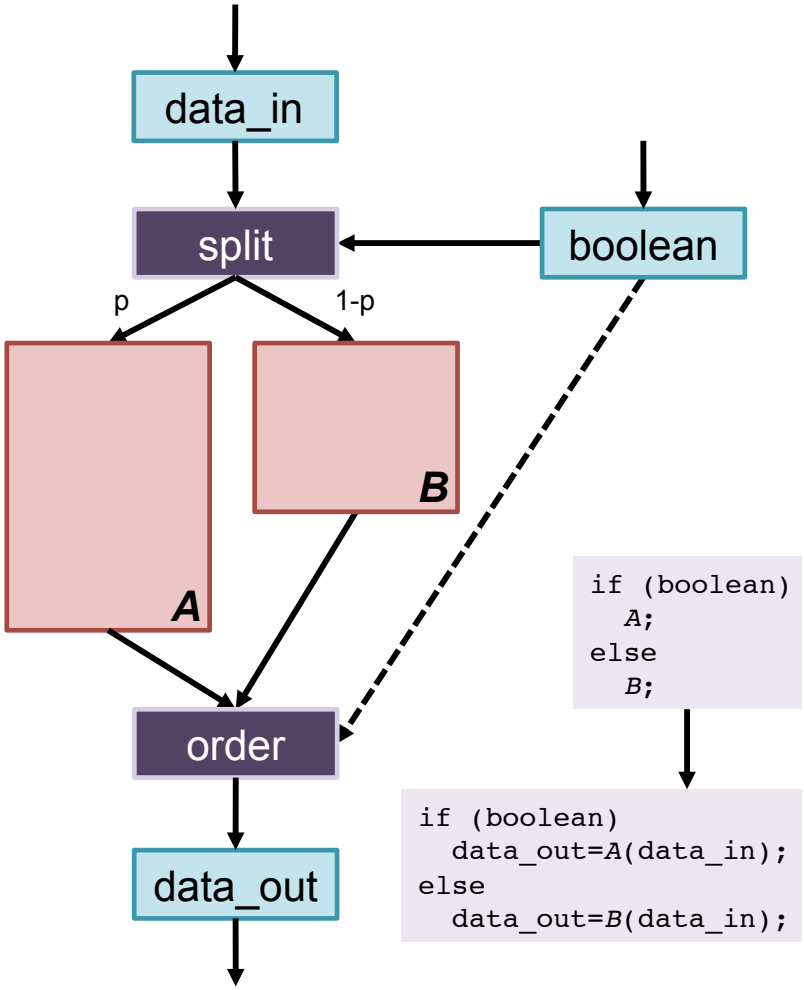

Figure 5.20: Early evaluation

The benefit of the conditional assignment route is it essentially removes control-driven choice, allowing us to easily model conditionals using our ILP method. However, this approach does have significant drawbacks: (i) increased energy consumption, (ii) worst-case latency, and (iii) increased area. For example, while operations in both paths are mutually exclusive and could share an adder with no additional cost, conditional assignment requires *both* to be computed, therefore sharing the adder would incur a latency penalty for the block, while allocating an additional adder would incur an area penalty.

The second method for handling conditionals is illustrated in Figure 5.20. Here we have converted the original source into a two-port specification style for simplicity.

In this example, the value of the boolean is computed prior to execution of either branch. Then, the appropriate data is forwarded to one of the branches using a *split* element. The appropriate branch then performs computation on the data. After performing computation, the block forwards the output data to the *order* element that operates as follows:

1. Wait for *sel* value to indicate which port to read from.
2. Wait for value on port indicated by *sel*.
3. Transmit the value through its output port, along with a request.
4. Wait for an acknowledgement from the output channel to ensure that the data was received.
5. Acknowledge both inputs.

The *order* element therefore selects the appropriate value based on the value of the conditional and forwards it on its output port. Here, we must keep in mind that the value of the boolean must be appropriately slack-matched on its way to the *order* element (indicated by the dashed arc).

There are two caveats with this approach: (i) it introduces true choice in the model, which is not handled by the analysis in Section 5.6, (ii) it can incur additional latency because the boolean must be computed prior to execution of the branches. However, this approach has its merits: lower energy consumption and the potential for better average case performance.

In order to handle the problem of choice in the performance analysis, we can turn to the method described by (Gill, 2010). In this work, the author illustrated one possible

method to model the performance of a conditional block is by multiplying the computed throughput for each block by the inverse of the probability of each path being taken.

In a normal block where the probability of being taken is 1, the block must always be able to support the full throughput constraint specified by the designer. However, if the probability of a block being taken is 0.5, it only needs to support *half* the rate.

Therefore, when we schedule the branch, we only require that the branch meet a throughput constraint of $p*T$, where p is a pre-computed probability of the branch being taken, and T is the throughput constraint set by the designer. When we replace the block with a pair of single-path approximation arcs, the delay on each arc is multiplied by p in order to appropriately model its throughput.

There are a few limitations to this model: (i) the conditional probabilities must be pre-computed, (ii) each conditional probability is assumed to be independent, (iii) the interface elements are assumed to be infinitely fast in (Gill, 2010), and (iv) clusters of items selecting the same branch may have a significant detrimental impact on throughput.

One final item to note is that during scheduling, we may consider both branches as separate blocks, and schedule each with a disjoint set of resources. However, if the blocks are sufficiently small, they can be scheduled simultaneously with the same set of resources, allowing mutually exclusive operations to share the same resource.

Loops. Let us now consider how loops are handled in the hierarchical approach. Let us begin with our proposed architecture, as shown in Figure 5.21. In this example we assume the throughput constraint is so tight that each operation has received its own dedicated function unit, but no additional buffers have been added.

Figure 5.21a shows the body of a Fibonacci loop without any additional control structure. Here, data enters at the top and is routed through the appropriate function units, trickling down until the final values are produced for k, a, and b through a

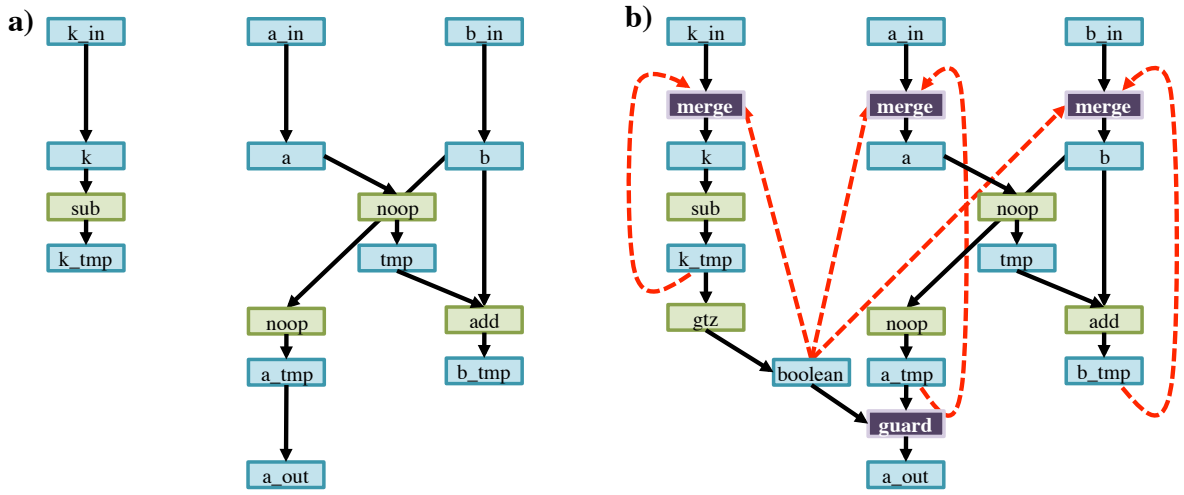

Figure 5.21: a) Loop body without control elements and b) loop body with control elements inserted

single iteration. This original implementation does not include any choice points, and is therefore easily modeled.

Now, we must incorporate control to feed data back in order to allow multiple iterations. Figure 5.21b shows how data is routed back into the loop using *merge* elements. Here, the input values *k_in*, *a_in*, and *b_in* are fed into *merge* elements that will select either these new input values (*i.e.*, a new problem instance), or the values produced at the end of the loop (an previous problem instance), *k_tmp*, *a_tmp*, and *b_tmp*, respectively. The appropriate data is selected based on the value of the boolean clause $k\_tmp > 0$.

The operation of the *merge* element repeats as follows:

1. Wait for all three inputs to be available: var_{in}, which is the input to the loop, var_{loop}, which is the value of a variable after executing the loop body, and *sel*, which decides which value to select for input.

2. When all three values are available, select the appropriate value and feed it through its output port, sending out a request.

3. Wait for an acknowledgement from the output channel to ensure that the data was received.

4. If the item selected was var_{in}, an acknowledgement occurs on the var_{in} channel.

5. In all cases, an acknowledgement occurs on the var_{loop} channel, regardless of which data was selected.

On the opposite end of the loop, data is being selected for output via a *guard* element. The operation of the *guard* element repeats as follows:

1. Wait for both inputs to be available: var_{loop}, which is the value of a variable after executing the loop body, and *kill*, which decides whether to allow the value to pass through the loop or be consumed.

2. When both values are available, the *guard* block consults the value of *kill*. If *kill* is false, a request is sent on the output channel, allowing the value to pass through. Otherwise, no request is sent on the output channel.

3. If a request was transmitted, wait for an acknowledgement from the output channel to ensure that the data was received.

4. All input signals are acknowledged.

By constructing the *merge* and *guard* elements in this fashion, we can essentially remove the notion of choice from inside loop body itself. The choice points occur on the outside of the loop: either external data is read by the *merge* element or it waits, and either data is produced by the *guard* to the external block or the external block must wait. Therefore, in all cases, the *internals* of the loop operate without choice.

This allows us to model the loop much like we did a general block, with one modification. Because a token must repeat the computation in a loop body multiple times

before exiting the block, the throughput constraint on the loop becomes tighter by a factor of the iteration count of the loop. As an example, if the throughput constraint of the full specification is $T = 100$, and the iteration count of the loop is 10, the loop must be able to produce a throughput of 10x, or $T_L = 1000$. Therefore, every cycle in the loop must meet the tighter constraint. This modification allows us to model the interface to the loop where the choice points are now occurring, since data is being consumed every 10 iterations.

When modeling the full loop as a block, we can turn once again to canopy graphs for analysis. Since the block's forward latency is multiplied by the iteration count of the loop, the slope of each forward path's line will become shallower. Therefore, the single-path approximation arc that represents the block must have its delay multiplied by the iteration count. However, on the reverse path, the reverse latency of only one iteration need be considered. We can reduce the reverse latency further if a global controller is managing the token count of the loop; then, the reverse delay can be modeled as the delay from the controller to the loop's input port.

One additional optimization is performed by the hierarchical method: automatic selection of token count. In Figure 5.21b, the dashed red arcs highlight feedback data paths that are initialized with a single token representing the loop's occupancy. The ILP approach presented in Section 5.6.3 has been modified to allow these feedback data arcs to become variable, and instead fixes the associated reverse arcs to a static value of 0. In the ILP, each of these data arcs are required to have the same value, otherwise data will be synchronized erroneously across problem instances. In this way, we can allow the ILP to select the preferred token count for the loop.

