Analysis and Optimization for Pipelined Asynchronous Systems

Gennette D Gill

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
2010

Approved by:

Montek Singh, Advisor

Jo Ebergen, Co-principal Reader

Krishnendu Chakrabarty, Reader

Anselmo Lastra, Reader

Leandra Vicci, Reader

Kevin Jeffay, Reader

Abstract

Gennette D Gill: Analysis and Optimization for Pipelined Asynchronous Systems . (Under the direction of Montek Singh.)

Most microelectronic chips used today—in systems ranging from cell phones to desktop computers to supercomputers—operate in basically the same way: they synchronize the operation of their millions of internal components using a clock that is distributed globally. This global clocking is becoming a critical design challenge in the quest for building chips that offer increasingly greater functionality, higher speed, and better energy efficiency.

As an alternative, asynchronous or "clockless" design obviates the need for global synchronization; instead, components operate concurrently and synchronize locally only when necessary. This dissertation focuses on one class of asynchronous circuits: application specific dataflow systems (*i.e.* those that take in a stream of data items and produce a stream of processed results.) High-speed stream processors are a natural match for many high-end applications, including 3D graphics rendering, image and video processing, digital filters and DSPs, cryptography, and networking processors.

This dissertation aims to make the design, analysis, optimization, and testing of circuits in the chosen domain both fast and efficient. Although much of the groundwork has already been laid by years of past work, my work identifies and addresses four critical missing pieces: i) fast performance analysis for estimating the throughput of a fine-grained pipelined system; ii) automated and versatile design space exploration; iii) a full suite of circuit level modules that connect together to implement a wide variety of system behaviors; and iv) testing and design for testability techniques that identify and target the types of errors found only in high-speed pipelined asynchronous systems.

I demonstrate these techniques on a number of examples, ranging from simple applications that allow for easy comparison to hand-designed alternatives to more complex systems, such as a JPEG encoder. I also demonstrate these techniques through the design and test of a fully asynchronous GCD demonstration chip.

Acknowledgments

This dissertation would not have been possible without the help of a great many people. I credit Montek Singh—who has been my advisor through this entire process—with helping me become interested in asynchronous design. He has always been there to point me in the right direction when I get off track and help me see the big picture when I was stuck on the details.

I also highly value the help and support of the other students that I have worked with. John Hansen and I learned together how tedious the hand design of circuits can be, and this experience has inspired both of our research. Ankur Agiwal and Vishal Gupta both provided invaluable support through setting up and running simulations.

I would like to thank my committee for the time that they have spent helping prepare me to complete this dissertation. Jo Ebergen, Leandra Vicci, Anselmo Lastra, Krishnendu Chakrabarty, and Kevin Jeffay have all helped test my knowledge and given feedback to prepare me for this research undertaking. Jo Ebergen, who I did an internship with soon after I began my journey into asynchronous design, helped shape my ideas about both the benefits and the challenges of asynchronous design. Leandra Vicci, in addition to being on my committee, also played a vital role in the completion of the testing and demonstration setup for the Euclid test chip.

I would like to mention the administrative and IT support staff in the computer science department, who have always been so helpful and patient with me. I would especially like to thank Janet Jones for doing a spectacular job of making sure that I was never kicked out of school for missing paperwork.

I would be remiss if I failed to mention the funding sources that have helped me focus on my research, rather than on paying the bills. I received the NDSEG fellowship from the

Department of Defense and also a National Science Foundation fellowship. During my last semester, which I spent preparing this dissertation and writing one more paper, I was supported by the Alumni Fellowship from Computer Science department at the University of North Carolina at Chapel Hill.

Though I cannot give names, I would love to be able to thank the countless anonymous reviewers who have spent the time to critique my papers over the years. It has helped me to greatly improve my scientific writing. Some of the writing from those papers has also found its way into this dissertation, so this work was indirectly edited by many people who undoubtedly do no realize their role. One person who has helped in editing this document that I can name, though, is my friend Sharif Razzaque, who has always had a knack for being helpful.

Finally, I must admit that there is no possible way I could have completed this work without the support of my husband, Pepe Bastardes, who has always been an unending source of optimism about my research. I find that, in research, motivation and tenacity are an even more vital commodities than ideas and breakthroughs. Pepe has been my motivation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Goals and Solution Strategies

The goal of my work is to support the design of high-performance application specific dataflow systems (*i.e.* those that take in a stream of data items and produce a stream of processed results.) The ability to design such processors quickly and efficiently will likely be important to sustaining the current level explosive growth of consumer electronics, multimedia applications, and high-speed networking. My work identifies and addresses several key challenges in designing these systems.

1.1.1 Challenges to Designing Application Specific Dataflow Systems

One particular challenge is that application specific processors have short design cycles, because bringing new application specific hardware to market quickly is vital when improved algorithms and new standards are constantly being created.My work handles this challenge in several ways. First, it employs the benefits of asynchronous or "clockless" design. The removal of the clock improves the circuit's efficiency by eliminating the power requirements of propagating a global clock signal. Further, the modular nature of asynchronous design can increase designer productivity and therefore reduce production time and the associated production costs. Next, this dissertation presents both fast performance analysis and automated

system-level optimization for asynchronous dataflow systems. These methods aim to decrease design times both by improving the speed of the algorithms themselves and by automatically and systematically completing tasks that designers often spend long hours performing in ad hoc ways.

Another specific challenge is providing low-level support for the design of high-performance dataflow systems. Designers want to have some assurance that the systems they create will operate correctly and be practical to implement at the circuit level. This dissertation presents new testing approaches that are specific to asynchronous pipelined systems, which can expose errors caused by manufacturing defects. My work also includes a suite of new circuit-level designs, which can be connected together modularly to implement a wide variety of system functionality.

1.1.2 Solution Strategy

In order to make the problem of analyzing and optimizing pipelined asynchronous circuits more tractable, I focus on a special class of asynchronous systems: those with *hierarchically composed pipelined architectures.* Such structures are quite common when the system is designed using high-level translation methods (*e.g.*, Tangram/Haste [45], Balsa [14]) because high-level specification languages tend to be hierarchically block-structured. Moreover, even when the design approach is ad-hoc, designers often tend to implement systems with hierarchical structures. In particular, I target architectures that are hierarchical compositions of sequential, parallel, conditional, and iterative operators. By focusing on this special but practically useful class of systems, this work is able to leverage information about their hierarchy to provide fast runtimes while still maintaining accuracy.

1.1.3 Thesis Statement

The problem of analyzing and optimizing pipelined asynchronous systems can be made tractable by restricting the domain to hierarchically constructed systems.

The key insight is that restricting to the hierarchical nature of these systems can be exploited to make analysis and optimization faster. This dissertation will show that the set of hierarchical systems is expressive enough to implement a variety of practical systems. The results will also show that the resulting analysis method is fast and accurate enough to be used repeatedly in an optimization loop. Finally, the resulting, optimized systems can be practically realized in hardware.

1.2 Introduction to Asynchronous Design

1.2.1 Definition

Figure 1.1 illustrates the fundamental difference between asynchronous design and synchronous design. While the synchronous system has one global clock that must propagate to the entire chip, the asynchronous design uses handshakes to synchronize local communications. The switch from global clocking to local, as-needed synchronization can potentially offer ease of design due to increased modularity, faster speed because of more concurrency, and lower energy consumption because switching activity occurs only when and where needed.

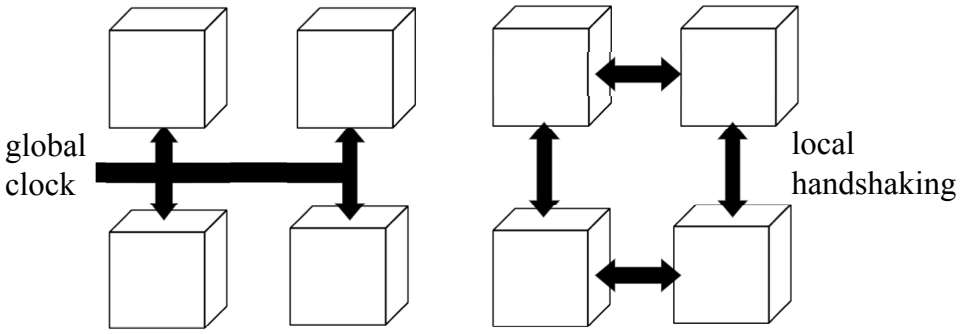


Figure 1.1: Comparison between synchronous and asynchronous design paradigms

1.2.2 Current Uses

As an indicator of asynchronous design's current commercial success, Handshake Solutions, a Philips subsidiary, has sold more than 750 million asynchronous microprocessor chips for use in low-power applications such as cell phones, pagers, smart cards, and cryptochips used in E.U. passports. In addition, Intel and Sun Microsystems have used asynchronous circuits in small targeted speed-critical circuits inside their commercial processors.

1.2.3 Future Applications

Asynchronous design will have a significant role in the design of future systems. In future multi-core systems, on-chip networks that connect hundreds or thousands of heterogeneous cores will likely be asynchronous in nature, leading to a globally asynchronous locally synchronous (GALS) architecture. Recently, a new style of clocked design has received much attention from researchers: synchronous elastic design or latency-insensitive synchronous design. These styles borrow the notion of elasticity (achieved via asynchronous handshaking) and apply it to clocked design to obtain synchronous elastic systems. Finally, asynchronous design will likely play a key role in the emerging technologies of nanoelectronic, quantum and biologically-inspired (e.g., DNA self-assembled) computing, in all of which timing issues are fundamentally too complex to be addressed by global clocking.

1.3 Past Work and Current Challenges

The overall goal of my line of research is to create a fully automated toolflow for the design of high-throughput asynchronous circuits that begins at a designer specification and ends with a gate-level hardware specification. Figure 1.2 shows an example design flow, which gives context for the work presented in this dissertation. If fully automated, this design flow will achieve the goals of creating efficient, high-performance circuits with low designer-time cost. This section goes step-by-step through the design flow to explain how each step helps achieve

the overall goals, describe the current work that has been done in each area, and outline the remaining work that this dissertation will add to the design flow.

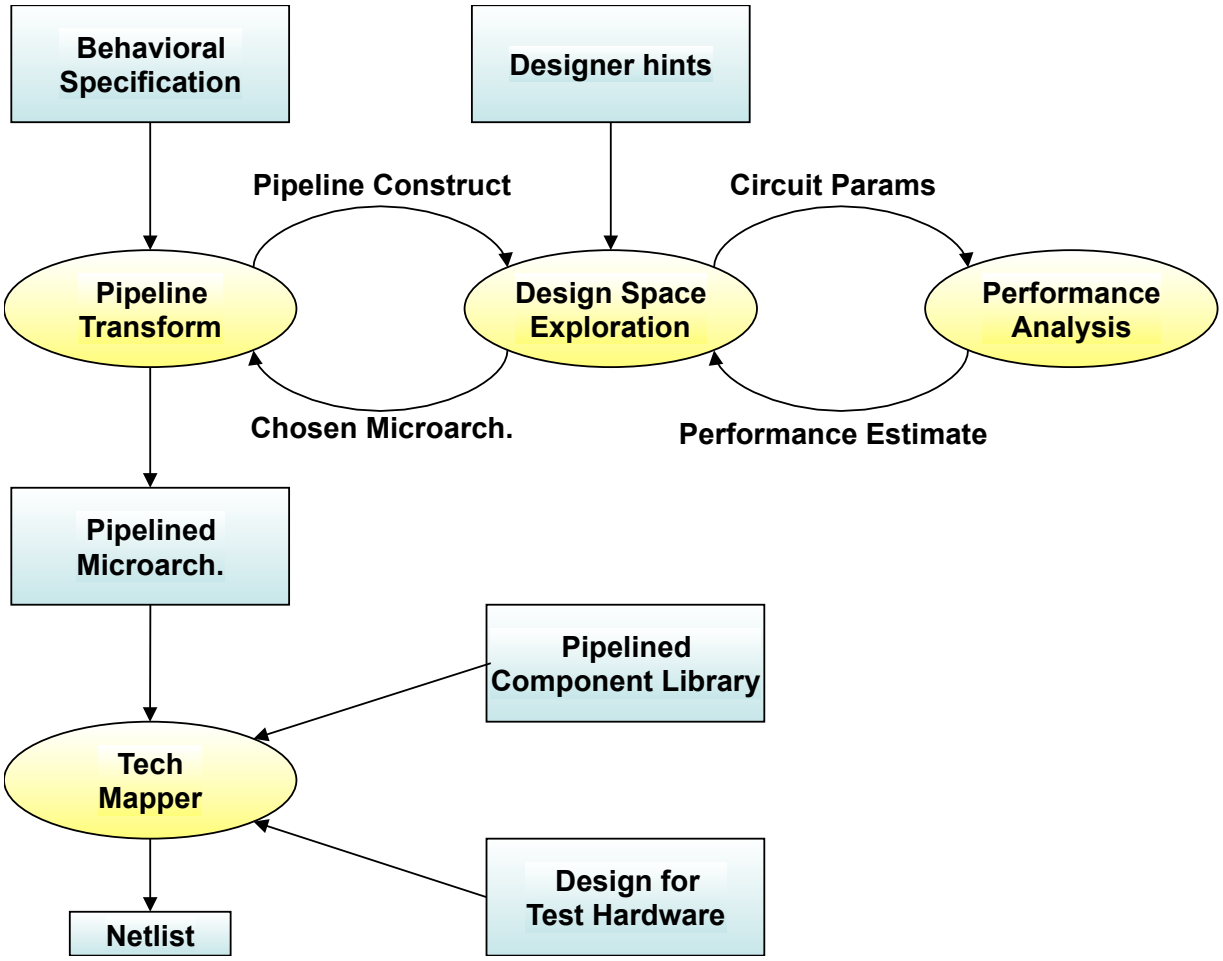


Figure 1.2: The ideal design flow to reduce designer effort.

1.3.1 High-level Code to Low-level Designs

In the design flow of Figure 1.2, the designer first gives a high-level behavioral specification for the circuit. Next, that control-driven specification is turned into a data-driven pipelined architecture. My work does not focus on this transformation step, because much current and ongoing work done by others has effectively targeted this area. Specifically, Hansen [19] has used Haste [45] as the input language for creating a data-driven system from a high-level

specification. Work done by Budiu [8] used C as the input language to create a hardware specification in which data flows directly from producer to consumer. Kapasi takes a different approach to specifying data flow hardware by introducing new language constructs that are tailored to data flow design [24]. Though Chapter 3 of this dissertation does give one possible algorithm for converting a specification written in a high-level language to a pipelined system, this is offered only for completeness and does not limit the work of this dissertation to using only that one approach for the first pipeline transform step of the design flow.

1.3.2 Design-space Exploration

Automated design-space exploration is key to reducing designer effort. In the design flow of Figure 1.2, design space exploration takes a non-optimized pipelined circuit as an input and returns an optimized version of the circuit. Since my work targets high-throughput systems, the final circuit should be optimized to meet some throughput goal, possibly with additional constraints on other cost functions such as delay or energy. The design space exploration should ideally take place with little to no direct intervention on the part of the designer. Currently, many circuit designers use time consuming ad hoc techniques based on designer knowledge and experience. More systematic approaches presented by Beerel *et al.* [3], Prakash *et al.* [47], and Smirnov *et al.* [1] remove bottlenecks by strategically adding buffer stages to the circuit. Other approaches target latency rather than throughput. One such approach is the CASH compiler by Budiu and Goldstein [8], which translates an ANSI C program into data-flow hardware. A different approach by Theobald and Nowick [63] targets generation of distributed asynchronous control from control-dataflow graphs, with the objective of optimizing communication between the controllers, and between a controller and its associated datapath object. To my knowledge, a system-level design space exploration system that focuses on throughput does not currently exist. My work on constrained optimization for pipelined systems aims to fill this gap in the currently offered set of optimization solutions.

1.3.3 Performance Analysis

Performance analysis is an integral part of design space exploration, because optimization often requires repeated analysis to monitor the effects of changes to the circuit. As seen in Figure 1.2, the design space exploration method uses performance analysis repeatedly, which indicates that fast and accurate performance analysis is important to making the design exploration work. Several innovations in the area of performance analysis of pipelined circuits serve as the basis of the work of this thesis. The idea that the throughput depends on the occupancy of a pipeline, which is the foundation of much of the analysis-level work of this thesis, was first introduced in a doctoral thesis by Williams [67] and later expanded upon by Greenstreet [18] and Lines [31], though none of these works provided a system-level approach. In addition, more general analysis methods [29][71] [36] can estimate the performance of entire systems, proving that accurate system-level throughput estimates can be obtained for pipelined asynchronous systems. Brining these ideas together, my analysis work provides a fast analytic solution like that of Williams [67] while maintaining some of the versatility found in past system-level analysis methods.

1.3.4 Circuit Components

The design space exploration step produces an optimized pipelined microarchitecture. To be realized in practice, every component within the microarchitecture must be implemented by a corresponding circuit component. Many pipeline circuit styles have already been developed, which serve as the basis for implementing any high-throughput pipelined system.[61, 15, 42, 59, 55, 72] Micropipeline styles that achieve high-speed while maintaining low control overheads enable the construction of fast and efficient asynchronous pipelines. My work in this area adds on to MOUSETRAP, a high-speed, low-overhead pipeline implementation [55]. Existing MOUSETRAP circuits can be used to implement a variety of structures, but are not expressive enough to implement all the possible behavior in the set of circuits that the design flow can produce. My work fills that gap by introducing five new pipeline circuit elements that

can, together with existing circuits, implement common hierarchical structures. These circuits can be composed together modularly to form a wide variety of structures and behaviors.

1.3.5 Testability

Finally, the design flow incorporates the design for testability. Incorporating testing into a design flow is vital to the success of any design flow, because designers are hesitant to use a technology that cannot be tested. Luckily, much work has already been done in the area of testing for asynchronous pipelines. Pagey *et al.* [43] proposed methods for generating test patterns to test stuck-at faults in traditional micropipelines. Another approach that was adapted for use with micropipelines is by [27], which focuses on test sequence generation for testing both stuck-at and delay faults inside *C-elements*—circuits that are often used as local synchronizing elements in asynchronous systems. My work on testing is based most closely on the work of Shi *et al.* [53], which is specifically targeted to testing high-speed asynchronous pipelines. My testing work focuses on achieving high fault coverage while trying to keep the design-for-test overheads low.

1.4 Contributions

The work of this thesis focuses on one type of asynchronous circuits—high-throughput steaming systems that are hierarchical in nature—in order to provide a great depth of resources for designing these types of circuits. Contributions range from high-level abstractions for circuit modeling to low-level circuits that are used as the basis for circuit design.

1.4.1 Modeling and Abstractions

Chapter 3 presents the models and abstractions that are used throughout the rest of the dissertation. In particular, it gives a method for obtaining the local performance metrics of a

system, defines the structure of a hierarchical system, and formalizes use of canopy graphs for performance analysis.

1.4.2 Performance analysis

Chapter 4 addresses the problem of system-level performance estimation. Although determining the performance of an arbitrary asynchronous system is NP-hard, my solution focuses instead on analyzing a special subclass: systems that are hierarchically composed of basic pipeline stages using sequential, parallel, iterative, and conditional operators. This restricted class is actually still quite rich, and includes most commonly used system topologies. By exploiting hierarchy, my algorithm achieves an expected linear running time. Results show that non-trivial examples with over 100 stages can be analyzed in less than 10 milliseconds.