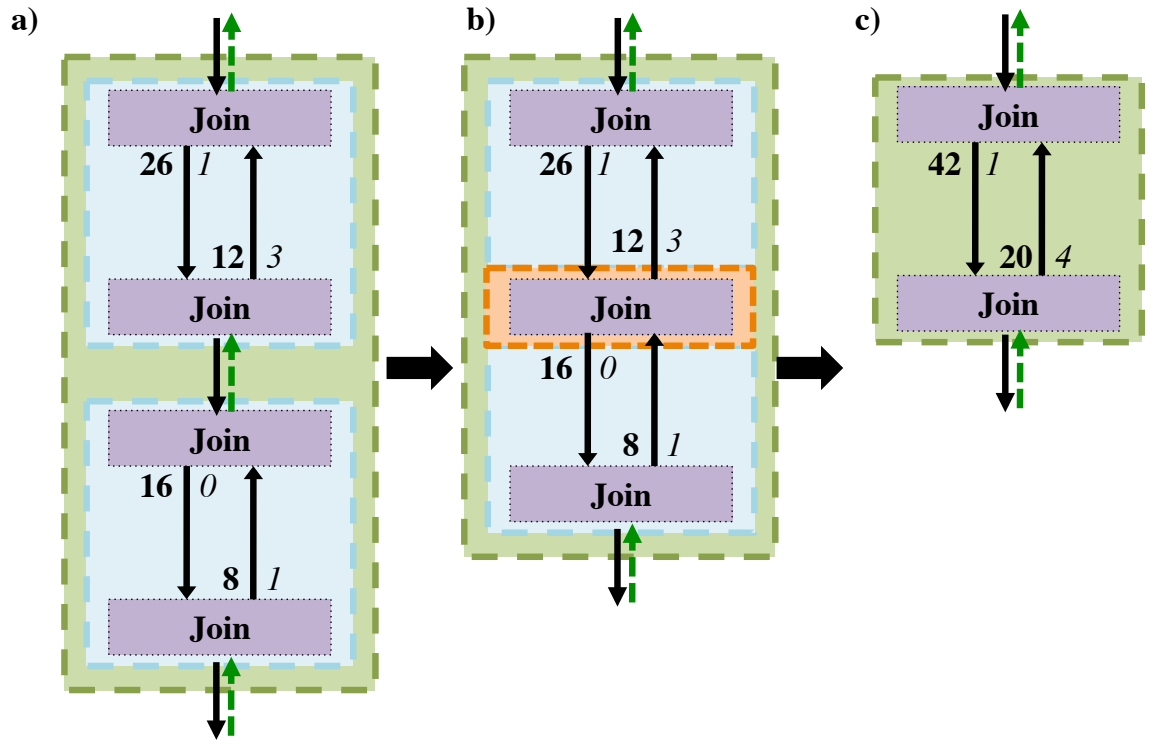

Figure 5.22: Composing sequential blocks into a single block

5.7.4 Hierarchical Composition

Once a block is scheduled and has been replaced with a simplified model, we can then schedule it in the next higher level in the hierarchy. In this subsection, we illustrate how blocks are combined when they are constructed sequentially, in parallel, or when one is nested inside another. In this section, we will use examples two-port specification for simplicity of explanation.

5.7.4.1 Example: Sequential Blocks

Let us start with the example shown in Figure 5.22a. Here we have two blocks that have already been scheduled and simplified. Each block now contains only a pair of arcs with unique weights and delays between two join buffers.

The first key observation is that since the data produced at the output of the first

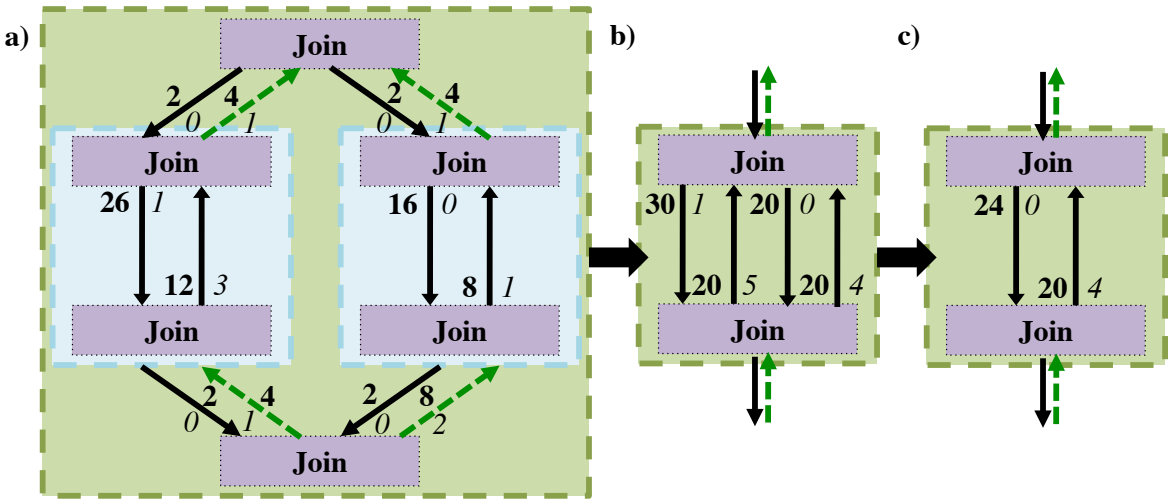

Figure 5.23: Composing parallel blocks into a single block

block and consumed at the input of the second block are the same, we can optimize by merging the two join buffers into a single buffer, as shown in Figure 5.22b.

Next, we aim to create a single block that represents the sequential behavior of the two blocks. In this case, the process is simple: we sum up the weights and delays on the forward path from input to output to produce a representative forward arc, and sum up the weights and delays on the reverse path to produce a representative reverse arc, as shown in Figure 5.22c.

It is worth noting that the throughput bounds provided by each individual block no longer exist in the model, but can be represented as a self loop with the appropriate delay and weight. However, had either block failed to meet the throughput constraint, the scheduling procedure would have failed prior to joining the blocks, and therefore these self-loops are unnecessary for scheduling.

5.7.4.2 Example: Parallel Blocks

Combination of parallel blocks is illustrated in Figure 5.23, using the same pair of blocks from Figure 5.22.

In this example, we have already scheduled the two parallel blocks, and have introduced a pair of synchronizing joins to model a two-port specification of their encapsulating block. The buffering requirements between the joining buffers and each block have already been determined and are shown in Figure 5.23a.

Observe in this example that the block on the right requires an additional buffer in order to meet the throughput constraint due to the slack mismatch between the two blocks. This situation can be contrasted with that of sequential composition, in which no additional buffers are needed for slack matching when combining the blocks.

In Figure 5.23b, we have removed the internals of the encapsulating block by enumerating all the paths between its interface nodes, leaving two arcs in either direction. We therefore perform a single path approximation to produce the final block, shown in Figure 5.23c.

5.7.4.3 Nested Blocks

The example of parallel composition illustrated in Figure 5.23 is actually one of nested composition: in this case there are two separate blocks nested inside an encapsulating block with its own buffering. A more general encapsulating block is capable of having its own computation nodes in addition to any nested blocks.

Arbitrary levels of nesting are handled by our approach, the only caveat being that blocks nested in loops must follow the rules specified in Subsection 5.7.3.5. In particular, the throughput of the internals of a loop must meet the throughput constraint *multiplied by the iteration count of the loop*. Thus, loops nested within loops will see this factor become the *product* of their iteration counts.

5.7.5 Hierarchical Area-Minimization

Figure 5.24 gives a basic overview of the proposed hierarchical area-minimization

```
procedure hierarchicallySchedule(block, parent){
   for each child_block in block{
      hierarchicallySchedule(child_block, block)
   }
   scheduleOptimally(block)
   compute interface for block
   for each pair (A,B) of interface nodes{
      compute all paths from A to B
      add single path approximation arc from A to B
      compute all paths from B to A
      add single path approximation arc from B to A
   }
   remove block internals
   replace block in parent with simplified representation
}
```

Figure 5.24: Basic hierarchical area-minimization algorithm

algorithm. The algorithm begins with the top-most block, and descends into its children depth-first until a block with no children is encountered (a "leaf" block). The block is scheduled optimally using the approach outlined in Section 5.6 in order to produce the minimum area implementation for the block.

Next, the solver computes the interface nodes for the block. For each pair of nodes on the interface, the solver determines the delay and weight of each path that goes between the nodes. Next, a single-path approximation is performed to replace the set of arcs going from one node to the other with a single arc. The same approximation task occurs for all the paths that travel in the opposite direction as well.

After all the single-path approximation arcs are generated between each interface node, the solver then removes all other internal nodes and arcs from the block. The new simplified model is then inserted in place of the original block for scheduling in its parent block.

If the parent block contains children that have yet to be scheduled, these blocks are then scheduled and simplified using the same method. Eventually the parent block

itself will be ready for scheduling; this process continues up the chain until the top level block is scheduled.

5.8 Results

This section illustrates the proposed method's capability to find an area-minimized solution under a variety of constraints. Utilizing benchmarks described in Section 4.6, the solver was run on a total of 21 test cases and compared to the previous results.

5.8.1 Setup

In experimentation, seven different benchmark DFGs were used for analysis. For each test case, we set constraints for cycle time and optimized for minimum area. We provided the same library of functional units to all benchmarks; Table 5.1 shows the parameters used in experimentation in the optimal approach, while Table 5.2 shows a different set of parameters used for the hierarchical approach. Buffer delays were modeled as 1 in both the forward and reverse directions.

Section 4.6 provides a description of six of the benchmarks. One additional benchmark was added, FIB, which computes the Fibonacci number of a given input. The TEA and FIB benchmarks both contained loop structures that were unrolled in a set of experiments for the hierarchical method.

The approach was implemented in Java using standard libraries. Benchmarks were tested on a Macbook Pro with a 2.8 GHz Intel Core 2 Duo processor and 4GB of RAM on JVM 1.6. Run-times are shown in seconds or milliseconds.

5.8.2 Discussion of Results

Table 5.3 shows the experimental results for the optimal approach. The first two columns list the benchmark and throughput constraint respectively. For the single-token solver in Chapter 4, this throughput constraint was equal to the latency constraint, as only single-token schedules were produced. The next column shows the logic area a single token schedule could produce using the single-token approach (this method ignores buffer area). The next three columns show the results of the multi-token approach, including logic, buffer, and total area. The final column shows the run-time in seconds.

The results shown in the table clearly show that a multi-token approach is superior to a single-token approach in terms of function unit area. In all test cases, the logic area was less than or equal to that of the single-token solver. In fact, in most cases, the *total* area (including buffering) was lower than the function unit area of the single-token solver. Given the available numbers, one can conclude that the multi-token approach provides a lower total area solution in terms of resource and buffer area in each case with the possible exception of TEA.

Further, there are several instances where the single-token approach *cannot* meet the throughput constraint, even if infinite resources were provided.

Now let us consider the effect of the throughput constraint on buffer area: when the throughput constraints become very tight we begin to see an increase in buffer area because more pipelining is needed. For example, consider the first two test cases of COS, where the buffer area is 52 under a tight throughput constraint of 16, but when the throughput constraint is relaxed to 32, the buffer area reduces to 48.

Finally, the runtime of this approach is illustrated in the last column. For the single-token approach, the run-time was under 5 seconds in each test case. In the multi-token test cases, the runtime was under 10 seconds in all but four test cases. Three of those

successfully completed in under an hour, while one test case did not complete within 8 hours and was manually terminated.

Now, let us consider a different set of results, shown in Table 5.4. The results in this table run exclusively on the FIB benchmark, a specification with a loop construct. The FIB benchmark had its main loop unrolled eight times, and in this table, the effect of average iteration count and cycle time constraint on total area is shown.

In this table, we see that the total area of a circuit increases as the cycle-time constraint becomes tighter. However, this ratio is not one-to-one. For example, halving the cycle-time from 200 to 100 for the 8-iteration case only results in 1.23x more area but provides 2x the throughput. Another observation that can be drawn from this table is that increasing the iteration count of a loop causes a circuit to consume additional area. This is illustrated by comparing the iteration counts at cycle-time constraint of 500; a loop with an iteration count of 16 requires about half the resources needed for a loop with an iteration count of 64.

Let us now consider the effect of unroll count and block size on the area of a circuit produced by our tool, as shown in Table 5.5. By unrolling a specification, the total area consumed increases, due to the fact that additional storage is needed in the pipeline. The total area seems to roughly double when increasing unrolling by a factor of 2x. By increasing the size of partitioned blocks, we see that the area cost is generally lower as the solution trends closer to the optimal value. One anomaly exists between TEA unrolled 4x with 12 versus 16-node blocks. This anomaly is due to the fact that the automated partitioning scheme does not optimally generate blocks; a better partitioning would likely produce a lower area circuit.

Finally, consider the run-time of the hierarchical method as a function of benchmark size and block size. When approximately doubling the number of nodes from 94 to 174, the execution time of the solver increased by 4-25x, dependent on block size.

Table 5.1: Functional unit parameters

Function Unit	Area (unit)	Latency (unit)
Add	8	8
Subtract	8	8
Multiply	48	9
Shift/Logical	8	8
Buffer	2	1 / 1

Table 5.2: Functional unit parameters

Function Unit	Area (unit)	Latency (unit)
Add	8	32
Subtract	8	32
Multiply	256	64
Shift/Logical	8	32
Buffer	2	1 / 1
Merge	3	2
Join	3	2
Guard	1	2

However, when approximately doubling the number of nodes again to 334, the execution time of the solver only increased by 3.5-4.5x. The key observation to take away from this example is that the hierarchical scheduling approach does not exhibit exponential increases in run-time based on node size, at least for this example.