1.4.3 System-Level Optimization

Chapter 5 presents optimization methods that leverage the fast analysis method presented in Chapter 4. Identification and elimination of bottlenecks is an important part of any design flow that aims to produce high-throughput systems. This thesis includes (i) a bottleneck detection algorithm that leverages performance analysis to pinpoint the parts of an asynchronous system that limit throughput, and (ii) a bottleneck alleviation algorithm that reports all potential bottlenecks to the designer and suggests modifications. A specific challenge is to overcome the performance bottleneck due to algorithmic loops. I have introduced a systematic approach for improving the throughput of algorithmic loops by allowing multiple problem instances to be computed concurrently within a single loop hardware (i.e., multi-token loops).

1.4.4 Testing and Design for Testability

Chapter 6 presents a testing approach to uncover timing violations within asynchronous pipelines. While the general problem of delay fault testing has challenged researchers for decades, my

approach targets a subclass of much interest to asynchronous designers: relative timing constraint violations. In particular, certain asynchronous circuits achieve high performance by making local timing assumptions. I have introduced the first practical method for testing for violations of these relative timing assumptions in pipelined systems. The approach is minimally-intrusive and includes both test application and test vector generation.

1.4.5 Circuit-Level Design

Chapter 7 presents the low-level pipeline stages that form the basis for implementing pipelined asynchronous systems. My component library includes circuit-level designs of all commonly used building blocks (i.e., design primitives), including pipeline stages, forks/joins, conditionals and loops, arbitration stages, etc. These designs were carefully constructed so as to obtain high throughput and low latency.

1.4.6 Case Studies

To demonstrate and evaluate these circuits, Chapter 8 presents asynchronous case study chip that implements Euclids iterative GCD algorithm; it was fabricated at IBM foundry in a 130nm CMOS process. The chip is one of only a few high-speed asynchronous chips that have been demonstrated so far, achieving the equivalent of 1 GHz operation, and robustness to wide variations in supply voltage (0.5-4V) and temperature (-45-150C), and has helped validate my design and test approaches. In addition, the chapter uses the analysis and optimization of a JPEG encoder as a detailed example of the steps taken to produce a final, optimized circuit. The results show a 40x improvement for this example.2

1.5 Dissertation Organization

The remainder of this document is organized as follows. First, Chapter 2 gives background information on performance analysis, optimization, and testing for asynchronous systems.

Next, Chapter 3 describes the abstractions and models that will be used throughout this dissertation. It also introduces new notations and definitions relevant to asynchronous design that are extensions upon previous work.

Chapter 4 introduces my analysis method. It first presents the analysis of the four hierarchical components considered in this dissertation—sequential, parallel, conditional, and iteration—and then describes how to bring them together to form a system-level, hierarchical analysis method. Chapter 5 goes on to present an optimization method that builds off of the analysis. It first describes a set of transformations that can potentially improve performance, then gives a method for detecting the location of bottlenecks within a system, and finally defines a framework for iteratively applying optimizing transformations to remove all the bottlenecks of a system.

Chapter 6 describes a testing method that applies to pipelined asynchronous systems. Chapter 7 provides low-level support for design by specifying a suite of five circuits that—together with circuits from past work—can be used to implement a wide variety of asynchronous systems. This chapter can serve as a reference for those who wish to implement any of the hierarchical structures described in this dissertation. Finally, Chapter 8 gives two case studies in asynchronous design. The first one, a system that calculates the greatest common divisor of two numbers, was fabricated in silicon and tested extensively. It serves as an example of the kind of performance that one can expect from pipelined asynchronous systems.

Chapter 2

Background on Asynchronous Design

2.1 Asynchronous Pipelines

In a pipelined asynchronous system, parts of the circuit synchronize locally using a request-acknowledge handshake protocol. These handshakes control the flow of data into and out of registers, which are usually implemented as latches. A pipeline operates correctly if the handshaking protocol prevents data from being lost or overwritten while still allowing for the flow of data between stages.

Figure 2.1 shows the basic structure of a self-timed pipeline. Each pipeline stage consists of a controller, a storage element ("data latch"), and processing logic. Typically, a stage generates a *request* to initiate a handshake with its successor stage, indicating that new data is ready. If the successor stage is empty it accepts the data and performs two further actions: (i) it acknowledges its predecessor for the data received and, (ii) initiates a similar handshake with its own successor stage. Many variations on this basic protocol have been developed, and a variety of asynchronous pipeline controller implementations exist.

2.1.1 Data Encodings within Asynchronous Pipelines

Pipeline styles can also be categorized based on the data encoding employed. Using data encodings to improve robustness or reduce energy consumption is an active area of research

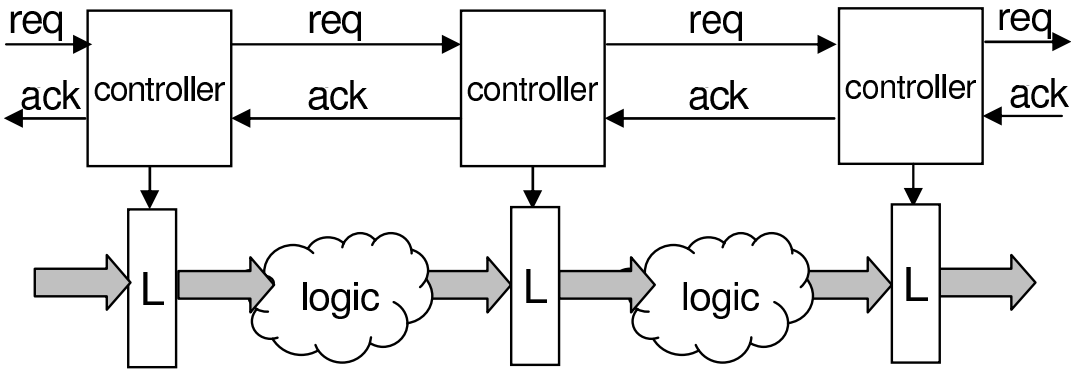


Figure 2.1: A simple self-timed pipeline

[38, 34], so this section will not provide a full description of possible data encodings. Instead, it will give some idea of the main types of data encodings used.

Bundled Data

A simple yet effective approach is the bundled data encoding, which is also known as single-rail. Each data bit is represented by exactly one wire, with one additional wire for the request bit. While the data bits are processed by a block of logic within the stage, the request bit must go through some *matched delay*—a set of gates that slows the outgoing request so it stays coherent in time with the data. This style is simple and practical, but it does not take advantage of function blocks that have variable delays.

Dual-Rail Encoding

Dual-rail data encoding uses two wires per data bit, so that the *request* signal is essentially built in to the encoding. Table 2.1 shows one possible dual-rail representation that encodes both the value of the data and the presence of new data.

This encoding scheme allows the stage to indicate completion as soon as it is finished, though detecting this completion can add extra overhead.

Table 2.1: Dual rail data encoding.

Dual-rail		code
0	0	data not ready
0	1	0 request
1	0	1 request
1	1	unused

2.1.2 Asynchronous Handshaking Protocols

two-phase Protocols

Two phase protocols have handshaking cycles that are made up of two events. Specifically, the value of the request changes once and then the value of the acknowledge changes once to complete the handshake. As a result, 2-phase protocols use transition based encodings: a transition on a line rather than the value of the line indicates an event.

four-phase Protocols

Four phase protocols have handshaking cycles that are made up for four events. Typically, the request goes high and the acknowledge goes high as a response. Then the request goes low to a reset value, and the acknowledge also resets to a low value. This protocol has more events than a two-phase protocol, but the logic implementations are often simpler, because level-based logic is a simpler design style than transition-based logic.

2.1.3 Asynchronous Pipeline Styles

Many different implementations for asynchronous pipelines have been developed, which use different combinations of data encodings and handshaking protocols. [61, 15, 42, 59, 55, 72, 67] In this dissertation, I use the following four circuit styles as examples whenever the circuit-level details of a pipeline stage need to be demonstrated.

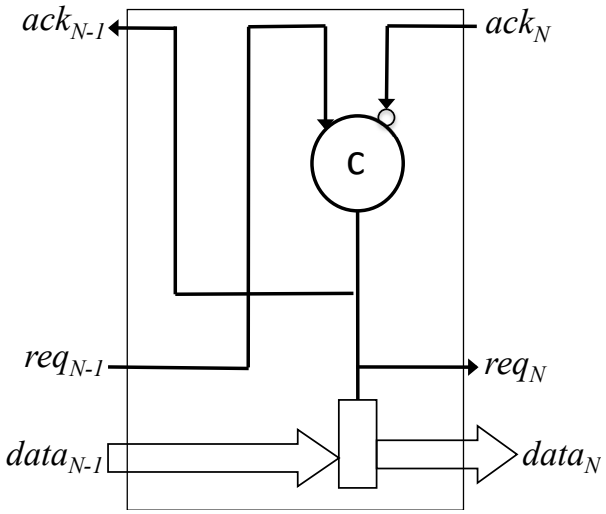


Figure 2.2: Sutherland's Micropipeline with Muller C-element.

Sutherland's Micropipelines

Suterland's Micropipeline style [61] is a two-phase controller that uses bundled data. As shown in Figure 2.2, it uses the Muller C-element [40] as the control circuit. Since the output of the C-element to the latch uses transition signaling, the latch is a special transition sensitive latch. When the pipeline line is in operation, every transition on the request wire—either high to low or low to high—indicates that a new data is available. The next stage will send back a corresponding acknowledge signal when it has finished processing the previous data.

MOUSETRAP Pipelines

MOUSETRAP [55], shown in Figure 2.3, is a two-phase pipeline style that uses static logic. Since MOUSETRAP uses transition signaling for the requests and acknowledges, every transition on a request wire indicates new data is ready and every transition on an acknowledge wire indicates that old data can be overwritten. Thus, when the request and acknowledge going into a stage are the same, the stage becomes "empty" and when they are different the stage becomes "full". The latches that hold data begin transparent and become opaque just after new data arrives. The latches themselves do not use transition signaling, since the *xnor* acts

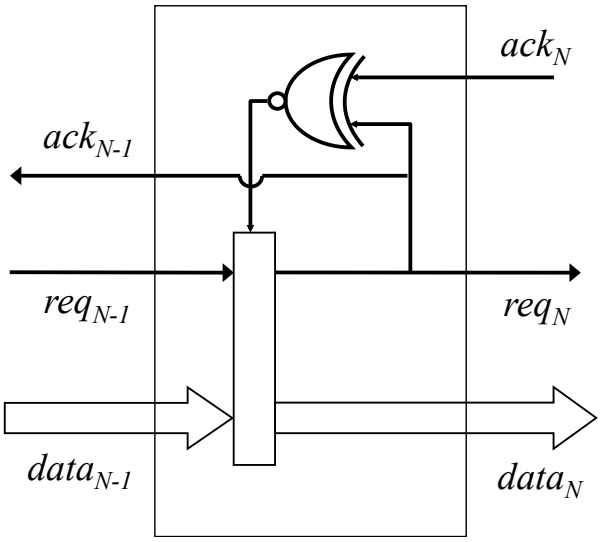


Figure 2.3: The MOUSETRAP pipeline style.

as a transition to level converter.

GasP Pipelines

GasP [59] is a four-phase pipeline style that uses static logic. The data latches in a GasP pipeline begin closed, and must open and then close again upon receiving each new data item. The most distinctive feature of GasP is that a single wire, called the state conductor, is used to transmit *both* the request and acknowledge signals between a pair of adjacent stages. A low signal on the state conductor wire indicates that the previous stage is"full" and a high signal indicates that it is "empty".

The controller for GasP is a self resetting NAND, as shown in Figure 2.4. When both of the inputs to the NAND go high, it goes low. After some delay, this low transition triggers the NAND to reset back to high. Specifically, two possible reset paths, r_1 and r_2, can cause the NAND to go high again. Path r_1 consists of one pull up transistor and one inverter (inv_a). Path r_2 consists of one inverter (inv_b) and one pull down transistor. The time, $t_{\rm reset}$, to reset the NAND is the *smaller* of the two times: $t_{\rm reset} = min(t_{\rm pull\_up} + t_{\rm INVa\downarrow}, t_{\rm INVb\uparrow} + t_{\rm pull\_down})$.

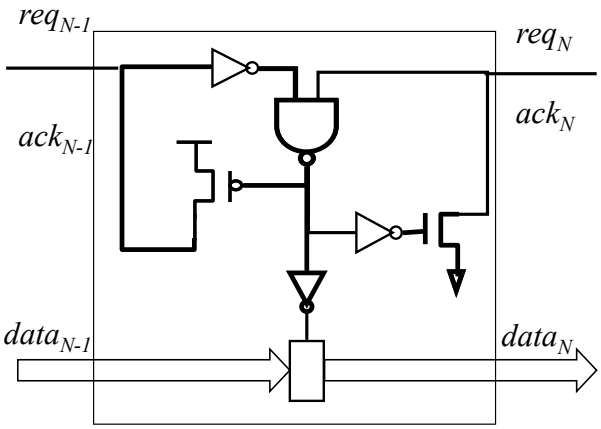


Figure 2.4: The GasP pipeline style.

High-Capacity Pipelines

The high-capacity (HC) pipeline [54] is a four-phase pipeline style that uses dynamic logic. It is *latchless*. Instead, the dynamic logic of each stage has an "isolate phase", in which its output is protected from further input changes. Specifically, during the isolate phase, the logic is neither precharging nor evaluating.

An HC pipeline stage simply cycles through three phases. After it completes its evaluate phase, it enters its isolate phase and subsequently its precharge phase. As soon as precharge is complete, it re-enters the evaluate phase again, completing the cycle.

2.2 Performance Analysis for Asynchronous Pipelines

2.2.1 Performance Metrics for Pipeline Stages

There are three metrics typically used to characterize the performance of a self-timed pipeline: *forward latency*, *reverse latency*, and *cycle time.* The forward latency is simply the time it takes one data item to flow through an initially empty pipeline. Thus, if the latency of Stage$_i$

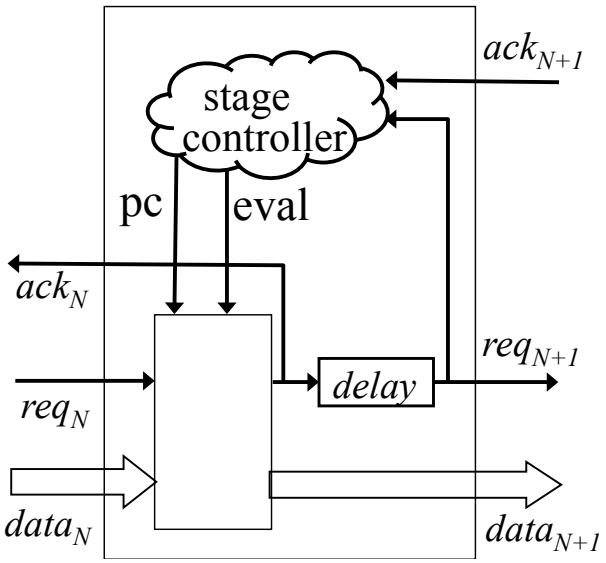


Figure 2.5: The high capacity pipeline style.

is F_i, then the latency of the entire pipeline is:

$$L_{Pipeline} = \sum_i F_i \tag{2.1}$$

Similarly, the reverse latency characterizes the speed at which empty stages or "holes" flow backward through an initially full pipeline. The reverse latency of the entire pipeline is simply the sum of the stage reverse latencies:

$$R_{Pipeline} = \sum_i R_i \tag{2.2}$$

The cycle time of a stage, denoted by T_i, is the minimum time that must elapse after a data item is produced by that stage before the next data item will be generated. Since a complete cycle of a stage typically consists of transmitting a data item forward followed by propagating a hole in the reverse direction, the following relationship holds for most handshaking protocols:

$$T_{Stage_i} = F_i + R_i \tag{2.3}$$

The maximum throughput a stage can support is simply the reciprocal of its cycle time. The cycle time of a linear pipeline is limited by the cycle time of its slowest stage:

$$tpt_{Pipeline} \;=\; 1 \Big/ \max_i(T_i) \tag{2.4}$$

2.2.2 Canopy Graph Analysis

The classic work on performance analysis of asynchronous pipelines by Williams and Horowitz [67] introduces the "canopy graph" to characterize the performance of a pipelined ring. Our approach builds upon this to handle more general systems.

Ring Performance Analysis

One useful metric of ring performance is the number of times any data item crosses a particular stage in the ring per second; we will refer to this measure as the *ring frequency.* The performance of the ring is highly dependent on its *occupancy, i.e.,* the number of data items revolving inside it. When the number of data items is small, the ring frequency is low, and the pipeline is said to be "data limited." On the other hand, when nearly every stage of the pipeline is filled with data items, the performance is once again limited, because holes are needed to allow data items to flow through the pipeline. In this scenario, the pipeline is said to be congested, or "hole limited."

Data Limited Operation. If there are k data items in the ring, then in the time a particular data item completes one revolution around the ring (*i.e.,* $\sum_i F_i$), all k items would have crossed each ring stage. Therefore, the maximum ring frequency attainable is proportional to the ring occupancy:

$$\textit{Ring Frequency} \;\leq\; k \Big/ \sum_i F_i \tag{2.5}$$

Hole Limited Operation. If the ring is filled with data items in nearly all stages, then the

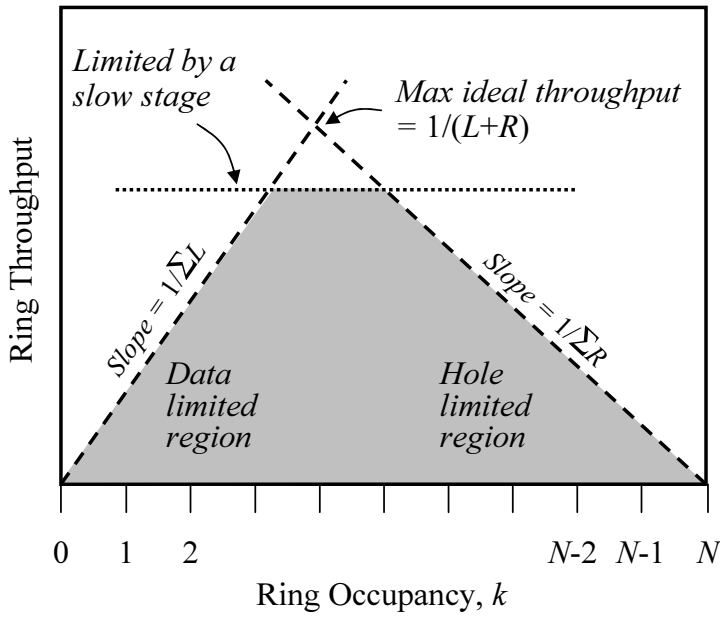


Figure 2.6: The upper bounds on the maximum ring frequency: shaded area represents operating region

ring frequency is limited by the number of holes in the ring. If there are h holes in the ring, then in the time a particular hole completes one revolution around the ring (*i.e.,* $\sum_i R_i$), all h holes would have crossed the each stage, 0 in a direction opposite to data. Thus, if N is the number of stages in the ring, then $h = N - k$, and we have the following bound on the performance:

$$\textit{Ring Frequency} \quad \leq \quad (N-k) \Big/ \sum_i R_i \tag{2.6}$$

Figure 2.6 shows a plot of the ring frequency versus its occupancy. The rising portion of the graph represents the data limited region, where performance rises linearly with the number of data items. The falling portion, similarly, represents the hole limited region, where performance drops linearly with a decrease in the number of holes.