The effect of block size on run-time is more significant. A 1.3-4x increase in run-time is shown when increasing the blocks size from 8 to 12. Increasing the block size from 12 to 16 shows a 2.5-7x increase in run-time. Based on this observation, maintaining smaller block sizes can make a significant impact on run-time, albeit at the cost of optimality.

Table 5.3: Run-time and results for throughput-constrained area-minimization

Benchmark	Cycle Time Constraint	Area (unit) 1-● Logic	Multi-● Logic	Multi-● Buffers	Multi-● Total	Run-time (S)
ODE	9	-	272	30	302	0.2
ODE	34	160	112	18	130	0.2
ODE	50	112	64	18	82	0.2
DP8	9	-	440	48	488	0.3
DP8	27	-	168	32	200	0.4
DP8	35	416	160	32	192	0.4
DP8	50	208	112	32	144	0.7
DP8	90	104	56	32	88	0.7
COS	16	-	800	52	852	3.8
COS	32	-	304	48	352	355
COS	75	208	104	48	152	1908
7TH	9	-	832	88	920	0.7
7TH	16	-	776	58	834	1.1
7TH	45	-	168	58	226	51
7TH	90	168	112	58	170	493
ELP	9	-	592	202	794	2.5
ELP	115	168	-	-	-	>8hr
TEA	32	-	40	36	76	7.8
TEA	40	48	32	36	68	4.1
TEA	43	32	32	34	66	5.9

5.9 Conclusion

In this chapter I described an optimal method for generating multi-token schedules for performing resource sharing in a pipelined system. I then illustrated how these schedules could be modeled graphically, described an architecture to implement these schedules, and developed an algorithm to minimize overall logic and buffer area while meeting a throughput constraint using a branch and bound approach. Finally, I proposed a hierarchical method for dealing with large, real-world examples.

The focus of this chapter has been specifically on the trade-off between performance and area. In the next chapter, a new set of constraints will be considered: power and energy.

Table 5.4: Effect of cycle-time constraint and iteration count on implementation area

Benchmark	Iteration Count	Cycle-Time Constraint	Block size: 32 nodes Area	Block size: 32 nodes Time (ms)
FIB	8	500	128	1100
FIB	8	200	168	580
FIB	8	100	208	880
FIB	8	50	344	700
FIB	16	500	152	1160
FIB	16	200	208	560
FIB	16	100	344	670
FIB	32	500	208	930
FIB	32	200	344	530
FIB	64	500	320	780

Table 5.5: Effect of unroll count and block size on implementation area and tool performance for TEA benchmark

Unroll Count	Node Count	Block size: 8 nodes Area	8 nodes Time (s)	12 nodes Area	12 nodes Time (s)	16 nodes Area	16 nodes Time (s)
4	94	2797	1.8	2291	2.3	2389	5.9
8	174	5927	6.4	5075	23.4	3745	142
16	334	11163	21.3	9173	95	7131	654

Chapter 6

Energy and Power Considerations

The previous chapters of this thesis have focused primarily on circuit area and performance, and have largely ignored two other aspects of emerging interest in high-level synthesis: power and energy. Particularly as consumer demand trends more and more towards mobile devices, battery life and peak power are significant constraints that need to be considered in the design process. Therefore, this chapter proposes several additions to consider power and energy in the synthesis process.

6.1 Introduction

This chapter presents two contributions in the area of energy and power in high-level synthesis. The first contribution is to incorporate modifications to the scheduling strategy given in Chapter 4 in order to make it energy and power-aware. These modifications include two additional constraints, one to constrain the maximum power consumption for a specified schedule (*i.e.*, reducing the maximum instantaneous power draw during the full runtime of a schedule), the other to constrain the maximum energy consumption of a full schedule. Another key modification is to incorporate *energy-minimization* as an alternate target of our scheduling approach, in addition to time and area-minimization.

The second significant contribution, orthogonal to that above, is to detect under-utilization of resources in a schedule (*i.e.*, *scheduling slack*) and dynamically scale down the voltage supply in order to trade off idle time for significant savings in energy and power (without lengthening the total execution time of the schedule). In the least intrusive fashion, this optimization can be done as a post-scheduling step by modifying the binding of operators to resources as to provide the greatest opportunity for voltage scaling.

The first half of this chapter focuses on the first contribution: extending the branch-and-bound strategy in Chapter 4 in two ways: (*i*) incorporating of power and energy constraints, and (*ii*) introducing an algorithm for energy-minimization. As the branch-and-bound method relies heavily on pruning for reducing the search space to improve performance, several additional bounds have been developed that take power and energy into account. In addition, new optimizations have been developed for exploring the search space in order to more quickly approximate the best solution.

This second half of this chapter focuses on the second contribution: an approach for energy reduction in an asynchronous shared-resource system by exploiting slack in the schedule. In particular, the approach accepts a high-level specification for which resource allocation, scheduling and binding have already been performed, along with an upper bound on the latency of the schedule. Given this input, the proposed method then determines available slack in the schedule, both along the critical path (*e.g.*, if critical path length is shorter than the given latency bound) and along non-critical paths. Finally, an assignment of supply voltages for the resources is determined such that energy consumption is minimized, while still satisfying the upper bound on the schedule's latency. I also address the practical problem of imposing a limit on the number of distinct voltage sources used by the implementation.

The experimental results are quite promising. The performance of the energy-

minimization approach is on par with that of the highly-optimized area and time-minimization approaches. The overhead of implementing each additional bound was minimal; in the end, each test case completed in under 5 seconds. For the proposed voltage scaling approach, the heuristic methods found a solution with an energy cost within 5% of the energy cost of the optimal solution for all but one test case. The runtime of the heuristic solutions was negligible (typically in milliseconds), and several orders of magnitude faster (from 20x to 10,000x faster) than the optimal one that searched the complete search space.

The rest of this chapter is organized as follows. First, Section 6.2 discusses relevant prior work. Next, Section 6.4 presents modifications to the scheduling approach presented in Chapter 4 to incorporate energy and power. Then, Section 6.4 presents several approaches to voltage scaling, including the optimal geometric programming formulation, and two faster heuristic ones. Section 6.5 presents experimental results, and we will conclude with Section 6.6.

6.2 Background

6.2.1 Energy and Power

Let us first start by distinguishing the terms energy and power. In this chapter, the total "energy" of a schedule refers to the total amount of energy consumed from start to finish, irrespective of how that energy is distributed. The term "power" refers to the instantaneous rate of draw of energy at a point in time.

As an example, let us consider two implementations that perform the same computation but have different schedules and bindings. Schedule X completes computation with latency $L = 100$ and total energy consumption $E = 500$, while schedule Y has the attributes $L = 50$ and $E = 400$. Here, schedule X consumes more energy than

schedule Y. However, if we assume the energy consumption is uniformly distributed over time, schedule X has a power consumption of $P = \frac{500}{100} = 5$ while schedule Y has a power consumption of $P = \frac{400}{50} = 8$. Thus, schedule X is lower power, but higher energy than schedule Y.

In practice, energy consumption will not be uniformly distributed over time for a full schedule. Instead, a designer will be concerned with the peak instantaneous power draw, since it defines the requirements on the source providing power to the circuit.

Therefore, the proposed approach will parametrize each operation with the attributes latency, energy consumption, and peak power for each function unit on which it can operate.

6.2.2 Previous Work

For performing scheduling and allocation under area, time, or resource constraints, several different high-level synthesis techniques have been discussed in Chapter 4. The majority of these approaches do not consider power or energy as part of the scheduling processes. However, heuristic solutions do exist that take into power, latency, and resource constraints into consideration in the scheduling process: (Manzak and Chakrabarti, 2002; Lin et al., 1997; Johnson and Roy, 1997).

Many other approaches address the task of energy minimization by static or dynamic scaling of voltages, but are often developed for synchronous uni-processor or uniform multiprocessor systems. However, most of these approaches are on a coarse granularity scale, considering several independent multi-instruction tasks operating on uniform function units (processors). By contrast, my approach is applied at a very fine granularity, on the order of individual operations. As a result, my approach is capable of handling heterogeneous function units, and must carefully consider operation dependencies at this granularity.

One recent approach is described in (Chen et al., 2006), where the authors provide an optimal solution to voltage assignment and resource binding for two voltage levels. However, my approach is capable of handling an arbitrary number of voltage levels, limiting the count only when specified by the designer.

In contrast to these approaches, my energy and power-aware scheduling work targets exact solutions (as defined in Section 6.1) to the scheduling and allocation process, given area, time, power, and energy constraints, for both the asynchronous and synchronous domains. I target a robust solution space by incorporating many-to-many operation to functional unit mappings, functional unit energy and power consumption, energy and power constraints, and minimization for energy in our approach. Then, since even a energy-minimized schedule may have further opportunities for energy reduction by voltage scaling, my approach further refines these low-energy solutions by performing voltage scaling as a post-processing step.

6.3 Incorporating Energy and Power Constraints in Scheduling

In this section we will address the problem of incorporating power and energy into the scheduling approach presented in Chapter 4. Since the resource-constrained time-minimization algorithm is at the core of the other minimization methods, I will first show how this method is modified to take into account specified upper bounds on peak power and total energy consumption. Next, I will explain how the allocation search space is modified. Finally, I describe an alternate scheduling strategy: energy-minimization.

6.3.1 Resource-constrained time-minimization

6.3.1.1 Constraints and targets

Recall that resource-constrained time-minimization aims to find the lowest latency schedule for a DFG given a specific resource allocation. In the energy and power-aware approach, the additional constraints of *total energy consumption* and *peak power* can be set to restrict possible solutions. This type of search may be employed by a designer working with a static architecture aiming to find the highest performance solution under the additional constraints of energy consumption and peak power.

6.3.1.2 Algorithm

The core algorithm for resource-constrained time-minimization is given in Section 4.4, but its most basic behavior will be briefly described here.

The initial step is to generate a worst case maximum time by summing up the worst case latencies for each operation. This will serve as an initial "best solution found" while exploring the search space.

The main procedure for exploring the search space is then initiated: a recursive method that expands the DAG in a depth-first fashion. As each node is reached, starting with the $root+$ node, all possible children are enumerated. Once a $sink-$ node is reached, the latency of that path is compared to the best time so far; if it is lower, this is the new best schedule. Along the way, several pruning optimizations are performed, such as tight, safe bounds, redundancy removal, and hashing.

Key additions. One of the core aspects of this algorithm is the selection of child nodes for each node in the DAG. In the original algorithm, only four aspects were considered: (i) dependence restrictions, (ii) availability of resources, (iii) binding of operation to functional unit type, and (iv) lexicographical ordering (to remove redundancy).

However, an energy and power-aware approach includes on top of those constraints: (i) free power remaining, and (ii) free energy remaining. These considerations serve to further reduce the search space.

The amount of free power and energy remaining are calculated by analyzing the schedule produced up to the current node in the DAG. As each binding of operation to functional unit is parametrized by an energy and power cost, the sum of all the energy costs of already scheduled nodes is subtracted from the total energy constraint to produce a "free-energy-remaining" term. The sum of all the power costs of currently executing nodes plus the leakage power of idle components is then subtracted from the power constraint to produce a "free-power-remaining" term. These two terms help trim the size of the DAG.

As a simple example, consider the case where two potential operations are available for execution, an add and a multiply. Assume the add and multiply can operate on an ALU as well as their own dedicated functional unit types. Here, let us assume that an add consumes a power amount $P_{+} = 4$ on a dedicated adder, and $P_{+_{ALU}} = 6$ on an ALU. The multiply consumes $P_{*} = 16$ on a dedicated multiplier and $P_{*_{ALU}} = 20$ on an ALU.

In this case, four total *node+* nodes are initially possible as children of the current node. However, as the selection of children is dependent on the other constraints (*i.e.*, power/energy), multiple children may be pruned. For example, assume the power budget is $P_{MAX} = 50$, but the currently budgeted draw is $P_B = 40$. In this scenario, only enough power remains to execute an add, so the multiply children are pruned from the search space as direct children of this node. They must therefore must be executed at a future time, when enough free power becomes available. This constraint will leave only the *node+* nodes corresponding to the add operating on the ALU or a dedicated adder.

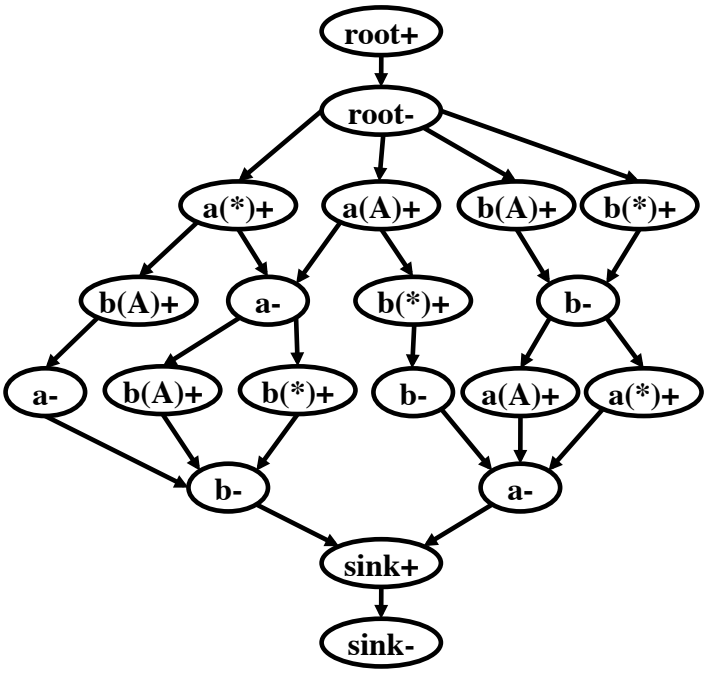

Figure 6.1: Full expansion of the DAG for a two-operation DFG with one ALU and one multiplier

Further, should the power budget be restricted further such that $P_{MAX} = 45$, then only the dedicated adder will be considered for the add to be executed (the ALU option is now pruned), leaving only one *node+* node possible. If the power budget is restricted even further, *e.g.* $P_{MAX} = 40$, then no *node+* nodes can be selected; a *node−* node must occur before another operation can be scheduled.

6.3.1.3 Optimizations

Each of the optimizations of Chapter 4 remain intact in the energy and power-aware approach in order to help reduce search time. However, because of the additional energy constraint, dominance for hashed nodes must be revisited.

Modified dominance check using hashing. At several stops during the exploration of the DAG, the same set of operations have been scheduled to start and finish, although in different orders. In Figure 6.1 we illustrated this by coalescing these paths

into the same location. However, despite arriving at the same location in the DAG, it is frequently the case that the paths have unique attributes, *i.e.*, latency and energy consumption.

Therefore, as each node is generated, a hashing string corresponding to all of the events that have occurred on this node's path is generated and a global hash is accessed. Any node that has the same set of started and finished events hashes to the same location, regardless of the order the events occurred. The current node is compared to all other nodes hashing to this location. If any node is found to be inferior, it is evicted from the hash. If the current node is found to be inferior, then all of its children are pruned from the search space. A node is determined to be inferior if:

1. the node has a latency greater than or equal to the compared node,
2. all active computations in a node have end times greater than or equal to those in the compared node, and
3. the energy consumption of this node is greater than or equal to the compared node.

The key addition under hashing is the final restriction; if one partial schedule's performance is better, but the other's energy cost is better, we cannot necessarily rule out either solution.

Additional time bounds. Utilizing a similar strategy as the $RCSTTF$ bound, we can introduce several additional time bounds based on the power constraint. Each of these additional time bounds are computed by solving a simpler scheduling problem in which one or more constraints are relaxed.

For the first bound, let us begin by relaxing the requirement that an operation may only execute on its legal set of functional units. In this case, we can select the lowest power units possible and schedule each operation on these units in $STTF$ sorted

fashion while remaining under the power bound. This relaxed scheduling will maintain the minimum legal execution time of an operation, rather than that of the function unit it is assigned to.

Another bound is to perform the above power bound on a per-operation-class basis, where only minimum power function unit types that can execute a given operation are considered. This approach prevents a high-power operation from executing on a low power unit on which it is unable to operate (which is allowed in the above scenario). However, this bound will require multi-purpose function units (*e.g.* ALUs) to be considered as available exclusively to each operation class when analyzing each class individually, much like the $RCSTTF$ bound.

One final simple bound is to sum the minimum-power by minimum-delay products of the outstanding unscheduled operations. This sum is compared to the power-constraint multiplied by the time remaining before the time bound is met. If the latter value is exceeded, then the current node is pruned from the search space.

6.3.2 Enumerating the allocation search space

Now let us consider resource allocation. Recall that the full search space under an area bound consists of a multi-dimensional "volume" of allocations (bounded by integer constraints), but, in terms of time minimization, an allocation on the surface of this volume is guaranteed to provide the best possible solution. Allocations below the surface will only contain fewer functional units than an allocation on the surface, therefore at best matching a surface allocation in terms of latency.

The method outlined in Chapter 4 for enumerating the search space is to create a tree of allocations, starting with a node with an empty allocation, as shown in Figure 6.2. This node is expanded by creating a set of new nodes under a set of restrictions:

1. each child node can add only one additional allocated functional unit,

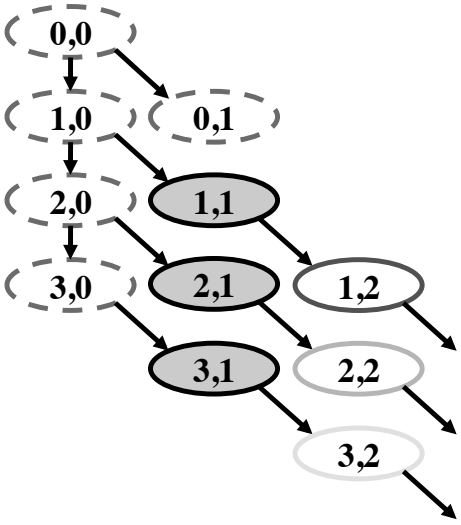

Figure 6.2: Allocation search space for two functional unit types

2. each child node must be distinct from its siblings,

3. the generated node must not exceed the area bound,

4. the selection of children may not break a lexicographical ordering, and

5. the number of functional units of a specific type multiplied by its minimum parametrized power cannot exceed the power bound.

The final restriction has been added in the power-aware approach in order to limit the overall search space. The intuition behind this restriction is that if more function units are allocated than the power bound can budget instantaneously, one or more function units will always be idle during operation, thus, making them unnecessary. These solutions would eventually be pruned by the algorithm, however, we remove them early in the process to avoid any unnecessary searching.

6.3.3 Energy-Minimization

Chapter 4 discussed several search strategies that can be employed by the designer, such as time-minimization and area-minimization. Now, let us consider a different strategy,

energy-minimization.

6.3.3.1 Constraints and targets

In energy-minimization, we aim to find the solution with the lowest energy consumption. Here, the area and time constraints are set, and we aim to find the lowest energy solution. This energy-minimization search strategy would most likely be employed by a designer aiming to find the most-energy efficient solution for applications such as mobile devices and embedded systems where battery life is a concern.

6.3.3.2 Algorithm and search space

The algorithm and search space we selected for energy-minimization are similar to that of generalized time-minimization. However, a few changes need to be made. For reference, refer to the pseudocode for energy minimization in Figure 6.3, where we illustrate a basic approach to both generalized energy-minimization and resource-constrained energy minimization.

In this basic approach, the most obvious difference to notice is that rather than recording minimum time, minimum energy consumption is considered. Of course, the solution must still meet latency, area, and power bounds set by the designer.

There are a few final optimizations to consider in the search algorithm. Energy-minimization can be aided in the allocation process by prioritizing low-energy function units. When deciding which allocations to test first between a set of "equal" function units, we can start by selecting allocations that do not include energy-hungry function units, then try more energy-hungry alternate allocations if the time constraint cannot be met. Similarly, in the scheduling process, we can sort otherwise equal children of a node (based on $STTF$) by minimum energy of the bound functional unit. This guarantees that the lowest energy solutions are considered first when attempting to

```
void EM(DFG dfg, int areaBound){
   AllocationList allocations =
            getPossibleEMAllocations(areaBound);
   int bestEnergy=getWorstCaseEnergy(dfg)+1;
   for (int x=0; x<allocations.length; x++)
      bestEnergy = min(RCEM(dfg,allocations[x]),
                     bestEnergy);
}

int RCEM(DFG dfg, Allocation alloc){
   expand(rootNode, alloc);
   return bestEnergy;
}

void expand(Node node){
   if (isSinkNode(node)){
      if (node.totalEnergy<bestEnergy)
         bestEnergy = node.totalEnergy;
      return;
   }
   NodeList children = getChildren(node, alloc);
   for each child in children
      if (node.totalEnergy>=bestEnergy)
         return;
   for each child in children
      expand(child);
}
```

Figure 6.3: Basic algorithms for energy-minimization

find a solution.

6.4 Voltage Scaling

In this section I will first describe the objective of the voltage scaling approach and describe preliminaries such as the input specification. I then formulate the energy minimization problem in an optimal fashion. Next, I will describe two heuristics, $\frac{de}{dt}$ and $\frac{dE}{dL}$, to approximate the minimum energy solution. Finally, I introduce a method to minimize the number of unique voltages using our heuristic approaches.

6.4.1 Objective and Preliminaries

In this section, the general objective is to minimize the overall energy consumption of a schedule by statically scaling the voltage of each function unit, while at the same time meeting a latency bound. The input to our approach is a scheduled, resource-bound DFG, and a maximum constraint on its latency. The solution space consists of all legal voltages for each function unit, and may be constrained by a maximum number of unique voltages (see section 6.4.5).

6.4.1.1 Binding

The general binding of operation-to-resource-class must be specified ($a + b$ on ALU vs $Adder$). However, the specific binding of operation to specific resource instance need not be specified ($a + b$ on ALU_1 vs ALU_2). In cases where the specific binding is not specified, we utilize a round-robin approach to distribute operations across function units in order to increase the opportunity for scaling.

6.4.2 Exact Problem Formulation: Convex Optimization

Now, let us formulate the problem by analyzing the operations and their data dependencies in the DFG, as well as the additional dependencies introduced by scheduling the operations on a finite set of resources. The following definitions will be used:

- r_i, resource i
- o_j, operation j
- v_i, the voltage of r_i
- v_i^0, the initial voltage of r_i
- e_j, the energy consumed by o_j
- e_j^0, the initial energy consumed by o_j
- t_j, the latency of o_j
- t_j^0, the initial latency of o_j
- E, total energy for schedule
- L, total latency for schedule
- L_{max}, maximum latency for schedule

The function we aim to minimize is the sum of the energy of each operation, after voltage scaling:

$$E = \sum_{\forall j} e_j^0 * (\frac{v_i}{v_i^0})^2 \tag{6.1}$$

where i is mapped such that r_i corresponds to the function unit on which o_j is bound. This sum is minimized by reducing the values of each v_i, subject to a set of constraints.

The constraints we specify are meant to ensure that the scaling performed does not violate the original schedule. The first constraint is to ensure that an operation occurs only after its dependencies have been resolved. For each operation o_j, we introduce a variable s_j that corresponds to the operation's start time. We generate a set of inequalities for each o_j. For each operation o_k that is dependent on o_j:

$$s_j + t_j^0 * \frac{v_i^0}{v_i} <= s_k \tag{6.2}$$

where i is mapped such that r_i corresponds to the function unit to which o_j is bound. This constraint ensures o_k cannot begin until o_j completes execution.

Two additional cases must also be considered. If an operation is dependent on no other operations, the same inequality is used, omitting s_j:

$$t_j^0 * \frac{v_i^0}{v_i} <= s_k \tag{6.3}$$

If an operation has no dependent operations, the same inequality is used, replacing s_k with the value of the latency constraint.

$$s_j + t_j^0 * \frac{v_i^0}{v_i} <= L_{max} \tag{6.4}$$

This constraint ensures that the operation on a resource does not exceed the latency bound.

This optimal formulation is solved using a convex optimization solver; in experimentation the constraints and minimization function were fed into `CVX`, a package for specifying and solving convex programs (Grant and Boyd, 2011; Grant and Boyd, 2008).