Limitations Due to a Slow Stage. If all the stages in the ring have similar forward and reverse latencies, then the maximum attainable performance will be the frequency at which the rising and the falling lines of Figure 2.6 intersect. This point represents a frequency that

is the inverse of the cycle time of each ring stage: $1/T = 1/(L + R)$. However, if some stages are slower than others, then the ring frequency will be limited by the cycle time of the slowest stage. In the figure, the horizontal line represents the maximum operating rate that can be sustained by the slowest stage in the ring [67]:

$$\textit{Ring Frequency} \leq 1 \Big/ \max_i(T_i) \tag{2.7}$$

The overall ring performance will always be constrained to lie under the canopy formed by the three lines in Figure 2.6.

Linear Pipelines

The behavior of a linear pipeline, *under steady-state operation,* can be modeled as that of a self-timed ring. In particular, in steady state, as one item leaves the right end of the pipeline, another item enters on the left. As shown by Lines [31] and Singh *et al.* [57], the linear pipeline's throughput is correctly modeled as a canopy graph, with the same three constraints on its operation (data limited, hole limited, and constrained by local cycle times).

However, there are two key differences that must be noted. First, the occupancy in a linear pipeline can be fractional. That is, unlike a ring which always contains an integral number of items, a linear pipeline can have an *average* occupancy that is non-integral (*e.g.*, 4.5 items) because of phase differences between entering and exiting items.

Second, a linear pipeline can actually operate in the *entire region under the canopy.* In contrast, the operating point of a ring in steady state is typically on the boundary of the canopy. The reason for this difference is that the ring analysis assumed that the ring is isolated and operates autonomously without any interaction with the environment; therefore, it operates at the maximum throughput possible for a given occupancy. On the other hand, a linear pipeline's operation is constrained by both the left and right environments, which can force it to operate at a throughput below the maximum attainable throughput for a given occupancy.

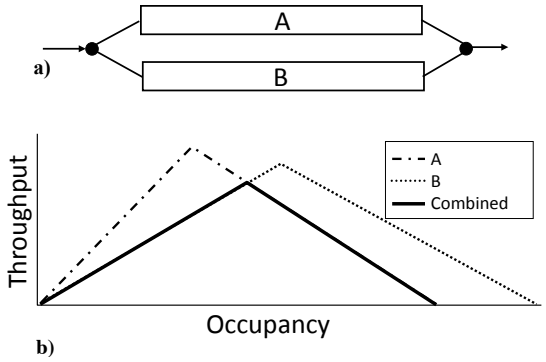


Figure 2.7: Parallel composition: a) structure, b) canopy graphs

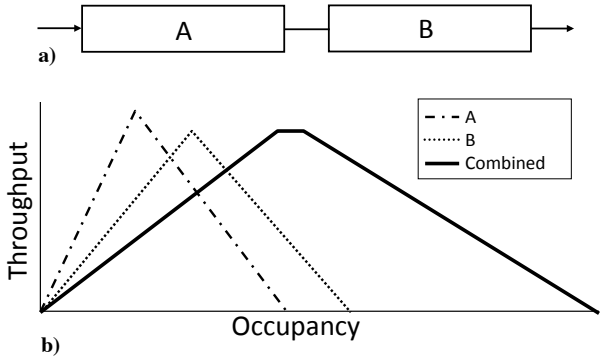


Figure 2.8: Sequential composition: a) structure, b) canopy graphs

Parallel and Sequential Composition

As shown by Lines [31], canopy graph analysis can also be applied to parallel and sequential compositions of linear pipelines.

For the fork-join parallel structure of Fig. 2.7a, the throughput of the composition is constrained by the *intersection of the canopy graphs* of the branches. The reason is that the operation of the fork-join pair is constrained so that (i) the throughput of each branch is the same, and (ii) the number of tokens in each branch is the same (assuming they were initialized empty). The result of the intersection of the constituent canopy graphs is also a canopy graph as shown in Fig. 2.7b.

Similarly, when two pipelines are composed sequentially, as in Fig. 2.8a, the throughput of the composition is constrained by the *horizontal sum of the canopy graphs* of the two constituents. The reason is that, once again, the throughput of each pipeline must be the same,

but the total occupancy is now the sum of the occupancies of the two pipelines. That is, the composition can attain any throughput that both of the pipelines can sustain, and for each such throughput, the net occupancy is simply the sum of the two occupancies. The result is also a canopy graph as shown in Fig. 2.8b.

Chapter 3

Models and Abstractions

Throughout the thesis, most of the discussions and proofs will deal not with the complex, underlying circuits but with the more abstract models. This chapter defines the models that will be used throughout this thesis and defines how they relate to real systems. First Section 3.2 gives the models for the performance of an individual stage and presents a method for mapping from a pipelined circuit to the abstract representation used in later chapters. Section 3.3 fully defines the set of hierarchical circuits that this thesis analyzes, and gives an example of the steps from high-level code to a pipelined representation. Finally 3.4 defines and further described the use of the canopy graph for modeling system-level performance information.

3.1 Background

3.1.1 Performance Metrics for Pipeline Stages

Three performance metrics are typically used to describe the performance of pipeline stages: forward latency, reverse latency, and cycle time. Following are the classical descriptions of these three performance metrics, adapted from the textbook by Sparsø. [58]

Forward Latency

The forward delay, also denoted as L_F, is the difference between the time that a request is received by an empty stage and the time that a request is received by the following stage. Figure 3.1 illustrates the forward delay of one stage.

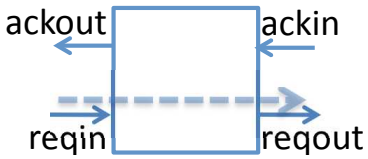


Figure 3.1: Forward delay for a pipeline stage

Reverse Latency

The reverse delay, also denoted as L_R, is the difference between the time that an acknowledge is received by a full stage and the time that an acknowledge is received by the previous stage. It is also often useful to think of the reverse delay as the time for one "hole" to travel in the reverse direction of the flow of data. Figure 3.2 illustrates the forward delay of one stage.

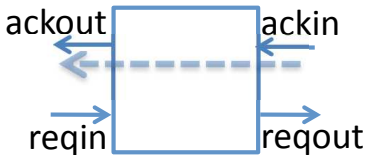


Figure 3.2: Reverse delay for a pipeline stage

Cycle Time

Cycle time, or T, is the time for a stage to progress from some state in relation to a data item until the time it arrives at the same state with the subsequent data item. Figure 3.3 shows a representation of the cycle time. Note that cycle time can cross stage boundaries, and that it does not necessarily include the entirety of either the forward or reverse cycle time for the stage.

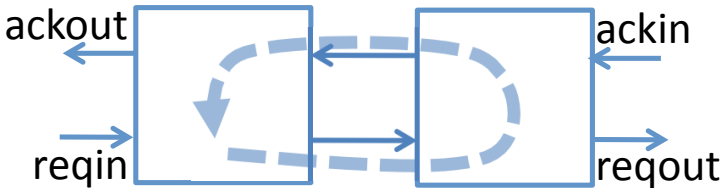


Figure 3.3: Cycle time for a pipeline stage

3.1.2 Abstractions for Asynchronous Pipelined Systems

One common representation for an asynchronous circuit is the Petri net, which has been used in much past work [30][36][71]. Petri nets are capable of capturing the high concurrency inherent to asynchronous systems. Moreover, timed petri nets, which assign a delay to every event within the system, allow for thorough analysis of the timing of the asynchronous system. [36].

The power of using a petri net model lies in its ability to represent all possible connections and configurations of a general asynchronous system. This kind of low-level modeling of connections is necessary when analyzing systems with complex connections and irregular topologies. The work in this thesis, however, focuses on systems that are composed of repeated instances of pipeline stages that are further organized hierarchically. For these purposes, a timed petri net representation is not necessary; the essence of the system can be captured by a more abstract, high-level representation.

3.1.3 System-level Performance Models

The most straightforward method for describing the performance of a pipelined system is to give one number: the throughput. Variations on the single-number performance characterization include reporting the cycle time, bounding the time separation of events, or finding the global critical cycle time for the system [36][30][70][65]. Each of these metrics provides a one-dimensional view of the performance of a circuit, in that a single performance number for the entire system is reported.

Williams and Horowitz [67] first introduced the use of a second dimension, occupancy, to characterize the performance of a system. Here, occupancy is defined to mean the total number of distinct data items that are contained in the entire pipelined system. This thesis relies heavily on the throughput vs. occupancy graph, also known as the canopy graph, first introduced for analyzing the performance of a pipeline ring. Later, Lines [31] and Singh *et al.* [57] applied this model to sequential pipelines, and Lines [31] further used it for systems consisting of a parallel operator. More details on the derivation and methods of this work can be found in the background section 2.2.2 of this thesis.

3.2 Modeling Asynchronous Pipeline Stages

As described in Section 2.1.3, many different types of asynchronous pipelines exist. My method for analyzing pipelines, however, is independent of the particular pipeline style used. This section describes how to map an actual asynchronous pipelined stage onto a representation that is independent of implementation by abstracting away the unnecessary details while retaining the necessary performance information.

3.2.1 Delay Models

The analysis method described in this thesis uses a fixed delay model. The commonly used bundled data encoding, which was described in Section 2.1.3, is very accurately modeled using fixed delays. Other data encodings, such as dual-rail, can have data-dependent execution times for individual stages. Though this work does not directly address probabilistic delays, it can still be used to perform worst case analysis by setting the fixed delays to be equal to the worst case delay for each stage. Worst case analysis is a useful lower bound on performance. [17]

3.2.2 Performance Metrics for System-Level Analysis

Figure 3.4 shows the abstract model for an asynchronous pipeline stage that will be used throughout the rest of this thesis. Each stage is represented by a box. This is a further abstraction of the pipeline stage shown in Figure 2.1, which removes the unnecessary details of the stage's controller, latch, and logic.

The connections between stages are represented by arrows. The direction of the arrows indicates the direction of the flow of data while the flow of "holes" is implied to be in the opposite direction. The internal workings and even the handshaking signals have been removed for simplicity and are indeed not necessary to perform analysis on this system.

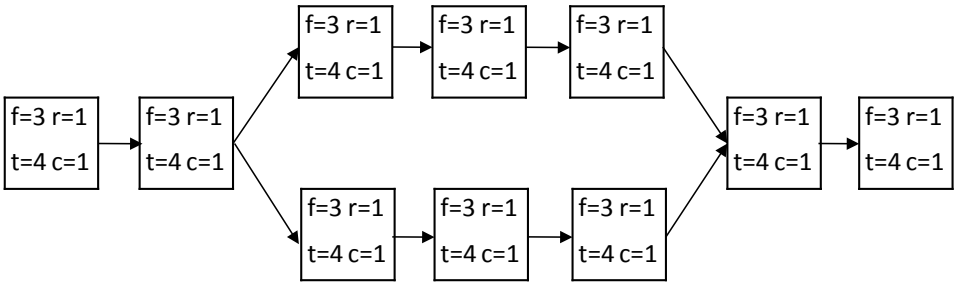


Figure 3.4: General model for a pipelined system.

Each stage is also annotated with the local performance metrics that are necessary for performing analysis: forward latency, reverse latency, cycle time, and capacity. These four metrics fully characterize the performance of each individual stage for the purposes of performing system-level analysis. This notation for graphically describing a pipelined system will be used throughout this thesis. For conciseness, the graphical representations often omit the cycle time and capacity, as has been done in Figure 3.5. If not specified, cycle time is assumed to be the sum of the forward and reverse latencies and the capacity is one.

This thesis uses the same definition of cycle time as give in background section 3.1.1. However, the definitions of forward and reverse latency are slightly changed, capacity is a newly defined term. The precise meaning of these terms is given in the following section.

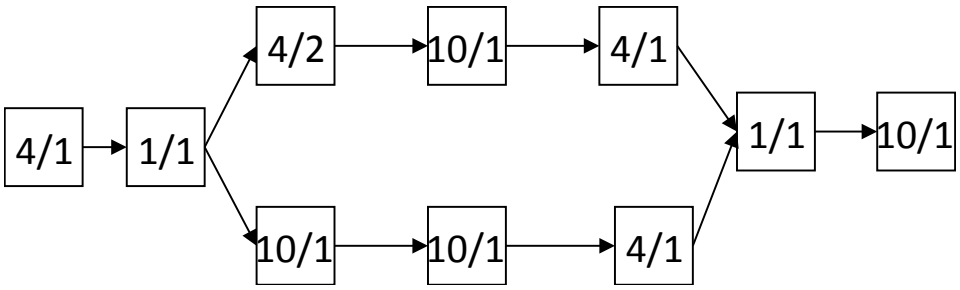


Figure 3.5: Model for a pipelined system used throughout this thesis.

Capacity

The capacity of a single stage to hold data is a feature of the chosen pipeline style. All commonly used pipeline styles have a capacity of either one or one half data items per stage, although other capacities are possible in theory. A capacity of one indicates that two stages in a row can both contain distinct data items while a capacity of one half indicates that every other stage can contain a distinct data item.

3.2.3 Determining Performance Metrics

Although the models used for analysis do not depend on the type of pipeline stage being used, determining the values of the four parameters for each stage–which must take place before analysis–does depend on the type of pipeline stage being used. In particular, one cannot always assume that cycle time for a stage be calculated using the forward and reverse delays only.

This section describes my technique for determining the necessary metrics. The remainder of the thesis assumes that these metrics have already been determined, and the accuracy of the performance analysis will, of course, depend on the accuracy of performance metric values for each stage.

Challenges to Determining the Performance of a Single Stage

A common method for finding the cycle time is simply summing the forward and reverse cycle times [67, 58]. Though this method is it simple and elegant, it does not always give an accurate local cycle time. The local cycle time often includes the delays of gates across two or more pipeline stages, so summing the forward and reverse latencies is inaccurate when gates in adjacent stages have differing delays.

As an example, see cycle time for the pipeline stage $S0$ shown in Figure 3.6. Compare the sum of the forward and reverse latencies of the stage to the cycle time.

$L_{F0} + L_{R0} = d_{L0} + d_{xnor0} + d_{L0}$

$T_0 = d_{L0} + d_{L1} + d_{xnor0}$

Note that the expressions are identical only if the delays of latches $L0$ and $L1$ are the same. Therefore, if d_{L1} is greater than d_{L0}, then the sum of the forward and reverse delays will be an underestimate of the actual cycle time for $S0$.

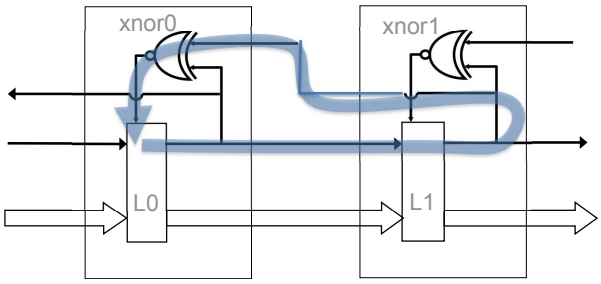


Figure 3.6: Cycles cross stage boundaries.

Synchronization Point Method for Determining Performance Metrics

Despite this subtle difficulty in assigning performance metrics to individual stages, it is still possible to assign local cycle times to individual stages. In particular, if the properties of neighboring stages are known at the time of analysis, the difference in gate delays can be taken into account when assigning performance metrics to every stage. If the connections are not known (*e.g.* different parts of the system are designed and analyzed independently

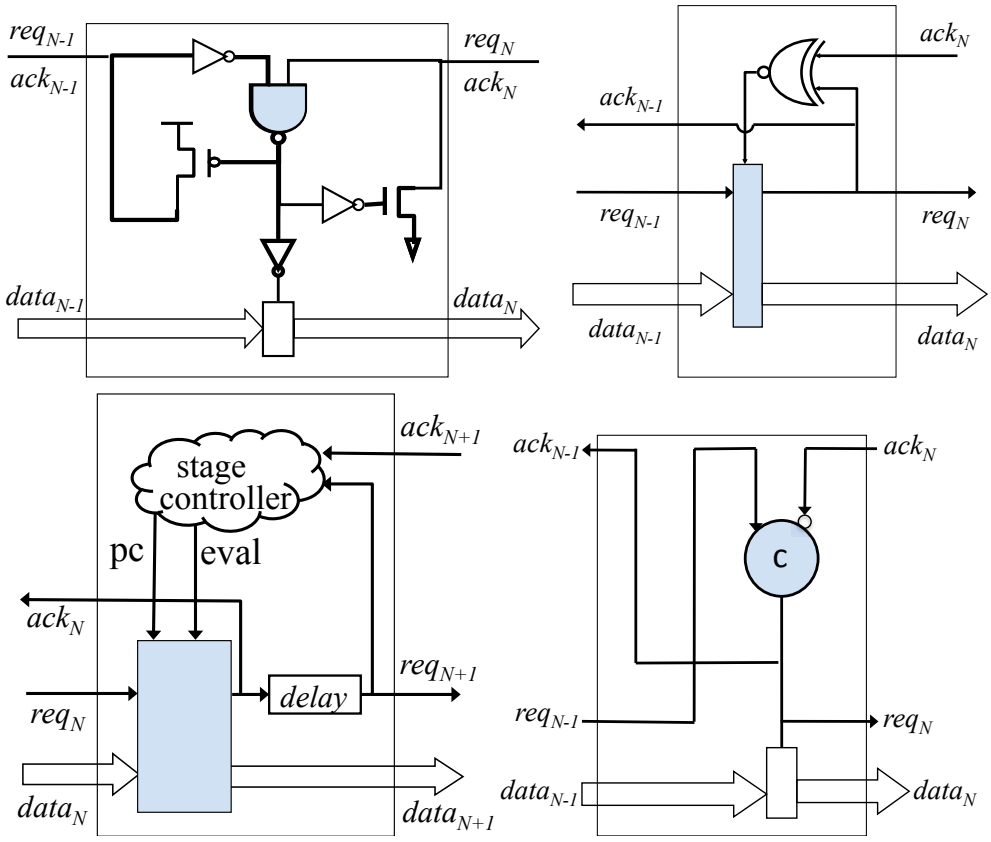


Figure 3.7: Synchronization points in various pipeline styles; a) Synchronization point for GasP is a *nand* gate b) Synchronization point for MOUSETRAP **is a latch; c) Synchronization point for high capacity is the domino logic; d) Synchronization point for Sutherland's Micropipelines is the C element**

before being composed together as modules), some additional information is retained for the stages on the edge of the module. This section introduces the *synchronization point* method for assigning performance metrics, which can be leveraged in either case.

To perform synchronization locally through handshaking, every pipeline style must have a synchronization point: a location at which some set of signals from predecessor and successor stages come together. The synchronization points of common pipeline styles that have synchronization points are highlighted in Figure 3.7. The gates that comprise the synchronization point are shaded in each example. In GasP, the incoming request from the previous stage and the incoming acknowledge from the next stage synchronize at the *nand*. Similarly, in Sutherland's Micropipelines, the *c-element* is the synchronization point. In Mousetrap, the

synchronization point is the latch, with the request coming into the latch from the predecessor stage and the acknowledge coming coming from the successor stage. Similarly, in the high capacity pipeline style, domino logic–which plays a similar role to a latch–is the synchronization point.