6.4.3 Basic Heuristic Method: $\frac{de}{dt}$

Because a convex minimization problem can be complex to solve, particularly as the number of variables and constraints increases, let us now consider an alternative algorithm, which we refer to as $\frac{de}{dt}$, in order to approximate the optimal solution in significantly less time.

The primary goal of this heuristic method is to reduce the voltage of the function units that can produce the greatest reduction in energy (de) for a given change in latency (dt). We can calculate this given the following equations

$$e = e^0 * (\frac{v}{v^0})^2 \tag{6.5}$$

$$v = v^0 * \frac{t^0}{t} \tag{6.6}$$

from which we can formulate the following equation

$$e = e^0 * (\frac{t^0}{t})^2 \tag{6.7}$$

which has the following derivative with respect to t

$$\frac{de}{dt} = -2 * e^0 * \frac{(t^0)^2}{t^3} \tag{6.8}$$

Based on these equations, we can select the function unit with the greatest $|\frac{de}{dt}|$ for scaling.

The basic heuristic algorithm using $\frac{de}{dt}$ shown in Figure 6.4. The algorithm starts by creating a list of all the function units available to scale, and stores them in a list of legally scalable function units. The algorithm then finds the maximum $\frac{de}{dt}$ of all function units in the list, and selects those to scale. Those units are scaled until either

```
DEDT{
  Initial:
    scalable units = all function units
  Start:
    I.   calculate maximum DEDT for scalable units
    II.  select function unit(s) with maximum DEDT
    III. scale selected function units until
        a) next closest DEDT is hit
        b) latency bound is met
          1. calculate units on critical path
          2. remove all units on critical path
             from list of scalable units
    IV.  repeat until list of legal units is empty
}
```

Figure 6.4: Algorithm for $\frac{de}{dt}$ based energy minimization

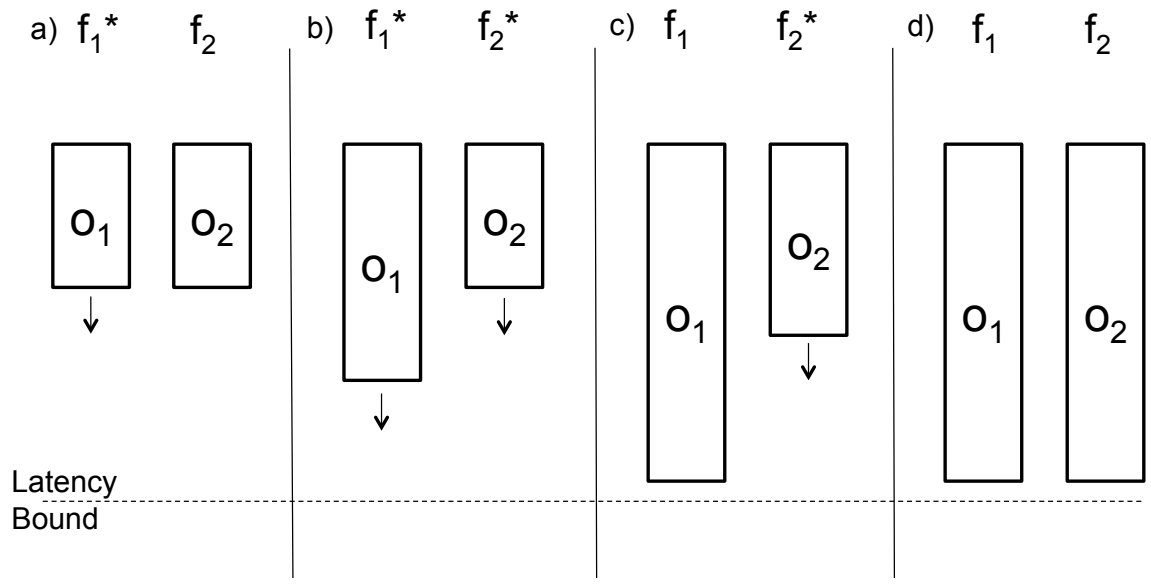

Figure 6.5: Parallel example of $\frac{de}{dt}$ scaling

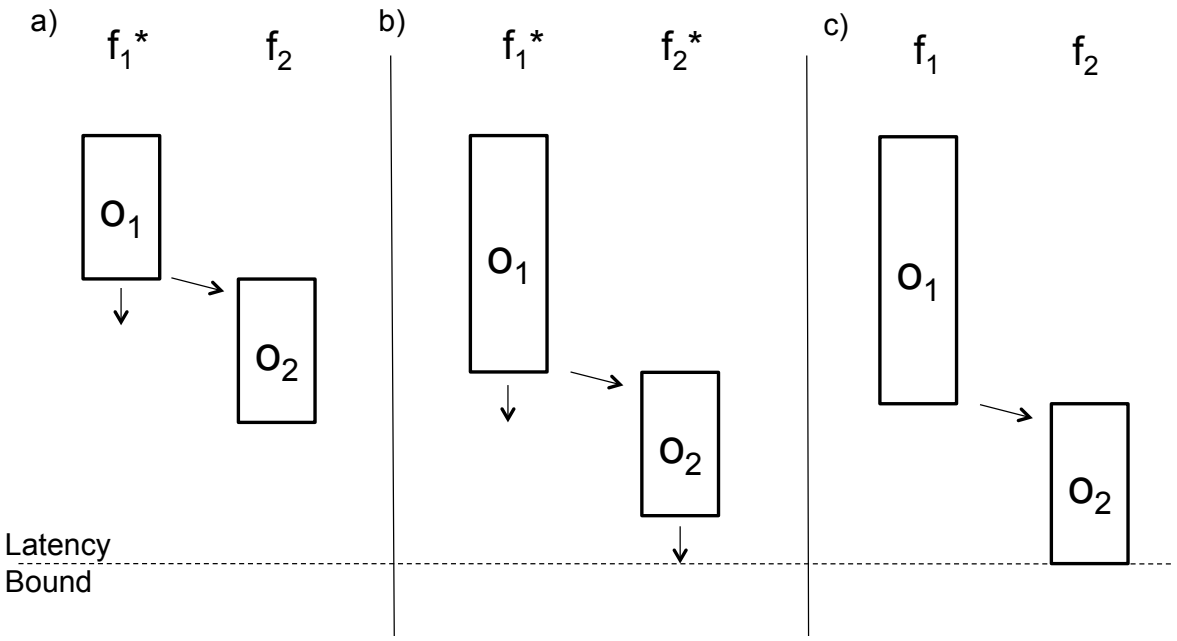

Figure 6.6: Sequential example of $\frac{de}{dt}$ scaling

(i) their $\frac{de}{dt}$ value matches that of another, previously lower $\frac{de}{dt}$ function unit, or (ii) the latency bound is met.

In the case that a $\frac{de}{dt}$ match occurs, the matching $\frac{de}{dt}$ function unit(s) will now be scaled along with the current set of scaled function units. In the case that the latency bound is met, the algorithm calculates the list of function units on the critical path, and removes all those function units from the list of legal function units to scale.

Once the list of legal function units to scale is exhausted, execution completes.

Two example scalings are shown in Figures 6.5 and 6.6. In Figure 6.5a, two operations are being executed in parallel on separate function units. In this example, we assume operation o_1 has a greater $\frac{de}{dt}$ than o_2. As a result, f_1 is scaled until its $\frac{de}{dt}$ matches that of f_2, as in Figure 6.5b. Then, both function units are scaled concurrently, as they have equal $\frac{de}{dt}$. In Figure 6.5c, o_1 meets the latency bound, so f_1 is no longer scaled. Because f_2 is not on the critical path, it can continue scaling until o_2 meets the latency bound (Figure 6.5d).

In Figure 6.6a, two operations are being executed in sequence on separate function units, with the same $\frac{de}{dt}$ values as before. Again, f_1 is scaled until its $\frac{de}{dt}$ matches that of f_2, as in Figure 6.6b. Then both function units are scaled concurrently, as they have equal $\frac{de}{dt}$. In Figure 6.6c, the latency bound is met, and since both function units are on the critical path, they can no longer be scaled.

6.4.4 Advanced Heuristic Method: $\frac{dE}{dL}$

While $\frac{de}{dt}$ is a good starting heuristic, the algorithm presented in 6.4.3 does not consider: (i) the number of operations executing on a function unit, and (ii) internal slack within the schedule. In particular scaling a function unit may have no impact on the overall latency of the schedule due to internal slack, or scaling a unit may have only a slight impact on latency but reduce the energy consumption of multiple operations. As a result, I formulated a different metric, $\frac{dE}{dL}$.

The $\frac{dE}{dL}$ metric is similar to the $\frac{de}{dt}$ metric, with the exception that we calculate the *total* amount of energy change for a change in the *total* schedule latency. To calculate $\frac{dE}{dL}$, we compute the following:

$$dE = \sum (\frac{de}{dt}_j * t_j) \tag{6.9}$$

which sums up the current $\frac{de}{dt}$ value of each scaled operation, weighted by its current latency. We then divide this term by:

$$dT = \sum t_k \tag{6.10}$$

consisting of only the t_ks on the critical path.

We then follow a similar algorithm as before, as shown in Figure 6.7. However, two new steps must be performed: (i) we stop scaling a set of function units if two

```
DEDL{
  Initial:
    scalable units = all function units
  Start:
    I.   calculate maximum DEDL for scalable units
    II.  select function unit(s) with maximum DEDL
    III. calculate impact of scaling on each path
    IV.  scale selected function units until
        a) next closest DEDL is hit
        b) two new operations abut
        b) latency bound is met
          1. calculate units on critical path
          2. remove all units on critical path
             from list of scalable units
    V.   repeat until list of legal units is empty
}
```

Figure 6.7: Algorithm for $\frac{dE}{dL}$ based energy minimization

operations freshly abut, and (ii) we determine when future abutments will occur at each scaling step.

In Figure 6.8a, a new scenario is illustrated with the same $\frac{de}{dt}$ values as before. Here, the initial $\frac{dE}{dL}$ of f_2 is greater than that of f_1, because it is scaling two operations at the cost of only the increased latency of one operation. So, f_2 is scaled until an abutment occurs (Figure 6.8b), which results in the $\frac{dE}{dL}$ of f_2 being cut in half abruptly. Then, f_1 is scaled until the latency bound is met (Figure 6.8c).

6.4.5 Minimizing Unique Voltages

As the number of unique voltages available on a chip is typically limited, I incorporated into the proposed heuristics an additional method to reduce the number of unique voltages. Here, I will explain these using $\frac{dE}{dL}$ as an example.

The first step is to run the $\frac{dE}{dL}$ heuristic, and generate a list of the unique $\frac{dE}{dL}$ values that constitute the initial solution. We then generate a voltage grouping that minimizes overall energy consumption, using the algorithm shown in Figure 6.9. This algorithm

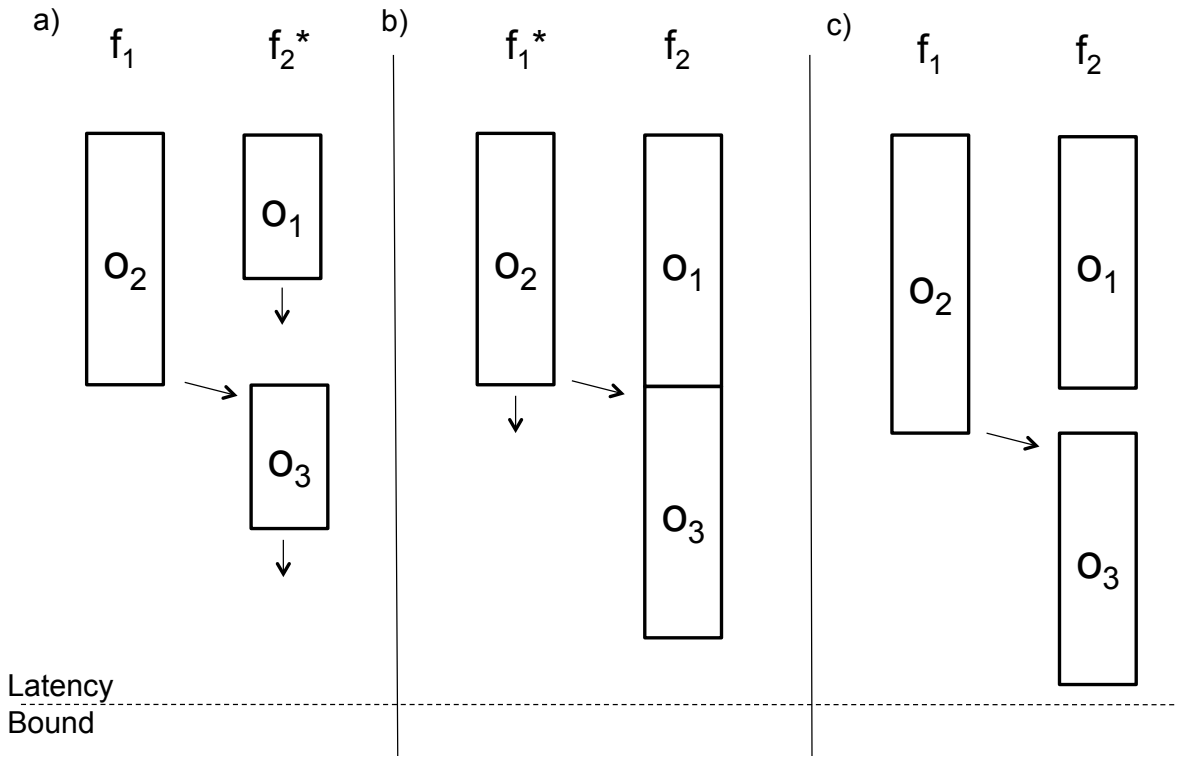

Figure 6.8: Example of abutment in $\frac{dE}{dL}$ scaling

aims to minimize the total energy while increasing the voltages of some units such that there are N unique groups.