A more general model of a stage with a synchronization point is shown in Figure 3.8. Note that the paths for both the forward and reverse latency pass through the synchronization point. In addition, the model of Figure 3.8 makes apparent two previously undefined paths that pass through the synchronization point of each stage: the response path and the revival path. The response path is the path from an incoming request through the synchronization point to the outgoing acknowledge. The revival path is the path from the incoming acknowledge through the synchronization point to the outgoing req.

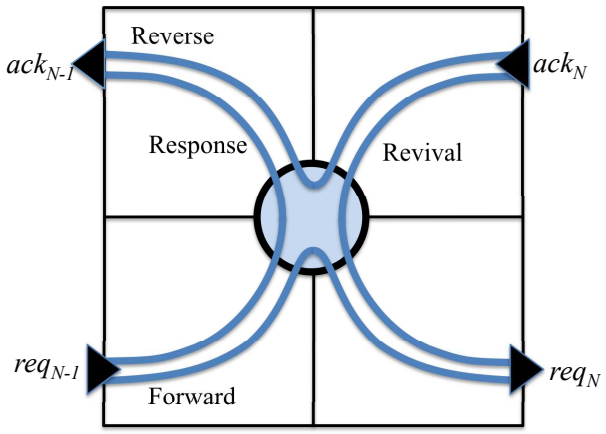


Figure 3.8: Abstract model for pipeline stage N with shaded synchronization point

Forward Latency The definition of forward latency used in this thesis is the delay between a request signal entering the synchronization point of some stage, S_n, and a subsequent request signal entering the synchronization point of the next stage, S_{n+1}. This is similar to the definition of forward latency given in Section 3.1.1 except that the path that is used to calculate the latency crosses what are generally considered to be the boundaries of a stage. In

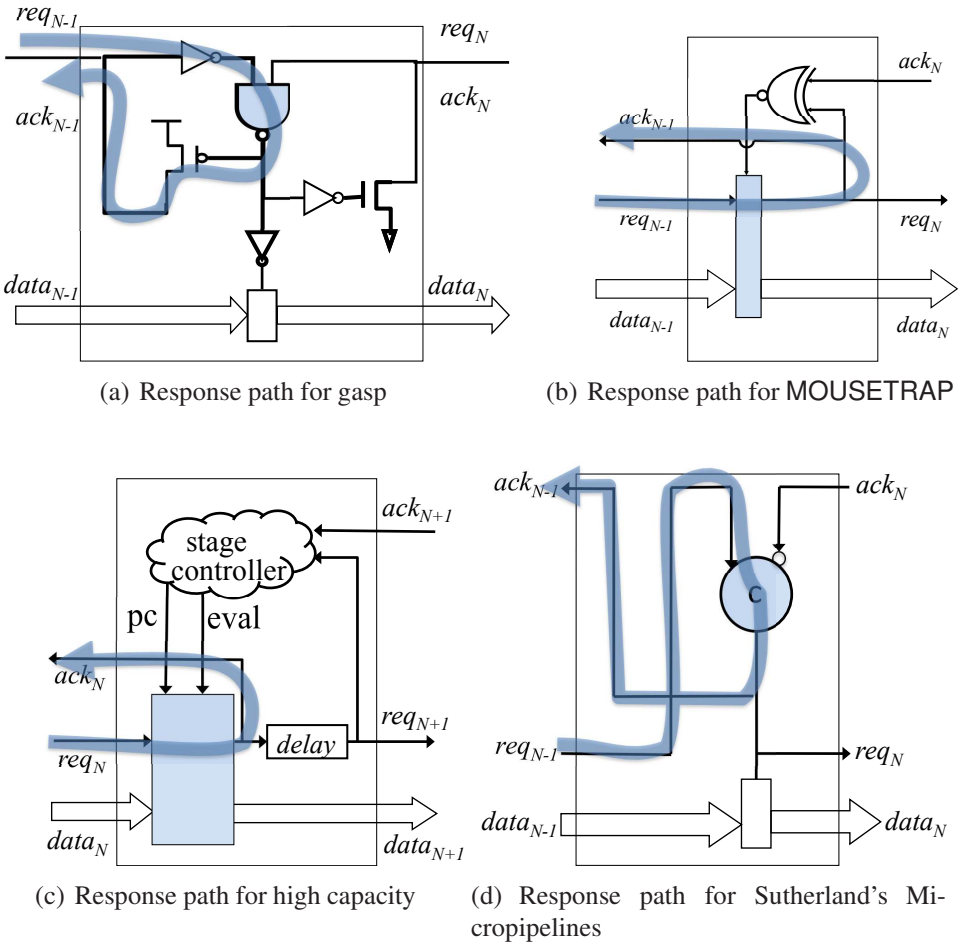
particular, the forward latency path excludes forward delays before the synchronization point of S_n while including those before the synchronization point in S_{n+1}.

Reverse Latency The definition of reverse latency used in this thesis is the delay between an acknowledge signal entering the synchronization point some stage, S_n, to it entering the synchronization point of the previous stage, S_{n-1}. This is similar to the definition of reverse latency given in Section 3.1.1 except that it crosses what are generally considered to be the boundaries of a stage. In particular, it excludes reverse delays before the synchronization point of S_n while including those before the synchronization point in S_{n-1}.

Response Latency Response latency is the time from when a stage receives a request to the time that it sends an acknowledge. Figure 3.9 traces the response paths for several different pipeline styles. The response latency of a stage can be estimated counting the gate delays that the path passes through. Alternatively, it can be measured for an individual stage through simulation by sending a request into an empty stage and recording the delay before an acknowledge is received.

Revival Latency Revival latency is the latency between receiving an acknowledge and sending a request. Figure 3.10 traces the response paths for several different pipeline styles. The revival latency of a stage can be estimated counting the gate delays that the path passes through. Alternatively, it can be measured for an individual stage through simulation by sending an acknowledge into a stage that has a data item waiting (*i.e.* in a full pipeline) and then recording the time until a new request is generated.

Stage Model with Response and Revival Time To characterize the performance of a single stage, then, the standard forward and reverse delay metrics are augmented by two additional metrics: response latency and revival latency. Like the forward and reverse latency of a stage, the response latency and the revival latency can be observed through the external signals of



(a) Response path for gasp

(b) Response path for MOUSETRAP

(c) Response path for high capacity

(d) Response path for Sutherland's Micropipelines

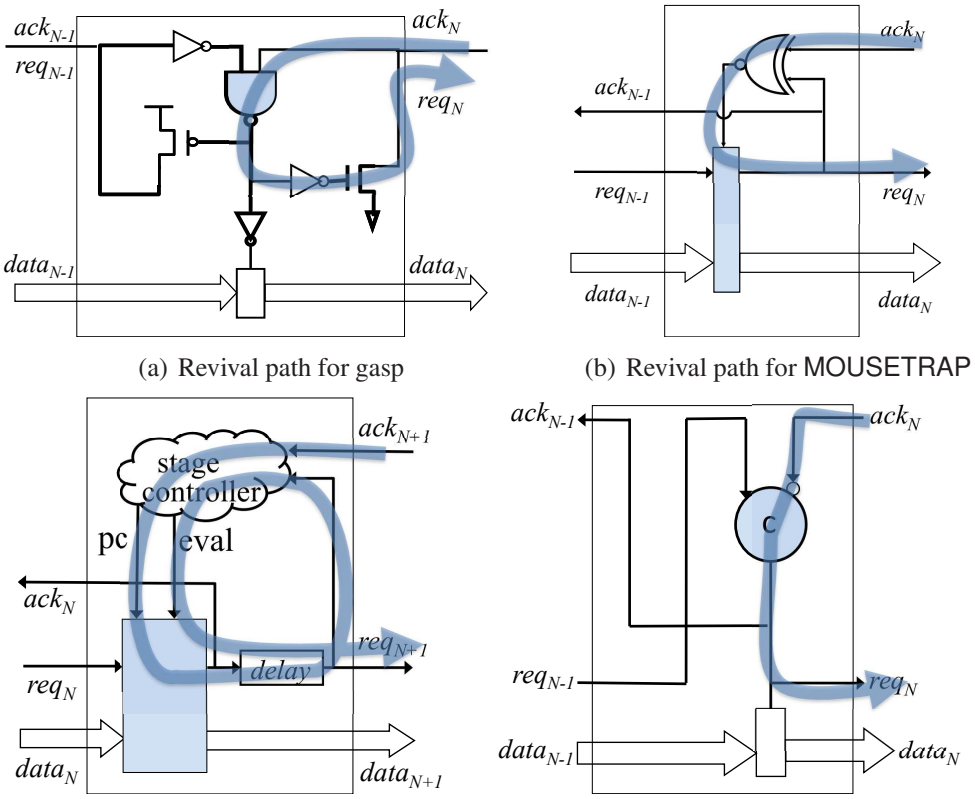
Figure 3.9: Response paths for various pipeline styles

the stage. That is, finding values for these metrics does not require knowledge of the particular internal implementation of a stage.

Response and revival delay are not used directly during system-level analysis, but instead will aide in finding the value of the local cycle time when stages are composed together. In particular, a stage S_n is designated with a cycle time that is its revival time plus the response time of its successor stage, S_{n+1}.

$$T_n = revival_n + response_{n+1} \tag{3.1}$$

Figure 3.11 illustrates how interstage cycle times form when several pipeline stages are



(a) Revival path for gasp

(b) Revival path for MOUSETRAP

(c) Revival path for high capacity

(d) Revival path for Sutherland's Micropipelines

Figure 3.10: Revival paths for various pipeline styles

joined together in a pipeline. The revival path of stage $S0$ combines with the response path of stage $S1$ to create the cycle time T_{S0}.

An even further abstracted version is shown in Figure 3.12; this notation will be used when describing performance for an individual stage in later chapters. In the figure, the numbers denote the latencies before and after the synchronization points. For example, stage $S0$ has a forward delay of zero before the synchronization point and a forward delay of one after the synchronization point, for a total forward delay of one. The cycle time of S0 is the revival time of S0 summed with the response time of S1. In the example of Figure 3.12 the total cycle time for S0 is four.

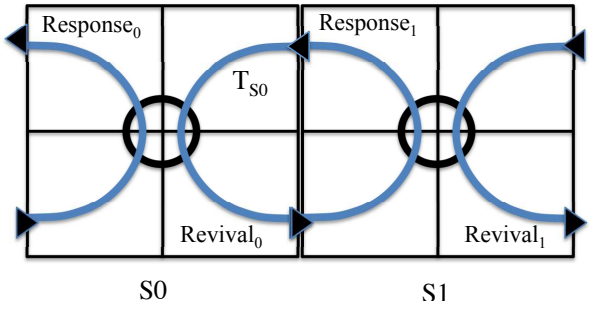


Figure 3.11: A cycle time composed on one revival and one response time.

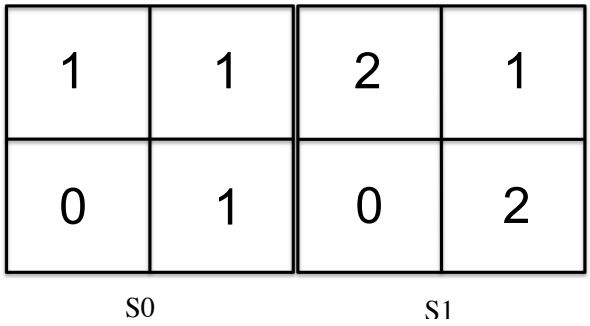


Figure 3.12: Indicating revival and response times.

Models for Protocols with Complex Cycles Thus far, the modeled stages contain cycles that cross only one stage boundary between two adjacent stages. Some pipeline styles, such as PS0[67], have more complex cycles that can cross several stage boundaries. Most half-capacity pipeline styles have some cycle that crosses at least two stage boundaries. In these cases, the same notation can still be used, but several sets of response, revival, forward, and reverse delays of stages must be added.

Figure 3.13 illustrates composing stages that have a cycle that crosses more than one stage boundary. The cycle includes delays in stages S0 and S1 as well as S2. In many pipeline styles, an additional cycle that passes through the same two synchronization points twice: once to indicate that a data item is present and a second time to indicate that the data item is no longer present. If four-phase handshaking is used, the first time through the cycle, the request values are rising and the second time through they are falling. Figure 3.14 illustrates such a cycle. The cycle time, T_0 is the *maximum* of the two cycles shown in Figures 3.13 and

3.14.

$T_0 = max(revival_0 + revival_1 + response_1 + response_2, revival_0 + response_1, revival_0 + response_1 + revival_0 + response_1)$

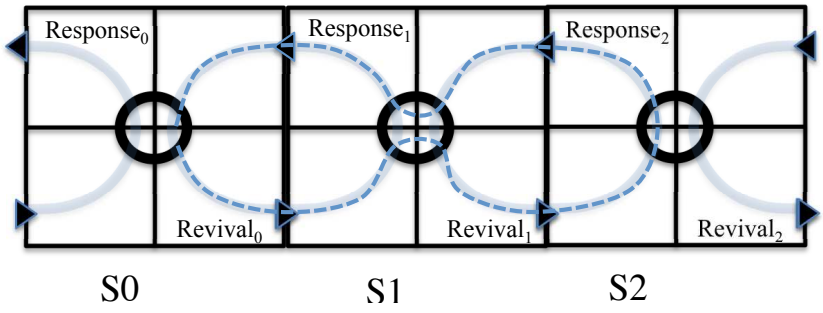


Figure 3.13: A cycle time that crosses two stage boundaries.

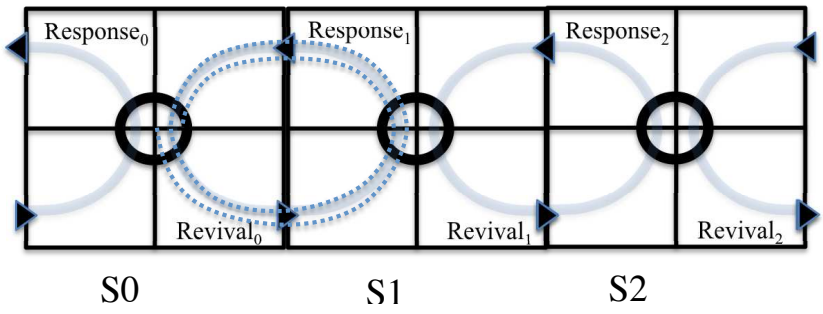


Figure 3.14: A cycle that operates simultaneously with the one of Figure 3.13

Models for Asymmetric Stage Implementations Gates can sometimes have different rising and falling delays, which further complicates finding performance metrics. To model this occurrence using the synchronization point method, two copies of each set of delays needs to be maintained. Figure 3.15 shows one setup for finding the performance metrics of stages that have asymmetric delays and that have cycle times that cross more than one stage boundary. Several different cycles exist within this system, and are indicated in the figure. The cycle time T_0 is the *maximum* length cycle of all the cycles shown.

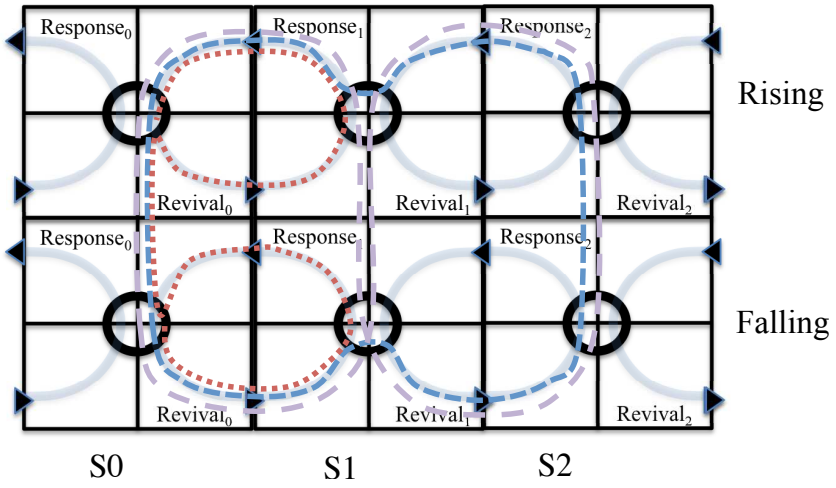


Figure 3.15: Stages with asymmetric rising and falling delays require further information for analysis

3.2.4 System-level Use of Performance Metrics

Once each stage is characterized using synchronization point notation, stages that are composed together can be assigned the four performance metric values that are needed to perform system-level analysis: forward latency, reverse latency, cycle time, and capacity.

If the properties of neighboring stages are known at the time of analysis, every stage can have the four metrics assigned. At times, however, parts of the circuit are designed modularly and to be composed later with other circuit components. If this is the case, some assignment of values can take place, though the response and revival times of some stages must be maintained.

Figure 3.16 illustrates the entire process of assigning performance metrics to stages, beginning with a pipelined circuit that uses the Mousetrap pipelining style. Either simulation data or estimates using gate counts in the pipelined circuit of Figure 3.16(a) provide delay numbers for the abstract representation of Figure 3.16(b). Finally, the forward latency, reverse latency, and cycle time are calculated using the formulas presented in Section 3.2.3 to yield the stages of Figure 3.16(c). In this example, stages S0 and S3 retain their respective response and revival times so any further stages that are attached later can use this information to calculate

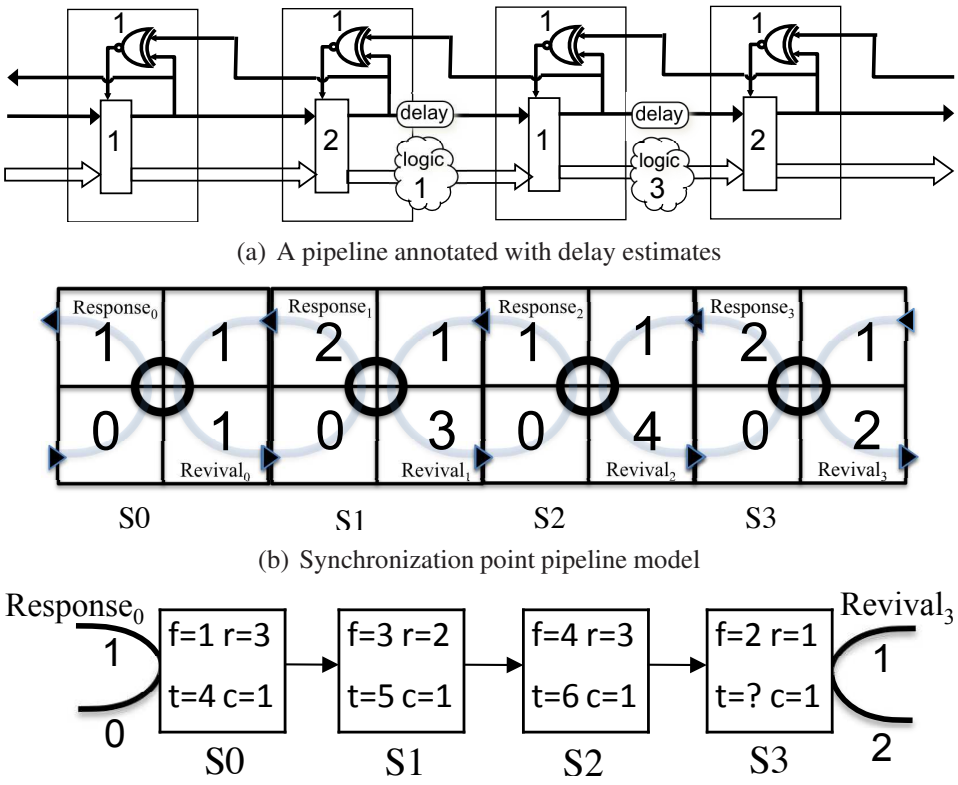


Figure 3.16: The process of assigning performance metrics to stages

performance metrics.

3.3 A Hierarchical Model for Asynchronous Systems

This thesis focuses on the set of pipelined circuits that are hierarchical. This section formally defines hierarchical composition and introduces the kinds of hierarchical constructions that the analysis method currently supports: sequential, parallel, conditional, and loop composition.

3.3.1 Definition of Hierarchical

For the purposes of this thesis, a given microarchitectural construct is hierarchically composable if it has exactly one input port and one output port. Moreover, connections into or out of a component other than at its unique entry and exit points will break its hierarchical nature. A

component indicates a passive input port with an open circle, ○, and an active output port with closed a circle, ●. These ports are connected to ports of other stages or blocks in the next level of the hierarchy.

To be usable in the analysis method of Chapter 4, a hierarchically composable component needs to have a defined set of inputs and outputs that can be obtained through dataflow analysis. This is analogous to performing *live variable data-flow analysis,* in order to find IN and OUT sets of the components code [25]. In particular, the IN set is the set of data values that are required as input to a component either because they are used within the component or because they must be relayed to a successor. The OUT set of a component is the union of the IN sets of all of its successors. These sets are used to determine the context of a component—the set of values that need to be communicated between stages. In general, calculating the IN and OUT sets for a system with arbitrary connections is a difficult problem. Each of the components, however, has a formula for computing the IN and OUT sets when composed hierarchically.

3.3.2 Supported Hierarchical Constructs

This section lists the hierarchical constructs that will be used throughout this dissertation: a single stage, sequential constructs, parallel constructs, conditionals, and iteration. For each construct, this section describes its behavior and illustrates its structure; this is sufficient information to understand the use of these constructs in the remainder of this dissertation. For the reference of readers wishing to implement dataflow analysis using these components, an equation for the IN set of each component is also listed. Given a system that has a known hierarchical structure, using these equations obviates the need to run time-consuming dataflow algorithms while ensuring that each stage receives all the necessary data and no unnecessary data.

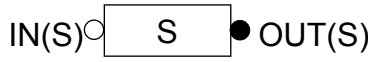


Figure 3.17: A hierarchically composable single pipeline stage

A Single Stage

The single pipeline stage is the building block of all the hierarchical components. Figure 3.17 shows a graphical representation of a single stage. The stage has one passive port for input and one active port for output.

The IN set of the stage, IN(S), is the set of values used in stage S in addition to the set of values used by components connected at its output port, less the set of values that are assigned new values by that stage (*i.e.* the killed values.) The out set of S can be computed only when the next stage in the hierarchy is known, and is equivalent to the IN set of the component connected at the output port. For reference, the equation for calculating the IN set of a single stage is shown below:

$$IN(S) = use(S) \cup out(S) \cap \neg killed(S)$$

Sequential

The sequential component takes two other hierarchically composable components and attaches the output port of one to the input port of the other in sequence. Figure 3.18 shows a graphical representation of the hierarchically composable sequential component. The two blocks, $c1$ and $c2$, in the figure each represent any possible choice of hierarchical component.

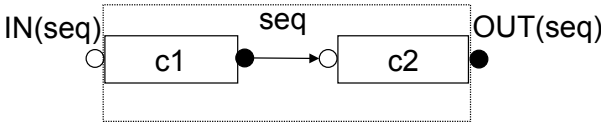


Figure 3.18: A hierarchically composable sequential component

The overall behavior of this component is to first route data through component $c1$, and then route the output of that computation to the input of $c2$. The IN set of the sequential

component can be computed using the IN sets of the two component parts.

$$IN(seq) = use(c1) \cup (use(c2) \cap \neg killed(c1)) \cup (out(seq) \cap \neg killed(c1) \cap \neg killed(c2))$$

Parallel

The parallel component takes two other hierarchically composable components and connects them using fork and join stages. Figure 3.19 shows a graphical representation of the parallel construct. The two outputs of the fork stage are attached to the respective input ports of the sub-components, $c1$ and $c2$. Similarly, the two inputs of the join stage are attached to the respective output ports of the sub-components.

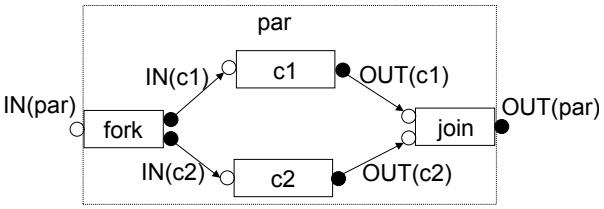


Figure 3.19: A hierarchically composable parallel component

Overall, the behavior of a parallel component is to route incoming data to both components $c1$ and $c2$ simultaneously. The fork stage sends exactly one data item to $c1$ for each data item it sends to $c2$, and the join stage waits to receive data from both $c1$ and $c2$ before it can send data along further.

The IN set of a parallel component is a combination of the use sets of the two components and the out set.

$$IN(seq) = use(c1) \cup use(c2) \cup (out(par) \cap \neg kill(c1) \cap \neg kill(c2))$$

Conditionals

A conditional (*i.e.* if/then/else) construct can be implemented hierarchically in one of two ways. A speculative conditional, as seen in Figure 3.20, is implemented as a parallel construct with three branches: the two computation branches and one branch to compute the Boolean

value. All branches operate in parallel, and both $c1$ and $c2$ produce data to send to the join stage. The join stage has logic functionality that uses the value of the computed Boolean to determine which values to send on and which to discard. In later chapters of this thesis, the speculative conditional is not treated as a special case, but instead is analyzed just as any parallel component.

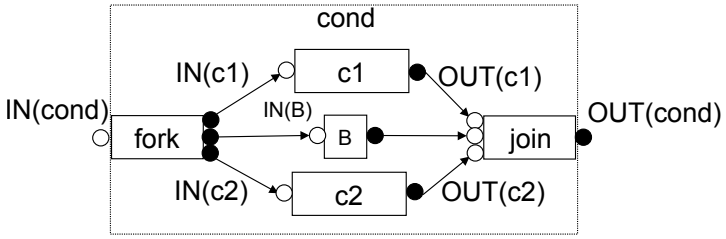


Figure 3.20: A hierarchically composable parallel component with speculative conditional

Conditionals can also be implemented non-speculatively. That is, the data is sent in either the *then* path or the *else* path, but not both simultaneously. Figure 3.21, shows an implementation of a non-speculative conditional. The Boolean value must be pre-computed and passed into the component, rather than being computed within the conditional.

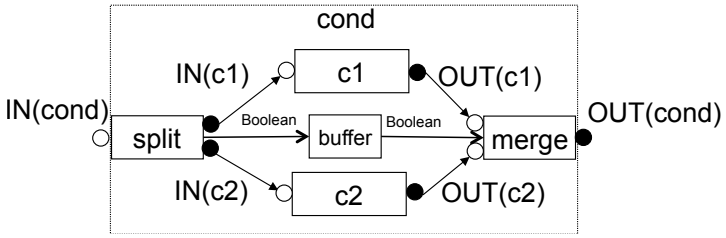


Figure 3.21: A hierarchically composable non-speculative conditional component

The behavior of the conditional is to wait for the Boolean value and the data value before sending the data to either component $c1$ or component $c2$. The Boolean value is also sent along a buffered path with no computation to the merge stage, which uses it to select a data value to send out.

The IN set for a non-speculative conditional can be computed from the in sets of the two sub-components, $c1$ and $c2$, and also includes the pre-computed Boolean value.

$$IN(cond) = use(c1) \cup use(c2) \cup (out(par) \cap \neg(kill(c1) \cap kill(c2))) \cup Bool$$

Iteration

The loop component, which is discussed in more detail in Section 5.3.3 implements an algorithmic loop. Figure 3.22 gives a graphical representation of a pipelined loop. The loop connects a hierarchical component $c1$ to a loop interface that handles the entry and exit of the data. The IN set for the loop is the same as the IN set for component $c1$.

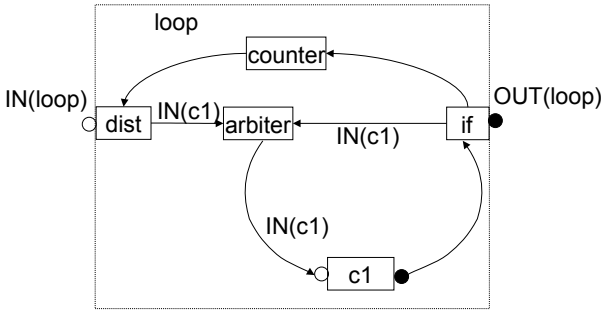


Figure 3.22: A hierarchically composable data-dependent loop component

3.3.3 Example Usage

Algorithm for Converting High-Level Code to Hierarchical Blocks

Figure 3.23 gives an algorithm for converting high-level code that is written using hierarchical constructs into a pipelined representation that uses the hierarchical components described in the previous section. This is a syntax-directed method that takes the parallel, sequential, conditional, and loop constructs in the high-level code and converts them directly to the corresponding pipelined component. This algorithm is presented as one possible way of generating pipelined representations that use hierarchical components; generating the hierarchal component representation from a high level specification is not the main focus of this thesis.

```
Pipe (P : program).
 begin
  if P is a single assignment statement S then
      output single_stage(S)
  else if P is the sequential block ‘‘P1; P2’’ then
     compose_sequential( Pipe(P1),  Pipe(P2) )
  else if P is the parallel block ‘‘P1 || P2’’ then
     compose_parallel( Pipe(P1), Pipe(P2) )
  else if P is the conditional ‘‘if(B) then P1 else P2’’ then
     compose_conditional( Pipe(P1), Pipe(P2) )
  else if P is a loop ‘‘for (n) P1’’ or ‘‘while (cond) do P1’’ then
     compose_loop( Pipe(P1), Pipe(cond) )
 end
```

Figure 3.23: Our transformation algorithm

High-level Code Example

Figure 3.24 shows example high-level pseudocode that can be converted into a hierarchical representation. It includes sequential, parallel, and loop constructs.

```
func compute(in_context)
 s1; s2;
  for i = 1 to N
   ( s3; s4 ) || ( s5; s6 )
  end
 s7; s8;
 return(out_context)
```

Figure 3.24: Sample code written in a hierarchical high-level language

Parse Tree Representation

One possible parse tree for the code of Figure 3.24 is shown in Figure 3.25. This is the parse tree generated by the algorithm of 3.23 given a left-to-right parse order. The parse tree serves as an intermediate representation between the high-level code and a pipelined representation.

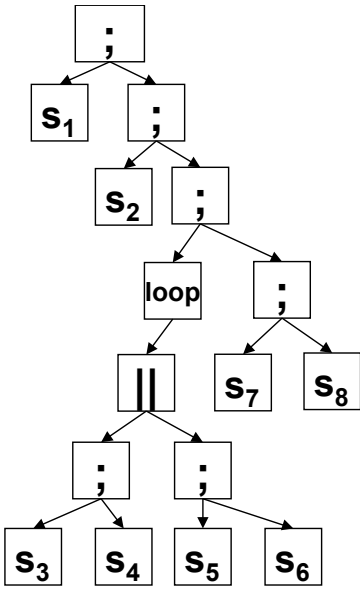


Figure 3.25: Parse tree for code in figure 3.24

Pipelined Hierarchical Component Representation

Finally, 3.26 shows the pipelined system that implements the high-level code of Figure 3.24. The algorithm of Figure 3.23 generates the pipelined system directly from the parse tree. Stepping through the algorithm, it first encounters a sequential operator, which generates the pipeline stage s_1 in sequence with the rest of the system. The algorithm then encounters two more sequential operators and puts stage s_2 and the loop in sequence. The algorithm then recurses into the body of the loop, encountering and placing parallel and sequential operators in turn. Finally, it reaches the last sequential operator, which leads to stages s_7 and s_8 being placed in sequence at the end.

3.4 Modeling System Performance : Canopy Graphs

This thesis uses the canopy graph, a plot of throughput vs occupancy, to represent the performance of a system. Though using these plots to describe system performance is not new [67, 57, 31], a key insight of this thesis is that retaining the full canopy graph information for

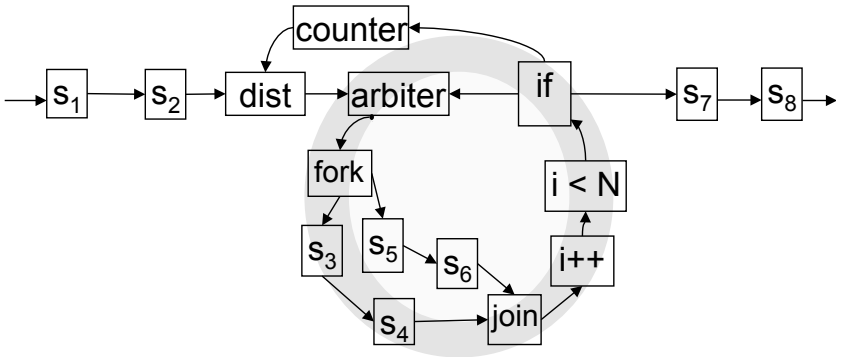


Figure 3.26: The result of hierarchical composition for the code of Figure 3.24

sub-circuits enables hierarchical composition of circuit performance. Detailed background information on the canopy graph can be found in Section2.2.2.

This section first formally defines the canopy graph; this definition will be used throughout the remainder of the thesis. It also describes the properties of canopy graphs and the assumptions and limitations associated with using them. It then goes on to give an example of the usefulness of calculating and storing an entire canopy graph as compared to calculating a single value for throughput.

3.4.1 Definition and Notation for Canopy Graphs

A canopy graph is a set of throughput-occupancy pairs that represents the operating region of a pipeline. By definition, the set of points within a canopy graph always form a convex polygon when plotted as throughput vs occupancy. Moreover, every canopy graph can also be defined in terms of a set of linear inequalities, which together yield a convex region.

Notation for Canopy Graphs The notation $\mathbb{C}_P$ indicates that the operating region of pipeline P is the set of points within $\mathbb{C}_P$. For sections of this thesis that discuss several pipelines–such as P_0, P_1, P_2–the shorthand notation $\mathbb{C}_0$ $\mathbb{C}_1$ $\mathbb{C}_2$ may also be used to refer to the canopy graphs of the respective pipelines.

Definition of Throughput The throughput of a circuit is the frequency with which the pipeline takes in data and produces data. A pipeline that operates within as environment that enters input and takes output at a constant rate will eventually settle into a steady rhythm of producing and consuming data; this known as *steady state* operation. Unless otherwise specified, the term *throughput* refers to the steady state throughput of a pipelined system, rather than a brief spurt of high throughput or a temporary lull of low throughput.

Definition of Occupancy In general, the occupancy of a pipeline is the total number of distinct data items that the pipeline contains. Note that if one incoming data item is forked onto two parallel paths, the total occupancy of the pipeline does *not* increase, because a copy of the same incoming data does not constitute a distinct data item. On the other hand, if a non-speculative conditional has one data item in the *then* branch and another in the *else* branch, these two data items are distinct and each is counted towards to the total occupancy of the pipeline.

Within this thesis, the term *occupancy* refers more specifically to the steady state occupancy of a pipeline. Although a snapshot of a pipeline at any discreet moment in time will show it with an integer occupancy, the steady state occupancy is not necessarily an integer. Consider a simple example of the linear pipeline shown in Figure 3.27, which shows four snapshots of a pipeline under steady state operation. A circle within a stage indicates the presence of a data item. At time 0, the pipeline contains three data items that are progressing forward through the pipeline. One time unit later, one of the data items has left the pipeline, but the environment has not yet provided a new one. At time 2, a new data item has entered and one time unit later another has left. In the steady state, then, this pipeline is currently operating at an occupancy of 2.5.

All throughput-occupancy points in the canopy graph for a pipelined circuit are valid operating points, not just the ones with integral occupancies, because steady state occupancy can be fractional.

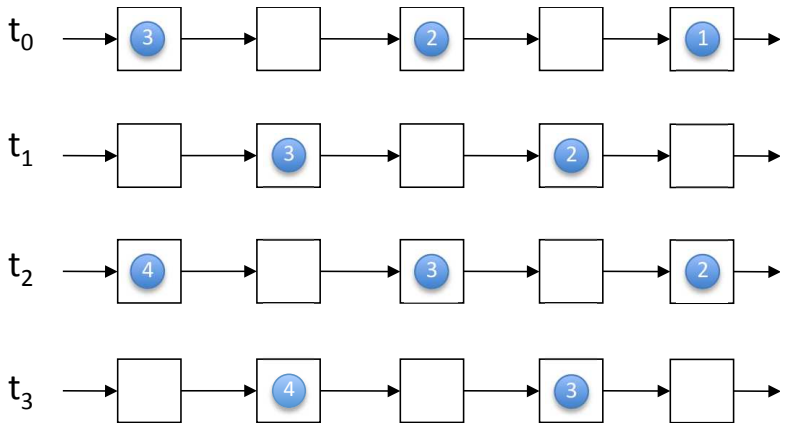


Figure 3.27: Snapshots of the operation of a pipeline with steady state behavior.

Definition of Operating Region The operating region of a circuit is the set of throughput-occupancy pairs that it can potentially be induced to work at, under some set of environment behaviors. For example, if the external environment provides inputs and receives outputs quickly (*i.e.* as soon as the pipeline is ready) the throughput of a pipeline system will settle into a steady rhythm of producing data; this is the maximum throughput of the pipeline. The maximum value of the throughput, however, does not fully define the operating region.

If, instead, the environment maintains a fixed number of data items within the pipeline, the system will settle into a different throughput. On the other hand, if the environment maintains a fixed throughput by limiting incoming data, the occupancy will naturally shift. Taken as a whole, these interactions between throughput and occupancy define the operating region of the pipeline. The relationship between throughput and occupancy will be further explained in Section 3.4.2.

3.4.2 Properties of Canopy Graphs

Every canopy graph shares three basic properties. All indicate that the throughput of a pipeline depends on its occupancy. All are bounded by a set of line segments that can be defined in terms of the type of interaction within the pipeline that limits the throughput, and all are convex polygons.