After the grouping is complete, new slack may be available, since the voltages of some units may have increased. We can therefore re-run the $\frac{dE}{dL}$ method, computing the $\frac{dE}{dL}$ in a group-wise fashion, which equations 6.9 and 6.10 are amenable to, keeping in mind that each time any member of a voltage group is scaled, the full group must be scaled with it.

6.5 Results

This section illustrates each method's capability to quickly determine an optimal solution under a variety of constraints and minimizations. Utilizing the benchmark set from Section 4.6, a total of 36 test cases were run for the scheduling approach. Next,

```
Grouping{
 Initial:
    Run DEDL
 Start:
   I.   sort function units by voltage selected
   II.  create an N pointers for the maximum groups
          -> all function units are mapped to the
          voltage of the closest pointer above it
   III. set each pointer to the highest voltage unit
   IV.  if any pointer can be shifted to a lower
          voltage unit and energy is reduced,
          shift and update
   V.   repeat IV until no energy reduction possible
}
```

Figure 6.9: Grouping algorithm for voltages

the heuristic voltage scaling solution was evaluated by comparison to the optimal formulation in section 6.4.2. For the voltage scaling optimization, a suite of over 120 test cases were performed to verify its effectiveness.

6.5.1 Setup

Two sets of experiments were performed: one set for the modified scheduling approach from Section 6.3, and another for the voltage scaling approach for Section 6.4. Each section used the same set of benchmarks, with the exception of the TEA test case which was slightly modified for each.

For each benchmark the same library of function units was provided, with parameters as shown in Table 6.2 for the synthesis approach and Table 6.4 for the voltage scaling approach. Each operation type was given its own dedicated function unit type, and a multi-purpose ALU was introduced on which any operation could run. The latency and energy of the ALU varied depending on the class of operation executed, but the power consumption remained constant. These values are assumed to include the latency and energy cost of handshaking when using asynchronous communication;

these additional costs can be ignored when targeting a clocked system.

Each approach was implemented in Java using standard libraries. Benchmarks were tested on a Macbook Pro with a 2.8 GHz Intel Core 2 Duo processor and 4GB of RAM on JVM 1.5. Run-times recorded are the output generated by the Unix time command.

6.5.1.1 Synthesis Experiments

For the first set of experiments the proposed synthesis approach was evaluated. In experimentation, six different benchmark DFGs were used for analysis, described in section 6.5.2. Two sets of constraints were generated for each DFG, constraining latency, area, energy, and power as shown in Table 6.3. The first set of constraints demands low latency but allows for greater area and power consumption, while the second set demands low area and power consumption but allows for a greater latency. Energy constraints were equal for each case. For one very large test case, TEA, constraints were selected that could not be met to illustrate the amount of time necessary to make the "no solution" determination for each minimization type.

For each DFG, six different test cases were performed, varying the parameters for each case by minimizing for either time, area, or energy, then performing each minimization strategy on both sets of constraints.

6.5.1.2 Voltage Scaling Experiments

For the voltage scaling approach, an area and time constraint were set for each DFG during scheduling, as shown in Table 6.5. Each DFG was then run through the scheduler to produce both a minimum energy schedule, *ME* and a minimum latency schedule, *MT*. These two schedules were compared to determine whether or not the minimum-latency schedule would have more opportunity for scaling despite the initially higher energy cost.

For two of the benchmarks, ELP and TEA, the *ME* and *MT* schedules were the same. In these cases, the latency bounds and allocations were varied to show the impact on the effectiveness of each heuristic.

For each test case, the resulting schedule had some remaining *slack*, in other words, it completed execution before the time bound was met. The percent of slack in the schedule is shown in Table 6.5.

Each of the schedules were run through the optimal (**OPT**), $\frac{de}{dt}$ (**DEDT**), and $\frac{dE}{dL}$ (**DEDL**) voltage scalers. For the large TEA test case, the optimal solver could not produce a solution, and it is therefore absent from the table.

Finally, each solution was then constrained by limiting the number of discrete voltages allowed by the **DEDT** and **DEDL** solvers, reducing the maximum number of discrete voltages to between one and four, as described in section 6.4.5.

The optimal voltage scaling approach was implemented in MATLAB 7.11 (MATLAB, 2010) and run using the `CVX` package (Grant and Boyd, 2011; Grant and Boyd, 2008).

6.5.2 Benchmark Description

Six different benchmarks were used in experimentation:

- **ODE**: solves ordinary differential equations using the Euler method. It receives as input the coefficients of a third-degree ordinary differential equation, along with additional parameters such as step size.
- **DotProd8**: performs a dot product on 8-element vectors.
- **Cosine**: approximates the cosine of a number using the first nine terms of its Taylor series.
- **Seventh**: runs a seventh order filter from the IMEC cathedral system.

- **Elliptic**: runs a fifth order elliptic wave filter.

- **TEA**$_1$: performs two complete passes of the fully unrolled (32x) tiny-encryption algorithm in parallel (very large example). This test case was used only in synthesis experimentation.

- **TEA**$_2$: performs four passes of the unrolled tiny-encryption algorithm in parallel, each unrolled eight times. This test case was used only in voltage scaling experimentation.

The node count of each benchmark, including root and sink nodes, is given in Table 6.1.

6.5.3 Discussion of Results

Since two main sets of experiments were performed, I will first discuss the results corresponding to the approach described in Section 6.3, then discuss the voltage scaling results based on the approach in Section 6.4.

6.5.3.1 Synthesis Results

Table 6.3 shows the experimental results as well as the given constraints for each test case. For each minimization result listed, the associated run-time of our tool is also given. Note that these run-times include JVM startup/shutdown time, which was estimated to be about 0.13s in experimentation.

Using the proposed method, the search space was exhausted within a second in 31 of 36 test cases. The most anomalous result in terms of performance is that of DotProd8 under latency minimization. This example has significantly higher initial concurrency, beginning with 8 parallel multiplications, each with an equal STTF. In this case, the solver compared several schedule permutations in which it chose between an ALU and a

dedicated multiply unit for each multiplication. The high concurrency mixed with this specific set of constraints compounded to result in a higher run-time. Under a tighter area constraint, the run-time may be reduced, as the ALU could not be selected for use. Under a broader area constraint, the run-time may also be reduced, since a more optimal solution with only dedicated function units would be explored.

Because the both the energy cost and the latency of the ALU are greater than that of the dedicated function units, in most cases the solution that minimized for energy was also the one with the lowest latency. However, for ODE and DotProd8, area and latency constraints were selected in such a way that an ALU was required to meet the latency constraint. In these cases, varying minimal energy results were produced dependent on the constraints specified.

In the final TEA example, in which the constraints were too tight for a solution to be found, we can see that the area minimization example took the longest to fail. In this case, each possible unique permutation (25 total) was tried heuristically in a lowest to greatest area fashion, all failing. After the full heuristic pass failed, an exact time-minimization was attempted for each failed allocation. As each allocation failed, any allocation subsumed by the failed allocation was also discarded; resulting in only four "surface" allocations that needed to be attempted by the solver. When these four failed, the solution space was exhausted. In contrast, the energy-constrained and time-constrained solutions needed to only consider the four surface allocations that dominate the other 21 allocations, resulting in a lower run-time.

6.5.3.2 Voltage Scaling Results

Table 6.6 shows a comparison of the optimal method (**OPT**) to our two heuristic methods. In this table there are three results columns: the energy consumed by the schedule after scaling, the energy consumed by the schedule when normalized to the

original energy of the schedule, and the run-time of the solver.

First let us compare the *ME* and *MT* scheduling methods. For ODE, COS, and 7TH, the energy of the solution provided by all solvers is lower when using the *ME* schedule, but for DP8 the energy is lower when using the *MT* schedule. Some insight can be gained by comparing the difference in energy between the *ME* and *MT* schedules to the slack on the schedule. For DP8, there is significantly more slack (7x) using an *MT* schedule at the cost of 14% more energy, while in the other examples the difference in slack is less pronounced. More experimentation is needed to determine which scenarios are more amenable to scaling using *ME* versus *MT* schedules, but at first glance the *ME* schedule seems to be a better choice.

Next, we can compare the solutions provided by the optimal solver to the **DEDL** and **DEDT** solvers. In the worst case (7TH-*ME*), the heuristic solution consumed 9.7% more energy than the optimal solution. For all other test cases, either the **DEDL** or the **DEDT** found a solution within 5% of the energy of **OPT**. For over half of the test cases, these heuristics found a solution that was within 1% of the optimal: (ODE-*ME*, DP8-*MT*, COS-*ME*, COS-*MT*, ELP-*2*, ELP-*3*).

Table 6.7 shows the results of reducing the maximum number of unique voltages. The first results column shows the normalized energy when using a separate voltage for each function unit. The next four columns show the normalized energy as the number of unique voltages is reduced from four to one. In some cases there were not enough function units to have four unique voltages (*e.g.*, ODE-*MT* has only two ALUs), so these results are omitted as they are the same as those in the first column.

Here we can see a general trend where the overall energy increases as we reduce the number of unique voltages. In some cases, a poor grouping is selected, limiting the amount of scaling on some function units by another function unit. As an example, consider ODE-*ME*, where the $N_v = 3$ grouping creates a higher energy solution than

Table 6.1: DFG nodes per benchmark

Benchmark	# of Nodes
ODE	11
DotProd8	17
Cosine	26
Seventh	31
Elliptic	36
TEA	1090

the $N_v = 4$ (expected) and $N_v = 2$ solutions. By modifying the grouping algorithm in future work, I aim to target these anomalies.

Table 6.7 gives insight into the effectiveness of *DEDT* versus *DEDL*, particularly the number of voltage groups changes. With a maximum number of voltages, *DEDL* outperforms *DEDT* in only two test cases, and does significantly worse in one test case. As the number of unique voltages is reduced to four, *DEDL* outperforms *DEDT* in four test cases. At three unique voltages, *DEDL* outperforms *DEDT* in six test cases. At two voltages, *DEDL* outperforms *DEDT* in all but one test case, TEA-*2*, where the difference is negligible. At one voltage, there is no difference between the two. We can draw the conclusion from these results that *DEDL* is more effective than *DEDT* at reducing energy consumption as the number of unique voltages decreases, while impacts are more varied at the maximum number of unique voltages.

Finally, consider the run-time of each solver, as shown in Table 6.6. In eighteen of the test cases, the solver ran in 40ms or less. Only two examples took longer than a second, those solving the large TEA benchmark. When compared to the run-time of the optimal solver, which took 2-14 seconds, the heuristic results were several orders of magnitude faster: from 20x at worst (DP8-*MT*) to $10,000$x at best (ELP-*1*). For the TEA benchmark, the optimal solver ran for several minutes but was unable to find a solution.

Table 6.2: Function unit parameters

Function Unit	Operation Class	Area (unit)	Latency (unit)	Energy (unit)	Power (unit)
Adder	+	24	8	80	10
Subtracter	−	24	8	80	10
Multiplier	*	96	16	240	15
XOR	^	8	6	60	10
Shifter	<<, >>	8	4	40	10
ALU	+	104	10	200	20
ALU	−	104	10	200	20
ALU	*	104	20	400	20
ALU	^	104	8	160	20
ALU	<<, >>	104	6	120	20

6.6 Conclusion

In this chapter I described two key extensions to a branch-and-bound framework for high-level synthesis. The first was to incorporate power and energy constraints into the synthesis and scheduling process, which included developing an approach for minimizing energy consumption. Through experimentation I have shown that this approach is capable of performing many different flavors of optimization rapidly, generates optimal solutions under a minute for each test case.

Next, I formulated an exact solution for energy minimization via static voltage scaling in the form of a convex optimization problem. Then, I presented two efficient heuristics for energy reduction, $\frac{de}{dt}$ and $\frac{dE}{dL}$, that compute a close approximation of the solution in a fraction of the time (20-10,000x faster than the optimal solution). In experimentation, these metrics were within 5% of the optimal energy in 10 of 11 test cases, and 1% of the optimal solution in 6 of 11 test cases.

Finally, I described a method for reducing the total number of voltages by grouping function units of similar voltages into distinct scalable groups. In future work, I plan to further investigate the relationship between *ME*, *MT*, and schedule slack, look at more

Table 6.3: Constraints and results for each benchmark

Benchmark	Constraints				Minimization Results					
	Latency (unit)	Area (unit)	Energy (unit)	Power (unit)	Latency (unit)	Runtime (s)	Area (unit)	Runtime (s)	Energy (unit)	Runtime (s)
ODE	120	140	3000	40	110	0.17	128	0.15	2440	0.17
ODE	50	350	3000	60	48	0.16	320	0.18	1520	0.15
DotProd8	130	210	4000	40	126	42.91	200	0.18	3640	6.77
DotProd8	75	600	4500	60	56	0.17	320	0.31	2480	0.15
Cosine	180	240	4500	40	160	0.18	216	0.17	4480	0.16
Cosine	130	450	4500	60	112	0.18	312	0.19	4480	0.16
Seventh	180	260	4800	40	144	0.22	240	0.18	4720	0.16
Seventh	120	400	4800	60	112	0.24	336	0.21	4720	0.17
Elliptic	160	240	5000	40	152	0.19	240	0.20	4000	0.16
Elliptic	145	320	5000	60	144	0.21	240	0.24	4000	0.16
TEA	5000	60	90000	40	4608	8.86	40	2.72	71680	3.89
TEA	1925	128	90000	60	-	3.11	-	38.66	-	0.57

Table 6.4: Function unit parameters

Function Unit	Operation Class	Area (unit)	Latency (unit)	Energy (unit)
Adder	+	24	10	100
Subtracter	−	24	10	100
Multiplier	∗	96	12	240
XOR	ˆ	8	8	80
Shifter	<<, >>	8	6	60
ALU	+	104	12	240
ALU	−	104	12	240
ALU	∗	104	14	280
ALU	ˆ	104	10	200
ALU	<<, >>	104	8	160

effective methods for grouping voltages to improve our results, and utilize design hints from our static voltage scaling approach as feed-back into the scheduling and binding portions of synthesis.

Table 6.5: Benchmark parameters

Benchmark	Ops (#)	Schedule Energy	Latency Bound	Schedule Slack	**Function Unit Allocation**					
					ALU	+	−	∗	>>	ˆ
ODE-*ME*	11	1600	80	*12.5%*	0	3	1	1	0	0
ODE-*MT*	11	2360	80	*17.5%*	2	0	0	0	0	0
DP8-*ME*	17	2620	80	*2.5%*	0	5	0	2	0	0
DP8-*MT*	17	2980	80	*17.5%*	1	1	0	2	0	0
COS-*ME*	26	4640	200	*44%*	0	1	0	3	0	0
COS-*MT*	26	5040	200	*47%*	1	1	0	2	0	0
7TH-*ME*	31	5000	110	*5.5%*	0	2	4	4	0	0
7TH-*MT*	31	5500	110	*7.3%*	1	1	1	4	0	0
ELP-*1*	36	4520	150	*2.7%*	0	3	0	3	0	0
ELP-*2*	36	4520	200	*27%*	0	3	0	3	0	0
ELP-*3*	36	4520	750	*80.5%*	0	3	0	3	0	0
TEA-*1*	610	46720	3500	*17.7%*	0	1	0	0	1	1
TEA-*2*	610	46720	1500	*32%*	0	3	0	0	2	1

Table 6.6: Comparison of optimal and heuristic methods

Exp. Setup		Energy	Energy	Runtime
Benchmark	**Strategy**	(scaled)	(normalized)	(seconds)
ODE-*ME*	**OPT**	1120	0.701	1.934
ODE-*ME*	**DEDL**	1120	0.701	0.004
ODE-*ME*	**DEDT**	1120	0.701	0.002
ODE-*MT*	**OPT**	1480	0.629	2.006
ODE-*MT*	**DEDL**	1520	0.645	0.002
ODE-*MT*	**DEDT**	1610	0.681	0.001
DP8-*ME*	**OPT**	2190	0.836	3.033
DP8-*ME*	**DEDL**	2310	0.881	0.006
DP8-*ME*	**DEDT**	2300	0.877	0.001
DP8-*MT*	**OPT**	1820	0.611	3.033
DP8-*MT*	**DEDL**	1880	0.631	0.143
DP8-*MT*	**DEDT**	1820	0.611	0.001
COS-*ME*	**OPT**	1260	0.271	8.880
COS-*ME*	**DEDL**	1260	0.272	0.555
COS-*ME*	**DEDT**	1260	0.272	0.003
COS-*MT*	**OPT**	1380	0.273	6.849
COS-*MT*	**DEDL**	1390	0.276	0.005
COS-*MT*	**DEDT**	1500	0.297	0.001
7TH-*ME*	**OPT**	2880	0.575	6.132
7TH-*ME*	**DEDL**	3160	0.632	0.284
7TH-*ME*	**DEDT**	3160	0.631	0.007
7TH-*MT*	**OPT**	3330	0.606	8.729
7TH-*MT*	**DEDL**	3420	0.622	0.008
7TH-*MT*	**DEDT**	3400	0.618	0.014
ELP-*1*	**OPT**	4080	0.903	13.062
ELP-*1*	**DEDL**	4180	0.924	0.001
ELP-*1*	**DEDT**	4160	0.919	0.001
ELP-*2*	**OPT**	2300	0.508	14.115
ELP-*2*	**DEDL**	2320	0.514	0.040
ELP-*2*	**DEDT**	2300	0.510	0.001
ELP-*3*	**OPT**	163	0.036	13.112
ELP-*3*	**DEDL**	165	0.037	0.124
ELP-*3*	**DEDT**	164	0.036	0.001
TEA-*1*	**DEDL**	22100	0.473	0.450
TEA-*1*	**DEDT**	22100	0.472	1.406
TEA-*2*	**DEDL**	20600	0.442	4.086
TEA-*2*	**DEDT**	20600	0.442	0.302

Table 6.7: Normalized energy versus number of unique voltages

Exp. Setup		**Number of Voltages (N_v)**				
Bchmark	**Strat**	Max	Four	Three	Two	One
ODE	OPT	0.478	0.478	0.478	0.479	0.490
ODE	DEDL	0.479	0.479	0.479	0.485	0.490
ODE	DEDT	0.526	0.526	0.526	0.487	0.490
ODE	DEDS	0.479	0.479	0.479	0.479	0.490
ODE	OPT	0.701	0.701	0.741	0.719	0.766
ODE	DEDL	0.701	0.701	0.744	0.719	0.766
ODE	DEDT	0.701	0.701	0.744	0.719	0.766
ODE	DEDS	0.703	0.703	0.745	0.719	0.766
DP8	OPT	0.854	0.854	0.854	0.854	0.964
DP8	DEDL	0.943	0.943	0.986	0.963	0.964
DP8	DEDT	0.943	0.943	0.986	0.963	0.964
DP8	DEDS	0.945	0.945	0.986	0.963	0.964
DP8	OPT	0.836	0.863	0.889	0.891	0.951
DP8	DEDL	0.881	0.904	0.891	0.948	0.951
DP8	DEDT	0.877	0.912	0.944	0.948	0.951
DP8	DEDS	0.885	0.914	0.947	-	0.951
COS	OPT	0.297	0.321	0.332	0.341	-
COS	DEDL	0.303	0.325	0.335	0.343	0.360
COS	DEDT	0.306	0.326	0.327	0.344	0.360
COS	DEDS	0.3 00	0.323	-	0.343	0.360
COS	OPT	0.271	0.271	0.271	0.295	0.314
COS	DEDL	0.272	0.272	0.271	0.295	0.314
COS	DEDT	0.272	0.272	0.272	0.295	0.314
COS	DEDS	0.272	0.272	0.271	0.295	0.314
7TH	OPT	0.575	0.620	0.655	0.691	0.894
7TH	DEDL	0.632	0.824	0.848	0.894	0.894
7TH	DEDT	0.631	0.824	0.848	0.895	0.894
7TH	DEDS	0.633	0.824	0.848	0.894	0.894
7TH	OPT	0.616	0.617	0.631	0.622	-
7TH	DEDL	0.623	0.624	0.626	0.658	0.683
7TH	DEDT	0.648	0.648	0.7 00	0.697	0.683
7TH	DEDS	0.624	0.663	0.624	0.655	0.683
ELP	OPT	0.903	0.903	0.903	-	-
ELP	DEDL	0.924	0.93	0.93	0.951	0.947
ELP	DEDT	0.919	0.919	0.919	0.951	0.947
ELP	DEDS	0.923	0.919	0.919	0.951	0.947
ELP	OPT	0.508	0.508	0.508	-	-
ELP	DEDL	0.514	0.513	0.518	0.514	0.533
ELP	DEDT	0.510	0.510	0.510	0.543	0.533
ELP	DEDS	0.515	0.522	0.520	0.539	0.533
ELP	OPT	0.036	0.036	0.036	-	-
ELP	DEDL	0.037	0.036	0.037	0.037	0.038
ELP	DEDT	0.036	0.036	0.036	0.039	0.038
ELP	DEDS	0.037	0.037	0.037	0.038	0.038
TEA	DEDL	0.473	0.473	0.473	0.477	0.677
TEA	DEDT	0.472	0.472	0.472	0.477	0.677
TEA	DEDS	0.477	0.477	0.477	0.479	0.677
TEA	DEDL	0.442	0.449	0.448	0.463	0.462
TEA	DEDT	0.442	0.462	0.462	0.462	0.462
TEA	DEDS	0.442	0.450	0.448	0.463	0.462

Chapter 7

Conclusion

7.1 Summary of Contributions

The work presented in this thesis aimed to provide a high-level synthesis approach, robust and powerful, in order to address a gap in existing asynchronous design flows. At the highest level, the overall goal was to develop an automated tool for designers in order to significantly reduce overall effort, particularly by allowing design-space exploration to be handled in an automated fashion. I have provided a comprehensive, systematic approach to meet this goal, developing several synthesis techniques and optimizations addressing specific problems in the realm of asynchronous high-level synthesis.

In Chapter 3, I provided an alternative to the low-to-medium performance, syntax-directed Haste design flow by producing a source-to-source compiler to transform a sequential specification into a highly concurrent one. This work produced high-performance, slack-matched, data-driven pipelines for situations when a designer is concerned primarily with speed, while still leveraging much of the Haste tool suite.

In Chapter 4, I tackled the problem of resource sharing. In this chapter I showed a fast, optimal approach for performing resource sharing under a variety of constraints. This work is further extended in Chapter 6 by incorporating energy and power. The approach provided several synthesis strategies to the designer, allowing optimization

for area, latency, and energy, as well as constraints on these quantities in addition to peak power. Key algorithms for scheduling and dynamic allocation were presented, as well as many optimizations to significantly increase the performance of the proposed branch-and-bound strategy.

Chapter 5 provided yet another path to the designer, a powerful hybrid approach that targeted both performance and area. In this work I developed a strategy for synthesis of pipelined, resource-shared systems. A key novelty is that this approach concurrently performs slack-matching and scheduling in order to produce high-performance data-flow pipelines. I further extended the approach to model loops and conditionals and provided a heuristic hierarchical method for attacking large examples.