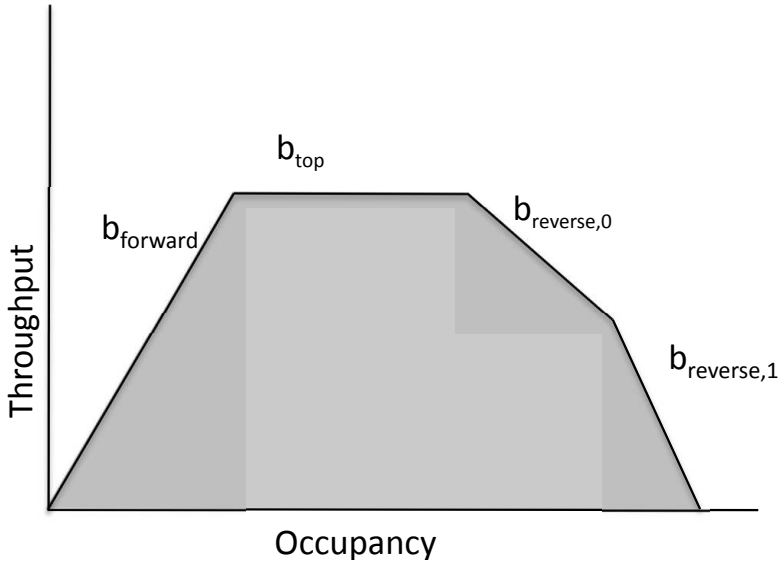


Figure 3.28: Canopy graph C with four limiting segments

Effects of Occupancy on Throughput

The throughput of a pipeline is affected by the total number of data items and also by the total number of stages without a data item, or "holes". The effect of occupancy on throughput is described in Section 2.2.2, which gave background on the use of canopy graphs. This section aims to generalize and give intuition about the relationship of occupancy and throughput in pipelined systems. For reference, Figure 3.28 shows a generic canopy graph.

The throughput and occupancy of a pipeline are highly dependent on each other. In particular, the throughput may be limited by the lack of data within the system; this occurs when the environment does not provide data as quickly as the pipeline can use it. Consider the case of only one data item flowing through the system. The frequency with which the pipeline can produce data when it is limited to contain only one data item is one over the total latency of the pipeline. Adding a second data item will double the total number of data items and therefore tend to double the throughput. This linear relationship between the data items and throughput holds within the data limited region. In Figure 3.28, this relationship gives the shape of the line segment labeled $b_{forward}$.

The throughput can also be limited by a lack of holes, if the environment is providing data

quickly yet not removing completed data items as fast as the system produces them. Holes, in some ways, are the dual of data within a system. There are some key differences, however, that lead the hole limited region to differ from the data limited region. For example, if a circuit contains a parallel construct, the two branches of the construct will always contain the same number of data items. The number of holes, however, is determined by the number of stages minus the number of data items. Two parallel branches, therefore, can have a *different* number of holes, even though they must always have the *same* number of data items. As a result, the hole-limited region of the canopy graph can be more complex than the data-limited region as different portions of the circuit begin to dominate at different occupancies.

Finally, the worst cycle time in the within the system system can also limit the throughput; this the speed limit set by the pipeline itself that cannot be exceeded regardless of the environment behavior. This occurs when neither data nor holes are lacking, but some stage or set of stages create a cycle that limits throughput. This limit to the throughput is analogous to previous methods that characterize circuit performance with single performance number [29, 36, 71, 65]. When applied to a canopy graph, however, this shows not only a single throughput number but also a range of occupancies that will allow the pipeline to reach that maximum throughput. This occupancy or range of occupancies is known as the *ideal occupancy* of the pipeline.

Bounded by a Set of Line Segments

Every canopy graph is bounded by a set of boundary line segments, B, which can be defined based on the type of interaction within the pipeline that is limiting the throughput. The set of boundary segments forms a function of throughput vs. occupancy, and the canopy graph is the hypograph of this function, excluding points that have negative values. $\mathbb{C}B = \{(k, tpt) : k \in \mathbb{R}^n \wedge k \geq 0,\ tpt \in \mathbb{R} \wedge tpt \geq 0,\ tpt \leq B(k)\} \subseteq \mathbb{R}^n$ [41]

Figure 3.28 shows an example canopy graph. The boundaries of this canopy graph consist of the following types of limiting segments:

1. *forward segments* bound the operating region in the data-limited region of the canopy graph and are determined by the total forward latency of the system. Because the throughput increases as a new data item enters, the slope of the forward line segments is always positive. The line segment labeled $b_{forward}$ in Figure 3.28 is a typical forward boundary segment.

2. *top segments* bound the operating region in the cycle limited region of the canopy graph and are determined by the longest effective cycle time. Because the limiting cycle in the pipeline does not depend on occupancy, the top segment always has a slope of zero. The line segment labeled b_{top} in Figure 3.28 is a typical top boundary segment.

3. *reverse segments* bound the hole-limited region and are determined by the maximum occupancy and reverse delay of the system. Because adding an additional hole (*i.e.* reducing the occupancy) will increase the throughput when the system is hole limited, the slopes of the reverse segments are always negative. The line segments labeled $b_{reverse,0}$ and $b_{reverse,1}$ in Figure 3.28 represent the kind of line segments that can define the operating region in the hole-limited region.

In a system that is initialized as empty, the forward segment of a canopy graph will cross through the point at the origin. As a result, such systems will always have exactly one forward segment and one or more reverse segments. In some cases, the top segment may be absent; this is a degenerate case in which the top segment is a single point.

Convex Set of Points

This section states without proof that all canopy graphs for systems that are composed of the hierarchical components that Section 3.3 describes will be convex. The conventional definition of *convex* is used here: for all points x and y that are in the set $\mathbb{C}$, all points on a line segment connecting x and y are also within $\mathbb{C}$. [41]

Chapter 4 will later give a constructive proof of convexity by showing that each type of

canopy graph composition maintains this property. Note that this thesis does not claim to prove that all pipeline operating regions can be defined by a canopy graph. In particular, no general proof of operating region convexity is provided.

Intuitively, since the throughput first increases with increased occupancy, then levels off with the maximum throughput, and then decreases with a decreased number of holes, this will always define a canopy-like shape that is monotonically increasing and then monotonically decreasing. Since both throughput and occupancy are also limited to be positive numbers, this bounds the entire canopy graph to a convex shape.

3.4.3 Advantages of Canopy Graph Analysis

It may not be apparent that storing information about the occupancy and throughput together will yield any advantage over simply storing the maximum throughput. From the viewpoint of someone using a pipelined circuit as a black box, knowing the range of occupancies that a pipeline could have as it operates at its maximum throughput seems quite irrelevant to understanding the overall performance. Input and output interfaces rarely seek to intentionally control pipeline occupancy since operating the environment as quickly as possible will achieve the best performance.

When analyzing entire systems of hierarchically composed components, however, storing the canopy graph yields an advantage. In particular, the canopy graph stores enough information to enable composing the performance information of sub-circuits to find the performance of an entire system. The work of Lines [31] on pipelined parallel structures reveals an example of this. Repeating the example from Section 2.2.2, note the throughput of pipeline A alone and the throughput of pipeline B alone. Keeping these two one-dimensional values—whether they are stored as throughput, time separation of events, or global maximum cycle time—does not yield enough information to find the combined throughput. In this example, the reverse segment of $\mathbb{C}_A$ together with the forward segment of $\mathbb{C}_B$ are both needed to compute the throughput of the combined circuit.

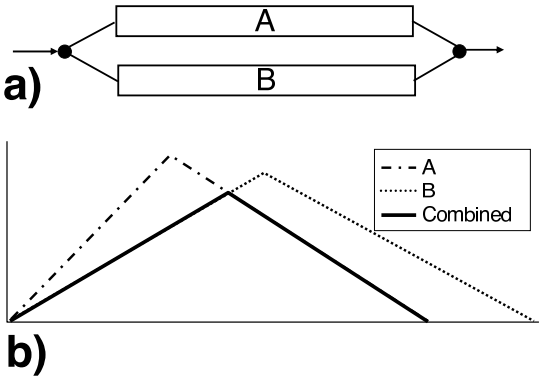


Figure 3.29: Parallel composition: a) structure, b) canopy graphs

3.4.4 Canopy Graph for a Single Stage

Chapter 4 will describe composing the canopy graphs of hierarchical components to characterize the performance of entire systems. Before this can happen, however, the base case of the canopy graph for a singe stage needs to be defined. Section 3.2 introduced performance models for individual stages and described how to derive them for different pipeline styles. These performance metrics for an individual stage can be used together to create the canopy graph for an individual stage, which in turn serves as the base case for system-level performance analysis.

Figure 3.30 shows how to use the four performance metrics–forward latency, reverse latency, cycle time, and throughput–to form the canopy graph for a single stage.

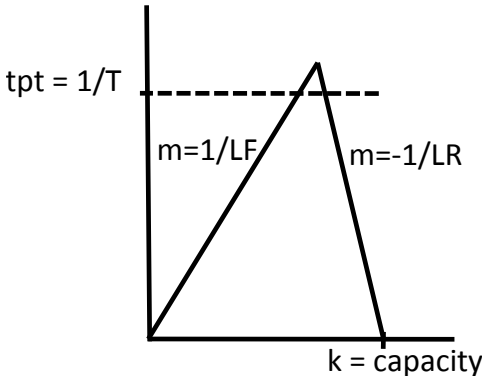


Figure 3.30: Canopy graph for a single pipeline stage

3.4.5 Assumptions

This section has introduced several assumptions about pipelines and canopy graphs. To ensure that these are not overlooked, they are listed below along with additional assumptions that hold when using canopy graphs for performance analysis.

Steady-state Assumption

Canopy graph analysis applies when a system is in "steady state" operation. A system is said to be in steady state when the frequency of items entering and exiting remains constant. A pipeline may, for a short period, have a throughput and occupancy that is outside the set of points in the canopy graph.

Empty Initialization

The pipeline is initialized to be empty; it begins without any data items present in the pipeline. The causes every forward line segment to pass through the origin, thereby ensuring that there is exactly one forward line segment. Though non-empty initialization is not explicitly covered in this thesis, the analysis can be modified to accommodate non-empty initialization, since none of the basics properties of the canopy graph described in Section 3.4.2 is violated by this change.

Performance Model

The pipeline stages use a fixed delay model, in which each stage has the same performance metrics regardless of the data passing through. This model is an accurate representation of pipeline stages that use bundled-data in the handshaking protocol and can act as a predictor or worst-case performance for stages that have data-dependent delays.

Second-Order Effects

The delay model ignores second-order effects (*e.g.*, the so-called Charlie and drafting effects [13, 69]). These can bring non-linearities to the boundaries of the canopy graph, which often has the effect of rounding-off the top of the canopy graph and preventing the ideal maximum throughput from being reached in practice.

Chapter 4

Performance Analysis

4.1 Introduction

This chapter presents an architectural-level approach for estimating the throughput of pipelined asynchronous systems. Unlike many previous performance analysis methods which operate on Petri nets[36][30][70], the proposed approach operates at a higher level of abstraction. In particular, it uses the models for pipeline stages and system performance presented in Section 3.2. Low-level events (i.e., signal transitions, handshakes, etc.) are hidden, and only higher-level information such as flow of data tokens and holes is used in the analysis. Using these abstractions enables this approach to have fast running times and also to be more intuitive to the user, while the results show that the method also gives accurate results.

This chapter presents methods for analyzing the performance of each of the hierarchical components introduced in Section 3.3: sequential, parallel, conditional, and iteration. For each of these hierarchical components, this chapter will give a method for determining performance that relies on hierarchical composition of canopy graphs. Specifically, the result of composing at each level of the hierarchy is always another canopy graph, which can in turn be used at the next level of the hierarchy for further performance analysis.

This chapter also provides a four-part proof that the canopy graph property of convexity is maintained through each hierarchical composition; this was earlier stated without proof

in Section 3.4. These proof shows that canopy graph composition for the given hierarchical components will always produce another canopy graph. This property ensures that any system that is a hierarchical composition of the four given components can be analyzed using the same analysis method without any special cases.

The analysis approach presented in this chapter extends and generalizes previous work [67] [18] [31] in pipeline performance analysis. The past work has shown that the performance of some asynchronous pipeline constructs—rings, fork/join constructs, and sequentially composed stages—can be characterized by relating throughput and occupancy using a canopy graph. This paper generalizes the canopy graph analysis to systems with any topology that is a hierarchical composition of a set of constructs found in high-level specifications: sequential and parallel composition, conditional computation ("if-then-else"), and iteration ("for" and "while" loops).

A key aspect of our work is the handling of choice and iterative loops. Choice is allowed in the system via conditional constructs (i.e., "if-then-else"). Previous approaches that handle choice typically are based on simulation or Markov chain analysis, and hence time-expensive. In contrast, our approach is analytic, not iterative or based on successive approximation, and therefore significantly faster. For loops, our approach can handle both basic single-token loops, and also the recent approach of loop pipelining [16] which allows multiple data items to be concurrently computed within the loop body.

The results indicate that an implementation of the algorithm described here gives throughput estimates that agreed with Verilog simulations of the same examples to within 4%, which indicates the validity of our analysis method. The runtime of the tool was practically negligible (less than 10 ms), even for examples with hundreds of stages.

The remainder of this chapter is organized as follows. Section 4.2 discusses previous work in performance analysis, and focuses on the most general, Petri-net-based analysis methods type of analysis methods. Section 4.3 presents our method, focusing on analysis of individual composition operators, and Section 4.4 shows how to bring them together to create a complete

hierarchical analysis method. Experimental results are interspersed throughout Sections 4.3 and 4.4. Those results are summarized, and results on additional larger examples are presented in Section 4.5.

4.2 Previous Work

My method focuses is on a special class of systems that typically result when high-level block-structured language specifications are compiled into pipelined implementations. It is intended to be integrated into a syntax-directed design flow to give fast performance estimates for system-level optimization. Although there has been much previous work on performance analysis of asynchronous systems, it does not adequately address the types of of systems that this dissertation targets.

While simulation-based methods [9][37][70] give the most accurate performance results for a given input dataset, they are too slow to be used as part of an optimization scheme in which the analysis step may need to be run repeatedly.

Other methods [30][36][71] use marked graph representations and Markov analysis to give bounds on the time separation of events. While these methods are generally faster than simulation based methods, they often suffer a state space explosion problem that makes them impractical for use on larger systems[71][30]. Work by McGee [36] avoids the state explosion problem by exploiting periodicity, thereby allowing the technique to work on larger examples. However, it is limited to relatively small systems in practice: run times on examples with as few as 12 pipeline stages are prohibitively high (*i.e.* on the order of hours) for use in a system optimization loop.

Previous analytic solutions target only a limited set of pipelined architectures. Williams and Horowitz [67] give an analytical, graph-based method for predicting the throughput of a computational ring. Greenstreet [18] analyzes rings with exponential delays, and Pang [44] extends this work to asynchronous mesh structures. My work significantly extends the class

of architectures handled by analytic methods.

Much previous work deals with models that have fixed or min-max bounded delays. Hulgaard[22] and Chakraborty [10] both give minimum and maximum bounds on time separation of events, using unfolding techniques. The unfolding methods used make the approaches unwieldy for large systems and also depend on the system being deterministic, thereby precluding choice in the system models.

McGee[35] also uses minimum and maximum delays, but avoids unfolding by exploiting the periodicity of asynchronous systems, thereby allowing it to run efficiently on larger models. It analyzes marked-graph system representations in polynomial time and gives min-max bounds of the time separation of events. Although some of these high-level constructs could be modeled using marked graphs, choice cannot be modeled. In contrast, my analysis method includes choice in the form of pipelined conditionals and data-dependent loops.

In summary, no previous work adequately addresses the challenges of analyzing large, data-flow systems with choice.

4.3 Analysis Method

This method for estimating the throughput of pipelined systems leverages the hierarchy of the original high-level code specification to allow analysis to take place in a single pass. Specifically, the algorithm first finds the throughput of the innermost nested constructs using canopy graph based analysis. Next, it repeatedly composes the resulting canopy graphs to find the throughput of the next higher level in the hierarchy.

This section presents four composition functions for use in the hierarchical algorithm that handle the following commonly used high-level constructs: sequential composition, parallel composition, conditional branches, and algorithmic loops. These correspond to the set of hierarchical components described in Section 3. Each of these functions takes as input one or more canopy graphs and computes the canopy graph of the composed pipeline. Each function

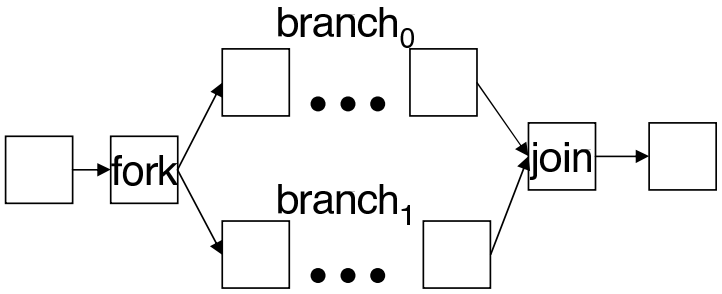


Figure 4.1: A pipelined parallel construct

maintains the convex nature of the canopy graph.

Although the use of parallel and sequential constructs was presented in past work [31], as described in Section 2.2.2, this chapter presents them along with additional formalism, examples, and comparisons to Verilog simulation. This section also aims to give some intuition about the phenomenon of slack mismatch. Slack matching—*i.e.,* balancing pipeline branches by adding buffer stages—has been shown to have a significant effect on the throughput of asynchronous pipelined systems by reducing stalling in shorter pipeline branches [4]. This chapter does not present a comprehensive approach for slack matching, but instead gives examples of slack mismatch that help demonstrate the way pipeline bottlenecks form from interactions between canopy graphs.

4.3.1 Parallel Composition

Parallel constructs are used to specify fork-join concurrency in a system. In addition, conditional (*i.e.* if/then/else) computation is often implemented using parallel constructs instead of true choice; both the 'then' and the 'else' paths are computed in parallel and the correct result is chosen by the join stage. Figure 4.1 shows a pipeline with fork-join concurrency. The ellipses indicate that some number of stages are present, without having to show each stage individually.

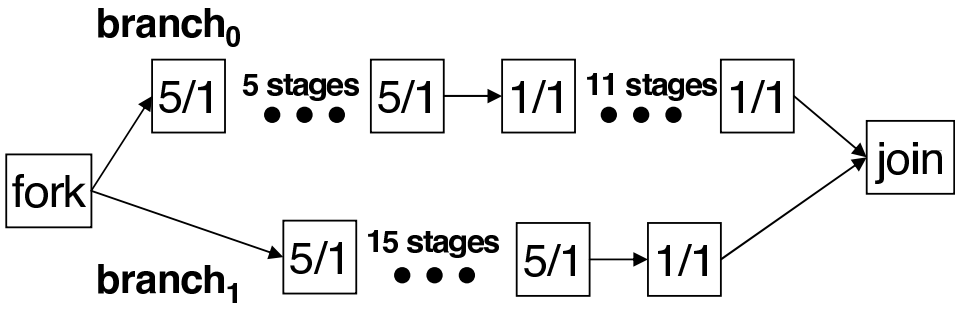


Figure 4.2: A parallel construct from CORDIC

Canopy Graph Method

Given two pipelines—each with its own canopy graph—the canopy graph of the two pipelines composed in parallel can be computed. This process involves canopy graph intersection; the new canopy graph is the area under both original canopy graphs. Intuitively, intersecting the two canopy graphs by plotting them on the same throughput vs. occupancy graph is meaningful because the two paths must have the same steady state occupancy and throughput. That is, when one item enters $branch_0$ a corresponding item always enters $branch_1$, and when an item leaves $branch_0$ a corresponding item always leaves $branch_1$. This synchronization of inputs and outputs between the two pipelines locks their throughput and occupancy together.

Looking at a specific example, the pipeline shown in Figure 4.2 represents part of a CORDIC rotation algorithm for computing trigonometric functions. The canopy graphs for its individual branches are shown in Figure 4.3. The canopy graph for the two branches composed in parallel is the area under both canopy graphs. To show that the predicted canopy graph is accurate, we also include data points from Verilog simulation of the system in Figure 4.3.