Finally, Chapter 6 provided a means for considering power and energy. Beyond extensions to the work presented in Chapter 4 to incorporate these constraints, I developed a post-scheduling method for voltage-scaling in order to further improve the energy consumption of a specification. This approach included both optimal and heuristic formulations to the problem, as well as a heuristic method for limiting the total number of unique voltage levels.

My work has tackled several challenging problems in asynchronous high-level synthesis, and provided solutions to several problems that either had not been solved before or had not been solved before exactly. In particular, my approach to single-token scheduling (Chapter 4) is the first exact solution to the asynchronous version of the problem. Furthermore, my approach to multi-token scheduling (Chapter 5) is the first exact solution to this problem in both asynchronous as well as synchronous domains. Elevating energy and power considerations as first-class constraints into scheduling (Chapter 6) is also the first exact approach to this problem.

The bulk of the work of this dissertation likely is applicable also to synchronous design (*including synchronous elastic systems*). Interestingly, while working on the

asynchronous problem, I was forced to think out-of-the-box—*i.e.*, in terms of relative order instead of absolute time—because asynchronous systems do not have a notion of clocking and absolute time. It turns out that, even though application to synchronous design was beyond the scope of this thesis, it is likely that my relative order approach is not only applicable to, but highly efficient for, synchronous systems as well.

Together, these contributions combine to create a rich set of synthesis methods, allowing a designer to quickly and easily perform design-space exploration in an effective, automated fashion.

7.2 Future Work

An exhaustive consideration of high-level synthesis for asynchronous systems is beyond the scope of any thesis; many significant avenues have yet to be explored. Several extensions were not incorporated into this thesis due to scope. I will briefly note here future directions for the research I have presented.

One key component missing in the synthesis flow is an appropriate back-end. The research in Chapter 3 produces a new Haste specification to run back into their compiler. However, this compiler is still syntax-directed, and the circuit produced, although relatively much faster, is highly control driven. Further, licencing for academic use has been discontinued, and therefore another viable option is needed. Similarly, the work presented in Chapters 4, 5, and 6 did not have an available back-end, and thus relied on my own event-driven simulator for verification.

A more robust treatment of loops and conditionals is needed across the board. Chapters 3 and 5 presented methods for handling conditionals, such as early and late decision, but a heuristic or optimal method for selecting the best conditional architecture has not been developed. A starting point for handling loops via loop pipelining

was illustrated in joint work with Gill et al. in (Gill et al., 2006), but an approach for determining token and unroll count has not yet been developed. A method for full design-space exploration for these constructs would be a key addition.

One necessary requirement to incorporate choice and non-determinism is a method for analyzing constructs that are stochastic in nature. The work in Chapter 5 could be extended to model variable count loops. Average-case analysis for operations could be considered rather than worst-case for the approaches in Chapters 4 and 5.

The energy and power approach in Chapter 6 still has opportunity for refinement. More sophisticated heuristics could be considered to produce results closer to the optimal values. Further, the voltage scaling method itself could possibly be incorporated into the scheduling procedure to produce even better results.

Finally, more experimentation is always needed to analyze where each approach is lacking and what opportunities exist for further optimization. Testing on large, real-world examples is needed. Average case performance of each tool based on attributes such as specification size, branching level, etc., would be highly useful.

Combining these proposed extensions with the existing approach in this thesis would potentially provide an immensely useful and robust design flow. I hope to have the opportunity to attack these problems in future work.

BIBLIOGRAPHY

Andrikos, N., Lavagno, L., Pandini, D., and Sotiriou, C. P. (2007). A fully-automated desynchronization flow for synchronous circuits. In *DAC*, pages 982–985.

Bachman, B. M. (1998). Architectural-level synthesis of asynchronous systems. Master's thesis, Utah.

Bachman, B. M., Zheng, H., and Myers, C. J. (1999). Architectural synthesis of timed asynchronous systems. In *In Proc. International Conf. Computer Design (ICCD)*, pages 354–363.

Badia, R. M. and Cortadella, J. (1993). High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *In Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press.

Bardsley, A. and Edwards, D. A. (2000). The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*.

Beerel, P., Ozdag, R., and Ferretti, M. (2010). *A Designer's Guide to Asynchronous VLSI*. A Designer's Guide to Asynchronous VLSI. Cambridge University Press.

Beerel, P. A., Kim, N.-H., Lines, A., and Davies, M. (2006). Slack matching asynchronous designs. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, page 184, Washington, DC, USA.

Budiu, M. (2003). *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department. Technical report CMU-CS-03-217.

Burns, F., Shang, D., Koelmans, A., and Yakovlev, A. (2004). Scheduling and allocation using closeness tables. *Computers and Digital Techniques, IEE Proceedings -*, 151(5):332–340.

Chelcea, T. and Nowick, S. M. (2002). Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conf.*

Chen, D., Cong, J., Fan, Y., and Xu, J. (2006). Optimality study of resource binding with multi-vdds. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 580–585, New York, NY, USA. ACM.

Cortadella, J., Kondratyev, A., Lavagno, L., and Sotiriou, C. P. (2006). Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921.

Coussy, P. and Morawiec, A. (2008). *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer.

Dasdan, A. and Gupta, R. K. (1997). Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899.

Edwards, D. and Bardsley, A. (2002). Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18.

Gill, G., Hansen, J., Agiwal, A., Vicci, L., and Singh, M. (2009). A high-speed gcd chip: A case study in asynchronous design. In *VLSI, 2009. ISVLSI '09. IEEE Computer Society Annual Symposium on*, pages 205 –210.

Gill, G., Hansen, J., and Singh, M. (2006). Loop pipelining for high-throughput stream computation using self-timed rings. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 289–296.

Gill, G. D. (2010). *Analysis and optimization for pipelined asynchronous systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA. AAI3402352.

Grant, M. and Boyd, S. (2008). Graph implementations for nonsmooth convex programs. In Blondel, V., Boyd, S., and Kimura, H., editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited. `http://stanford.edu/~boyd/graph_dcp.html`.

Grant, M. and Boyd, S. (2011). CVX: Matlab software for disciplined convex programming, version 1.21. `http://cvxr.com/cvx`.

Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. (2003). Spark: A high-lev l synthesis framework for applying parallelizing compiler transformations. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 461, Washington, DC, USA. IEEE Computer Society.

Hansen, J. and Singh, M. (2008). Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach. In *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pages 15 –25.

Hansen, J. and Singh, M. (2010a). An energy and power-aware approach to high-level synthesis of of asynchronous systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*.

Hansen, J. and Singh, M. (2010b). A fast branch-and-bound approach to high-level synthesis of asynchronous systems. In *Proc. Int. Symp. on Asynchronous Circuits and Systems (ASYNC)*.

Hansen, J. and Singh, M. (2012). Multi-token resource sharing for pipelined asynchronous systems. In *Proc. Design, Automation and Test in Europe (DATE)*.

Haste (2008). The Haste/TiDE Design Flow. `http://www.handshakesolutions.com`.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Jensen, J. B. and Nielsen, J. R. (2007). Compiling from haste to CDFG: a front end for an asynchronous circuit synthesis system. Bachelor of Engineering thesis, Technical University of Denmark (IMM-DTU), 2007.

Johnson, M. C. and Roy, K. (1997). Datapath scheduling with multiple supply voltages and level converters. *ACM Trans. Des. Autom. Electron. Syst.*, 2(3):227–248.

Kondratyev, A., Lavagno, L., Meyer, M., and Watanabe, Y. (2011). Realistic performance-constrained pipelining in high-level synthesis. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1382–1387.

Leiserson, C. and Saxe, J. (1991). Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35.

Lin, Y.-R., Hwang, C.-T., and Wu, A. C.-H. (1997). Scheduling techniques for variable voltage low power designs. *ACM Trans. Des. Autom. Electron. Syst.*, 2(2):81–97.

Lines, A. M. (June 1995, revised 1998). Pipelined asynchronous circuits. Master's thesis, California Institute of Technology.

lpsolve (2009). lp_solve: A Mixed Integer Linear Programming (MILP) solver. Available online at `http://lpsolve.sourceforge.net/`.

Manohar, R. and Martin, A. J. (1998a). Slack elasticity in concurrent computing. *Lecture Notes in Computer Science*, 1422.

Manohar, R. and Martin, A. J. (1998b). Slack elasticity in concurrent computing. In *Proceedings of the Fourth International Conference on the Mathematics of Program Construction, Lecture Notes in Computer Science 1422*, pages 272–285. Springer-Verlag.

Manzak, A. and Chakrabarti, C. (2002). A low power scheduling scheme with resources operating at multiple voltages. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(1):6–14.

Martin, A. J., Lines, A., Manohar, R., Nyström, M., Pénzes, P., Southworth, R., and Cummings, U. (1997). The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181.

MATLAB (2010). *version 7.11.0 (R2010b)*. The MathWorks Inc., Natick, Massachusetts.

Micheli, G. D. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education.

Nielsen, S. F. (2005). *Behavioral synthesis of asynchronous circuits*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU.

Nielsen, S. F., Sparsø, J., Jensen, J. B., and Nielsen, J. S. R. (2009). A behavioral synthesis frontend to the haste/tide design flow. In *Proc. Int. Symp. on Asynchronous Circuits and Systems)*, pages 185–194, Washington, DC, USA. IEEE Computer Society.

Nielsen, S. F., Sparsø, J., and Madsen, J. (2004). Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation. *Digital Systems Design, Euromicro Symposium on*, pages 298–305.

Paulin, P. G. and Knight, J. P. (1987). Force-directed scheduling in automatic data path synthesis. In *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, New York, NY, USA. ACM.

Plana, L. A., Taylor, S., and Edwards, D. (2005). Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 703–710.

Saito, H., Hamada, N., Jindapetch, N., Yoneda, T., Myers, C., and Nanya, T. (2007). Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E90-A(12):2790–2799.

Saito, H., Jindapetch, N., Yoneda, T., Myers, C., and Nanya, T. (2006). Ilp-based scheduling for asynchronous circuits in bundled-data implementation. *Computer and Information Technology, International Conference on*, page 172.

Singh, M. and Nowick, S. (2001). Mousetrap: ultra-high-speed transition-signaling asynchronous pipelines. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 9 –17.

Singh, M. and Nowick, S. (2007). The design of high-performance dynamic asynchronous pipelines: High-capacity style. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(11):1270 –1283.

Singh, M., Tierno, J. A., Rylyakov, A., Rylov, S., and Nowick, S. M. (2002). An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 GigaHertz. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, Manchester, UK. IEEE Computer Society Press.

Sllame, A. M. and Drabek, V. (2002). An efficient list-based scheduling algorithm for high-level synthesis. In *DSD '02: Proceedings of the Euromicro Symposium*

on Digital Systems Design, page 316, Washington, DC, USA. IEEE Computer Society.

Sutherland, I. and Fairbanks, S. (2001). Gasp: a minimal fifo control. In *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 46 –53.

Sutherland, I. E. (1989). Micropipelines. *Commun. ACM*, 32(6):720–738.

Taubin, A., Cortadella, J., Lavagno, L., Kondratyev, A., and Peeters, A. M. G. (2007). Design automation of real-life asynchronous devices and systems. *Foundations and Trends in Electronic Design Automation*, 2(1):1–133.

Teifel, J. and Manohar, R. (2004). Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, pages 17–27.

Theobald, M. and Nowick, S. (2001). Transformations for the synthesis and optimization of asynchronous distributed control. In *Design Automation Conference, 2001. Proceedings*, pages 263 – 268.

Tugsinavisut, S., Su, R., and Beerel, P. A. (2006). High-level synthesis for highly concurrent hardware systems. *Application of Concurrency to System Design, International Conference on*, pages 79–90.

Williams, T. E. (1991). *Self-timed rings and their application to division*. PhD thesis, Stanford University, Stanford, CA, USA. UMI Order No. GAX92-05744.

Williams, T. E., Horowitz, M., Alverson, R. L., and Yang, T. S. (1987). A self-timed chip for division. In Losleben, P., editor, *Advanced Research in VLSI*, pages 75–95. MIT Press.

Wilson, T. C., Mukherjee, N., Garg, M. K., and Banerji, D. (1995). An ilp solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis. *VLSI Design 3*, 1:21–36.

Wong, C. G. and Martin, A. J. (2001). Data-driven process decomposition for the synthesis of asynchronous circuits. In *IEEE Int. Conf. on Electronics, Circuits and Systems*.