Slack Mismatch

Slack mismatch in a parallel construct can cause the overall maximum throughput in $\mathbb{C}_{par}$ to be less than that in either $\mathbb{C}_0$ or $\mathbb{C}_1$ due to stalling. Specifically, stalls occur when one of the paths becomes full, thereby preventing data from entering either path. Our method for

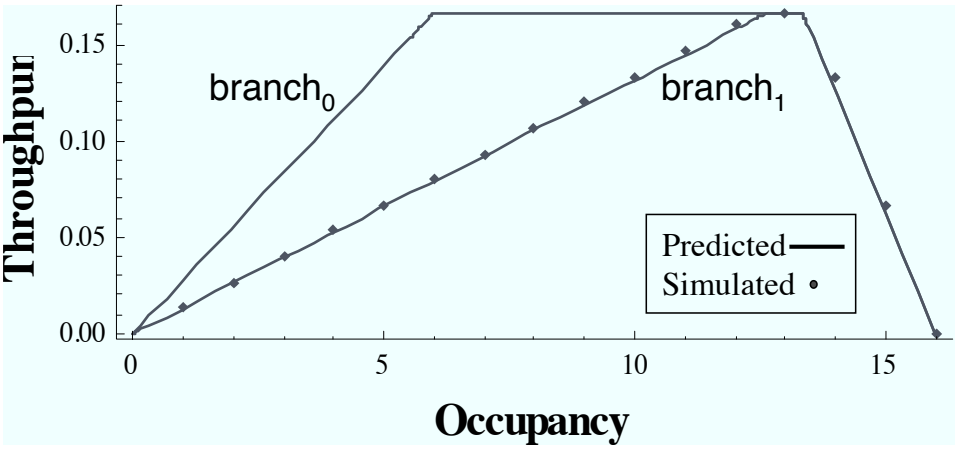


Figure 4.3: Canopy graph composition : fork/join

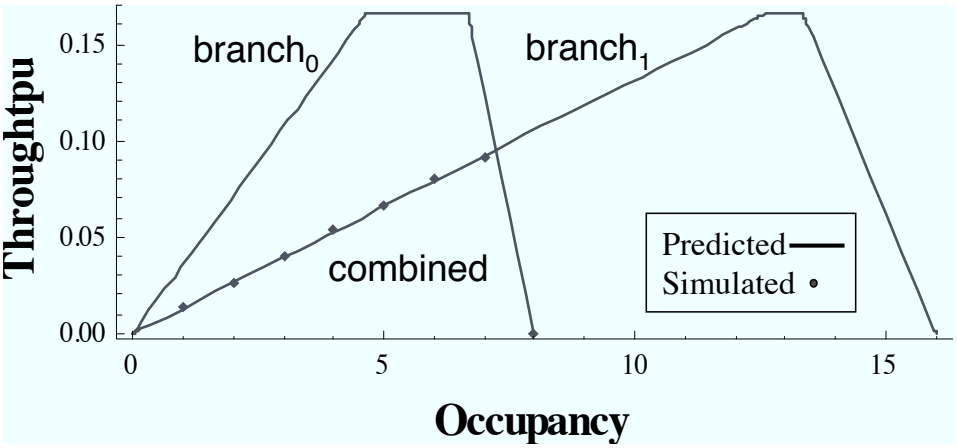


Figure 4.4: Slack mismatch example

estimating throughput implicitly handles slack mismatch. When the parallel construct is not slack matched, the peak value of the intersection region will be *below* the peaks of the canopy graphs of the two branches.

As an example, consider again the pipelined system in Figure 4.2, this time with the last eight buffer stages removed from $branch_1$, changing the total number of stages to eight. The canopy graphs for the two branches are shown in Figure 4.4. Note that the new canopy graph produced has a lower throughput than either of the original canopy graphs, indicating slack mismatch. Data from Verilog simulations, shown in the same graph, verifies that this parallel construct exhibits degraded performance due to slack mismatch.

Convexity Property

This section repeats a well-known proof [41] about the convexity of the intersection of convex polygons, this time focusing on its application to the canopy graph intersection for characterizing the performance of a parallel construct.

This proof is by contradiction and uses the definition of convexity presented in Section 3.4. Let points a and b represent any two points in $\mathbb{C}_0 \cap \mathbb{C}_1$. By the definition of intersection, both a and b belong to set $\mathbb{C}_0$ and $\mathbb{C}_1$. If $\mathbb{C}_0 \cap \mathbb{C}_1$ were not convex, there would be an a and b such that some point c on the line segment $\overline{ab}$ that does not belong to $\mathbb{C}_0$ and $\mathbb{C}_1$. This would mean that c does not belong to one of the two sets $\mathbb{C}_0$ or $\mathbb{C}_1$. For whichever set c does not belong to, this is a contradiction of that set's convexity, contrary to the assumption that both input canopy graphs $\mathbb{C}_0$ and $\mathbb{C}_1$ are themselves convex.

4.3.2 Sequential Composition

When two subsystems, p_0 and p_1, are composed sequentially, their throughput is constrained to be the same and the occupancy of the composed system is simply the sum of the individual occupancies. The steady state throughput must be the same because any data item that leaves p_0 must subsequently enter p_1, based on the definition of sequential composition. The occupancy of the system as a whole is the sum of the individual occupancies because, based our the definition of occupancy given in Section 3.4, each data item in p_0 and p_1 is distinct and counted towards the total number of data items. This relationship between throughput and occupancy of the two pipelines yields the following relationship for the combined canopy graph. In particular, a throughput-occupancy point (tpt, k) is a member of set $\mathbb{C}_{seq}$ if there exists a point $(tpt_0, k_0) \in \mathbb{C}_0$ and a point $(tpt_1, k_1) \in \mathbb{C}_1$ such that $tpt_0 = tpt_1 = tpt$ and $k_0 + k_1 = k$. For conciseness, the $+$ operator designates that two canopy graphs are composed in sequence.

In practice, the canopy graph for the composition of two pipelines is computed as follows: for each throughput value that is attainable in both systems, sum the ranges of allowed occu-

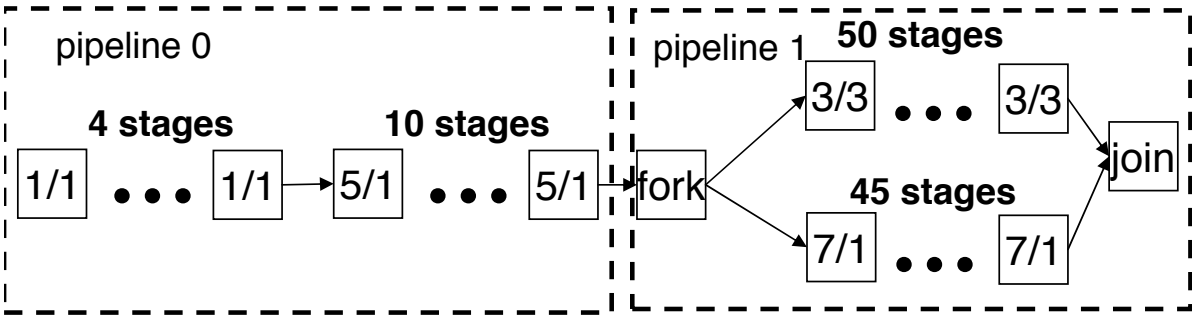


Figure 4.5: Sequential composition of a parallel construct

pancies for each subsystem at that throughput. This gives the allowed range of occupancies for the sequentially composed construct at each throughput.

Figure 4.5 shows an example pipeline with one sequential construct composed in sequence with a parallel construct. The canopy graphs for the stages in series and the parallel construct are shown together in Figure 4.6. To demonstrate the method for sequential composition, the figure indicates that points $k1$, $k2$, and $k4$ and points $k3$ and $k5$ respectively have the same throughput for both canopy graphs. The canopy graph for the sequentially composed system is shown in Figure 4.7. The figure indicates the points on the final canopy graph represent additions of occupancies in the original two canopy graphs. In addition, the throughput-occupancy points displayed in the figure were obtained through Verilog simulation; these closely follow the shape of the predicted graph.

Convexity Property

This section shows that the sequential composition of two canopy graphs maintains the canopy graph convexity property. Based on the definition of convexity, the canopy graph $\mathbb{C}_0 + \mathbb{C}_1$ is convex if for any two points, a and b, that are a members of $\mathbb{C}_0 + \mathbb{C}_1$, all points that lie the line segment between a and b are also members of $\mathbb{C}_0 + \mathbb{C}_1$.

Let the points a and b represent any two members of the set $\mathbb{C}_0 + \mathbb{C}_1$. Based on the definition of the $+$ operator for canopy graphs, there must be some point a_0 in set $\mathbb{C}_0$ and a_1 in set $\mathbb{C}_1$ for which $a_0.tpt = a_1.tpt = a.tpt$ and $a_0.k + a_1.k = a.k$. A similar statement is

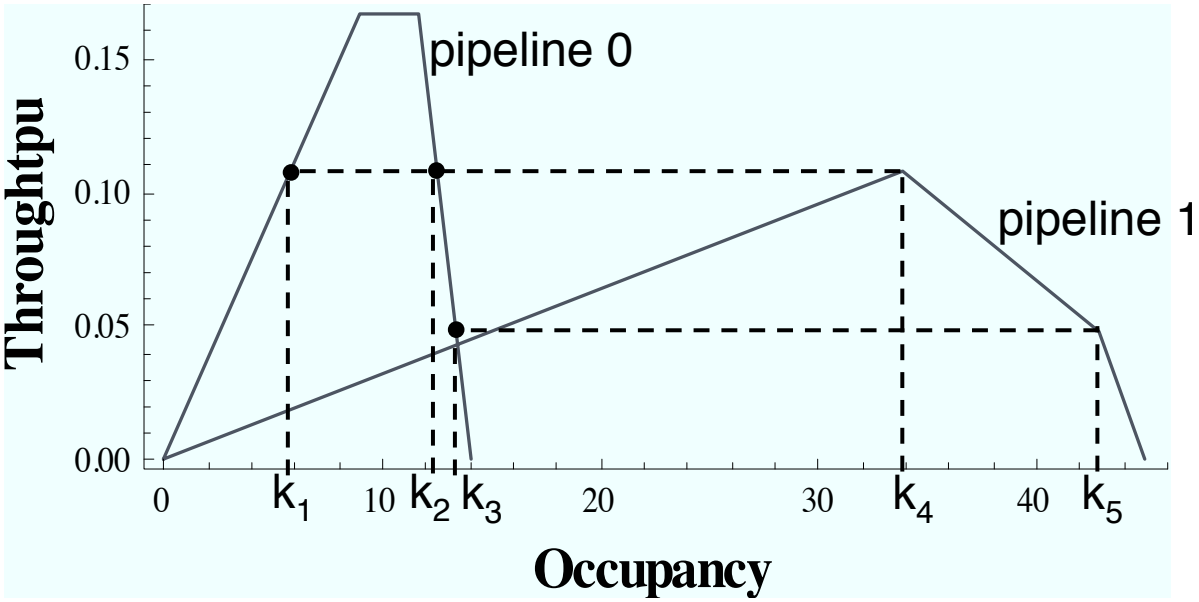


Figure 4.6: Composing in sequence

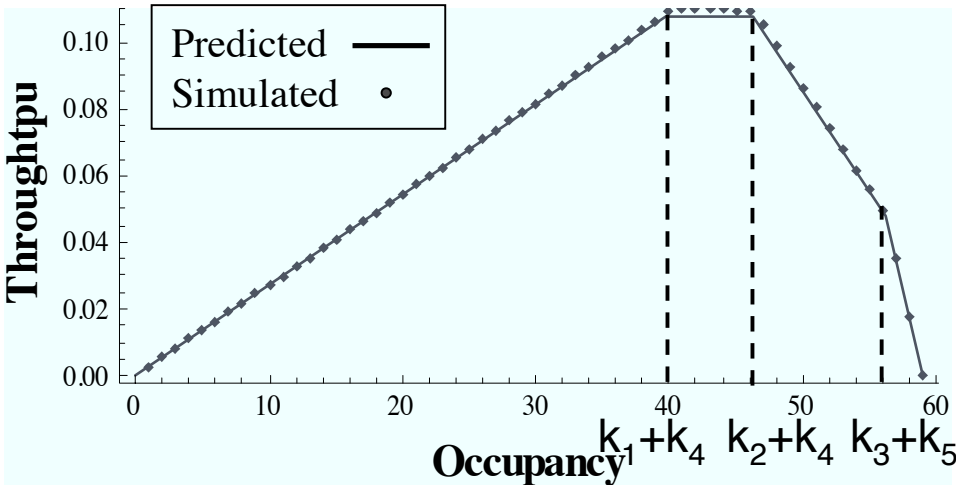


Figure 4.7: Canopy graph for sequential composition

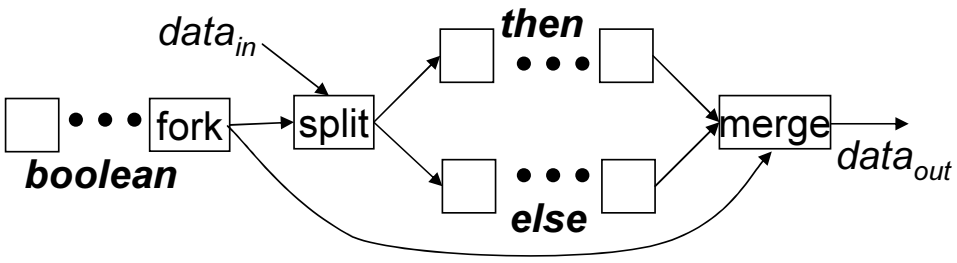


Figure 4.8: A pipelined choice construct

true for the point b. Since sets $\mathbb{C}_0$ and $\mathbb{C}_1$ are canopy graphs, they are themselves convex. This means that all points on line segments $\overline{a_0 b_0}$ and $\overline{a_1 b_1}$ must be included in sets $\mathbb{C}_0 and \mathbb{C}_1$ respectively. The horizontal sum of these two line segments is line segment $\overline{ab}$, which means that all points in line segment $\overline{ab}$ are in $\mathbb{C}_0 + \mathbb{C}_1$.

4.3.3 Conditional Constructs

Canopy graph analysis can also estimate the throughput of pipelined conditionals that are implemented with choice. Such conditionals implement if/then/else blocks by waiting for the Boolean decision value to be ready before beginning computation on one of two paths. Figure 4.8 shows an example of a pipelined conditional. Both the split and merge stages receive the Boolean input, to ensure that ordering is preserved as items exit.

For clarity of presentation, Section 4.3.3 uses two simplifying assumptions that will be relaxed in subsequent sections: 1) the pipeline is slack matched, and 2) the values of the Boolean data are not clustered (*i.e.,* at a probability of 0.5, the Boolean values arrive in this regular pattern: $0, 1, 0, 1...$). Sections 4.3.3 and 4.3.3 relax these restrictions and handle slack mismatch and clustering, respectively.

Canopy Graph Method

Let us first develop a relation between the throughputs and occupancies of the two branches of a conditional. Consider a conditional that has probability p_0 of $branch_0$ being chosen and p_1 of $branch_1$ being chosen. For every item that enters (or leaves) $branch_0$, $\frac{p_1}{p_0}$ items enter (or leave)

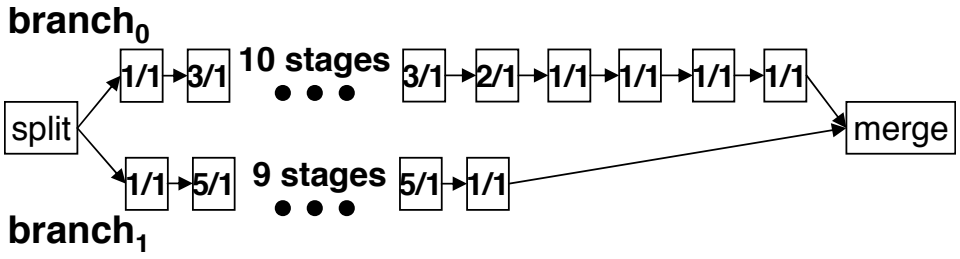


Figure 4.9: A conditional from CRC

$branch_1$. For example, at a probability of $p_0 = 0.2$, for every data item that enters $branch_0$, 4 items enter $branch_1$. Further, the data items must preserve their original order upon leaving the conditional. Therefore, *under steady-state operation,* the occupancies of the two branches, k_0 and k_1, must be proportional to their respective branch probabilities: $\frac{k_0}{p_0} = \frac{k_1}{p_1}$. Similarly, the branch throughputs are proportional to branch probabilities: $\frac{tpt_0}{p_0} = \frac{tpt_1}{p_1}$.

These equations tell us that the steady-state behaviors of the two branches are constrained such that their occupancies and throughputs, when divided by their respective branch probabilities, are equal. This result suggests that the canopy graph of the combined system can be computed by intersecting the canopy graphs of the two branches after appropriate scaling. Specifically, the canopy graph for each branch is scaled so both its axes are divided by the respective branch probability. The area underneath the intersection of the two scaled canopy graphs represents the feasible throughput of the conditional construct.

Figure 4.9 is an example of a conditional found within the cyclic redundancy check (CRC) algorithm. Suppose it is given that, for this conditional, $p_0 = 0.7$ and $p_1 = 0.3$. Figure 4.10 show the canopy graphs for $branch_0$ and $branch_1$ scaled by $\frac{1}{0.7}$ and $\frac{1}{0.3}$, respectively. The intersection of the two scaled graphs shows that the conditional has an overall maximum throughput of 0.36 and a maximum occupancy of 23. Interestingly, these throughput and occupancy values are higher than either of the branches on its own, because the two branches operate on distinct data sets in parallel. Figure 4.10 also shows data points from a Verilog simulation, which agree with the results of our analysis.

In general, the throughput of a conditional is dependent on the branch probabilities. The

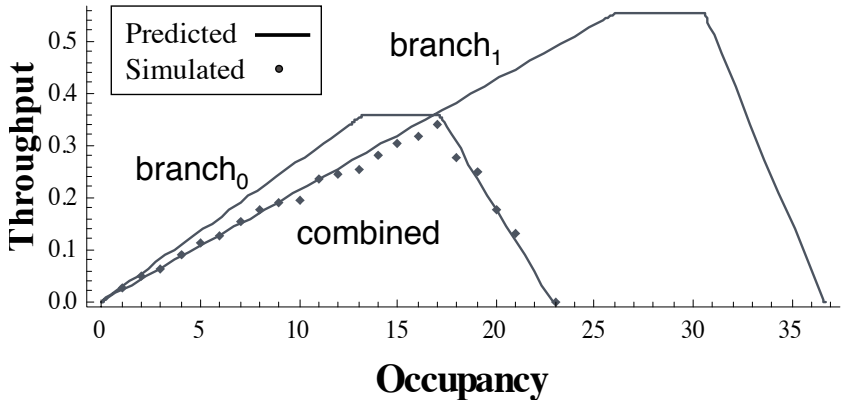


Figure 4.10: Canopy graph for CRC at p1 = 0.3

maximum throughput of each scaled canopy graph limits the throughput of the overall system, as given by:

$$tpt_{max} = min\left(\frac{tpt_0}{p_0}, \frac{tpt_1}{p_1}\right) \tag{4.1}$$

Applying this equation to the example from Figure 4.9 yields the graph in Figure 4.11, which shows the maximum throughput versus the probability of choosing $branch_1$. Data points from Verilog simulation are also shown on this graph, as a validation of Equation 4.1.

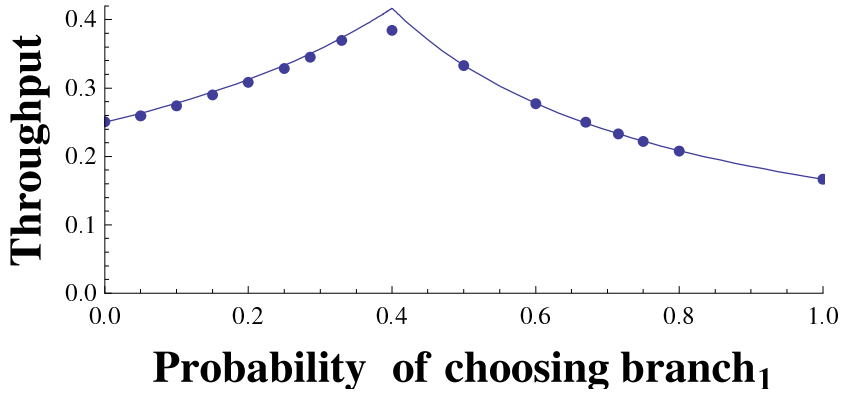


Figure 4.11: Throughput depends on probability

The Effects of Slack Mismatch

Slack mismatch occurs in conditionals when one branch has too few buffer stages, which causes it to become full and cause stalls. The amount of decrease in throughput due to slack

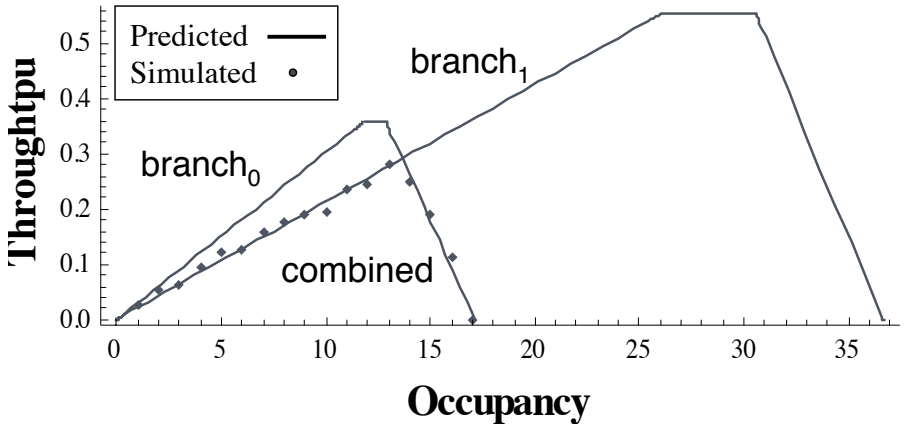


Figure 4.12: Canopy graph for CRC at p1 = 0.3 with slack mismatch

mismatch depends on the probability of each branch being chosen. In particular, as the probability of choosing a branch increases, the frequency of data entering that branch increases, thereby increasing the likelihood that it will become full and cause stalls.

A slack mismatch between the two branches of a conditional can be readily determined by inspecting their canopy graphs. As an illustration, let us return to the example of Figure 4.9, with the last four stages removed from $branch_0$, thereby decreasing its total number of stages to 12. Scaling the canopy graphs based on the probabilities $p_0 = 0.7$ and $p_1 = 0.3$ results in the graph of Figure 4.12. The maximum throughput at which the two graphs intersect is actually lower than the throughput of the slower branch, which indicates that slack mismatch is introducing stalling in the system. In particular, stalling occurs when $branch_0$ becomes full and prevents data from entering $branch_1$. The graph also shows data points from simulation, thereby demonstrating that the throughput decreases according to our predictions.

Figure 4.13 shows the overall throughput of the slack mismatched version of the pipeline of Figure 4.9 (*i.e.,* four stages removed from $branch_0$) as a function of the branch probability. For comparison, refer to Figure 4.11 again for the throughput for the slack matched version. The comparison shows that, as expected, throughput degradation occurs when $branch_0$ is likely to be chosen, and no throughput degradation occurs when $branch_1$ is likely to be chosen.[1]

[1]Notice that in the neighborhood of $p_1 = 0$ our analysis deviates slightly from the simulation results; in this

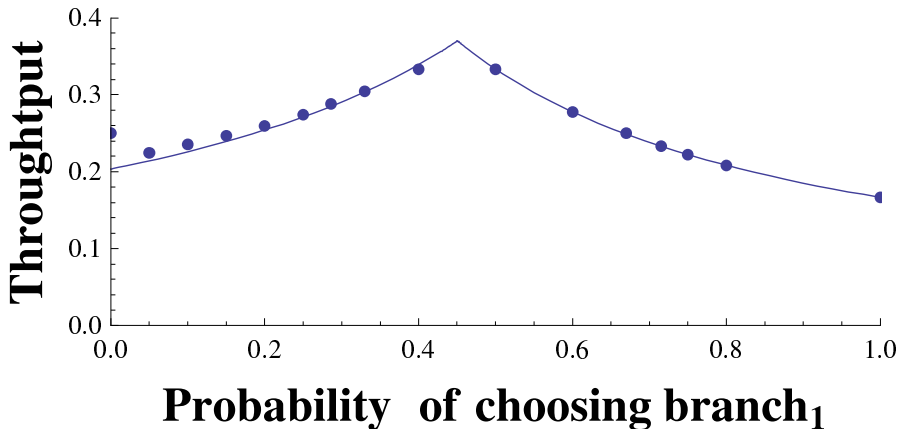


Figure 4.13: Slack mismatch leads to throughput degradation

Clustering of Boolean Values

This section introduces a more powerful modeling of choice that considers *correlation* between successive Boolean outcomes, which is captured in a metric termed the "cluster factor". Intuitively, the clustering factor quantifies the average run lengths of 1's and 0's in a sequence, normalized with respect to the base case given in Section 4.3.3 This quantity effectively models correlation between successive Boolean outcomes. A clustering factor of 1 indicates a negative correlation, which is the simple case considered in the previous sections. A high cluster factor indicates a positive correlation, in which long runs of 1's and 0's may occur. For zero correlation (*i.e.,* independent outcomes) the clustering factor is related to the probability. Specifically, the expected run length of ones for random, uncorrelated data is $E(run_1) = 1/p_0$.[2]

During circuit operation, clustering prevents throughput from reaching the maximum value predicted using the canopy graph method described in Section 4.3.3. The effective maximum occupancy also decreases, since clustered Boolean values lead to uneven filling of the two branches. However, the slopes of the canopy graph lines, which are determined by the forward and reverse latencies of the pipelines, do not change due to clustering.

---

region, the steady-state assumption does not hold.

[2]Expected value is found by summing this infinite geometric series $\sum_{n=0}^{\infty} p_1^n = \frac{1}{1-p_1}$

The reduced maximum throughput can be written as a modified version of maximum throughput Equation 4.1, with the cluster factor now included. This equation as written considers the situation in which the throughput of $branch_0$ is higher. In this Equation, run is the run length based on probability and $cluster$ is an estimate for the clustering factor, based on knowledge of typical input data sets.

$$tpt_{max} = \frac{cluster \cdot (run_0 + 1)}{cluster \cdot run_0 \cdot \frac{1}{tpt_0} + cluster - 1 \cdot \frac{1}{tpt_1}} \tag{4.2}$$

Applying this equation to the example of Figure 4.9 and assuming random, uncorrelated Boolean values, the solid line in Figure 4.14 shows how the maximum throughput varies with the probability of choosing $branch_1$. For comparison, it also shows the maximum throughput with a clustering factor of one (*i.e.,* completely anti-correlated Boolean values). In addition, the figure shows data from Verilog simulation, which is close to our predicted curve.

Clustering also affects the average maximum occupancy of the pipeline. Simple statistical methods determine the expected maximum occupancy of a pipeline with clustering—the expected value is found by summing the probability of each occupancy occurring multiplied by that occupancy.

Returning to the example of Figure 4.9, the canopy graph of the circuit is affected by clustering. If $p_1 = 0.3$ and the Boolean values are random and uncorrelated, Equation 4.2 predicts that clustering will reduce the throughput to 0.30. Figure 4.15 shows how the canopy graph will be cut off at the value 0.30. In this example, the expected maximum occupancy due to clustering is 22.75, which is only slightly lower than the non-clustered maximum occupancy. Notice that the slopes of the boundary lines remain the same, although the throughput is limited by the predicted value of 0.30.

Convexity Property

To prove that the canopy graph for a conditional is convex, it is sufficient to prove that the scaling operation introduced in this section preserves convexity. Once the canopy is scaled,

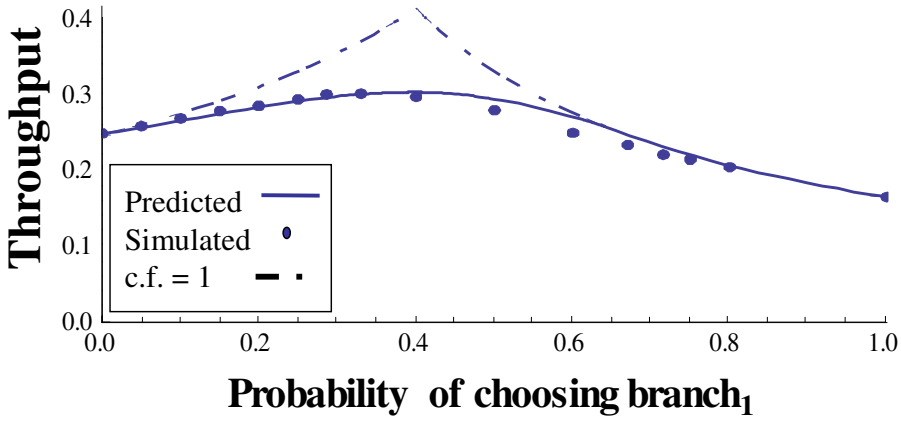


Figure 4.14: Clustering decreases throughput

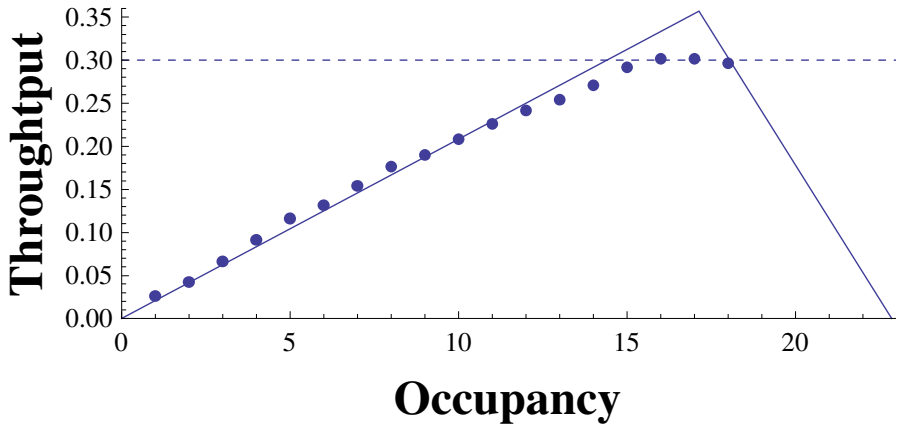


Figure 4.15: Clustering decreases max throughput

the next operation is intersection, which has already been shown to preserve convexity in Section 4.3.1. If the unscaled canopy graph is convex, then all points along a line between any two points in a and b will be elements of that set. Scaling the canopy graph is the equivalent to multiplying all points by the same constant multiple; this will not affect the linear relationship between a, b, and the points in between.

4.3.4 Iterative Constructs

Loops in high level code are implemented as rings in hardware. Although the classic work by Williams [67] studied the performance of rings, it finds the throughput within the ring whereas our technique finds the throughput of items leaving a ring based on some condition.

Traditional hardware design methods typically allow only a single token inside a ring. This limits the performance, but avoids the complications of allowing multiple data tokens within the ring. As will be explained further in Section 5.3.3, my work presents an approach to implementing *for* and *while* loops that operate on multiple tokens concurrently. This loop pipelining technique handles the flow of control and data dependency challenges created by allowing multiple tokens in the ring by including a special ring interface and duplicating data values within the ring. A monitor in the interface prevents overfilling of the loop by limiting its occupancy to some optimal value, k.

To analyze the performance of a loop, begin with the a canopy graph for the body of the loop. For traditionally implemented loops that contain only one token, cut the canopy graph off at occupancy 1. Similarly, for a pipelined loop with a maximum occupancy of k, cut the canopy graph off at k. Since the canopy graph of an algorithmic loop should report the rate at which tokens leave the ring, rather than the frequency of the data within the ring, the canopy graph is simply scaled down by the number of iterations each token undergoes. If the iteration count is variable or data-dependent, our approach approximates the throughput using the expected number of iterations. The model for expected occupancy is that a weighted coin-toss takes place at the beginning of each iteration.

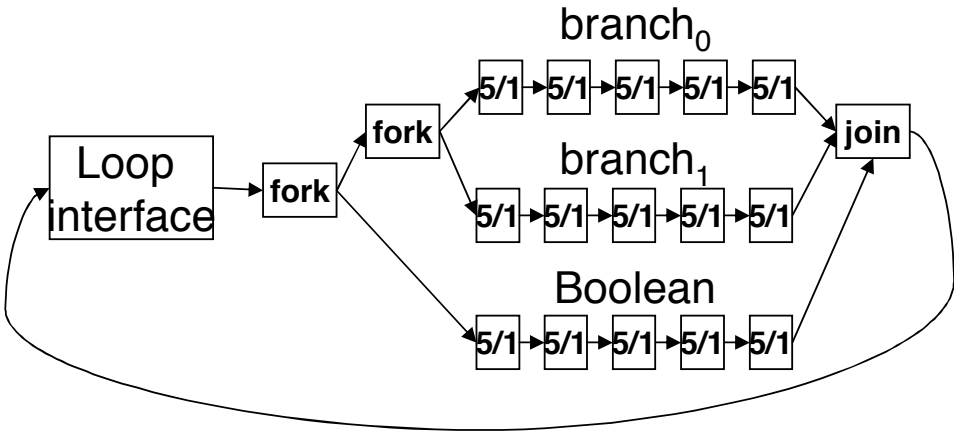


Figure 4.16: Pipelined GCD loop

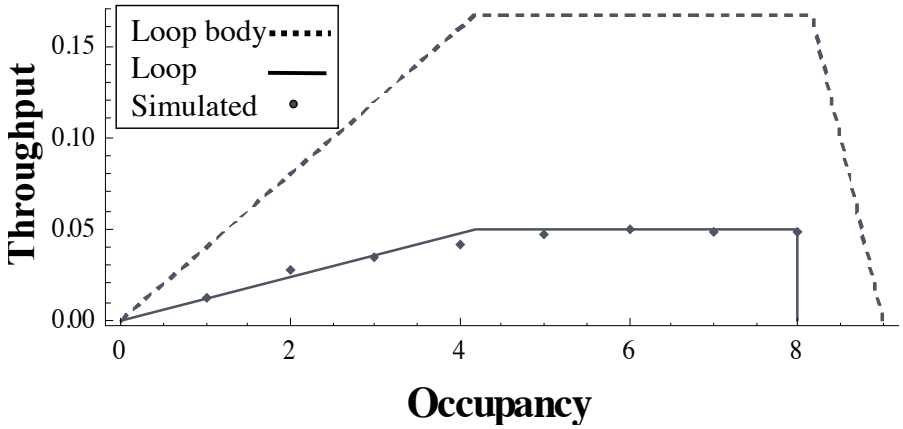


Figure 4.17: Canopy graph analysis for GCD

As an example, Figure 4.16 shows a pipeline ring that implements an iterative algorithm for finding the greatest common divisor of two numbers. The ideal occupancy of this ring is eight, so the interface will allow data items into the ring concurrently. The probability of a token leaving after completing an iteration is 0.3, which means that the expected value of the iteration count is 3.33. For the loop body, the same process given in Section 4.3.1 produces to produces the canopy graph shown in Figure 4.17. The canopy graph is scaled down by the expected iteration count, 3.33, and cut off at the max occupancy, eight. Figure 4.17 shows the resulting canopy graph. The figure also shows data from Verilog simulation, which follow closely with our predicted canopy graph.

Convexity Property

As discussed in Section 4.3.3, convexity is maintained under scaling. The canopy undergoes one further operation, which is cutting it off at some occupancy. This can be cast as the intersection of two convex sets. Construct a set of points that is bounded by a square set of points that has the domain from zero to the ideal occupancy of the ring, and range from zero to the maximum throughput of the ring. Intersecting this new convex polygon with the canopy graph will effectively cut off the occupancy at the ideal occupancy, and—since the intersection of two complex polygons is itself convex—the result of this intersection must be convex.

4.4 Algorithm Description

```
CanopyGraph analyze( tree t )
 if( currentNode == parallel )
   return parallel(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == if )
   return conditional(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == loop )
   return loopize(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == sequence )
   return sequence(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == leafnode )
   return CanopyGraph( Lf, Lr )
```

Figure 4.18: Our analysis algorithm

The composition functions described in Section 4.3 come together to form a complete algorithm for finding the throughput of a hierarchically composed pipelined system. Figure 4.18 the shows pseudocode for the algorithm, which generates the canopy graph for the whole system by traversing the hierarchy.

The algorithm takes as input the description of a pipelined system in the form of a syntax tree. The tree represents the hierarchy of the high-level description, and is also annotated to indicate the delays of the stages in the pipelined implementation. In addition to the system representation, the algorithm must also have access to the branching probability for each

if/then/else construct and while loop conditional. The algorithm proceeds much like expression evaluation: two throughput expressions (*i.e.*canopy graphs) are combined based on their parent operator.

```
Diffeq(x, y, dx, u, a)
1 while ( x < a )
2 (x1 := x+dx;
3 t1 := u*x;
4 t2 := t1 + y)
||
5 (t3 := 3*dx;
6 t5 := u*dx;
7 y1 := y+5)
8 t4 := t2*t3;
9 u1 := u - t4
<x1, u1, y1> = <x, u, y>
```

Figure 4.19: Differential equation solver

As an example, Figure 4.19 shows the high-level code for an iterative differential equation solver, which has been parallelized. Figure 4.21 shows one possible pipelined representation and Figure 4.20 shows an abstract syntax tree representation. The abstract syntax tree is annotated with the pipeline stage it corresponds to, which indicates the stages forward and reverse latency. The analysis of this example serves as an example of how the algorithm exploits this hierarchical structure to find the canopy graph for the entire system in a single pass. At each step shown, figures plot the data from Verilog simulation as points together with the predicted canopy graph; this indicates the accuracy of our predictions at each level of the hierarchy.

The algorithm traverses the tree; when it encounters the parallel operator, the algorithm finds the two canopy graphs of the two branches $\{$s2, s3, s4$\}$ and $\{$s5, s6, s7$\}$. These two canopy graphs are shown together in Figure 4.22. Since the composition algorithm for parallel structures, given in Section 4.3.1, calls for intersection of canopy graphs, the algorithm creates a new canopy graph that is the intersection of these two. This graph is then composed in sequence with the remaining stages, using the algorithm given in Section 4.3.2, to give the

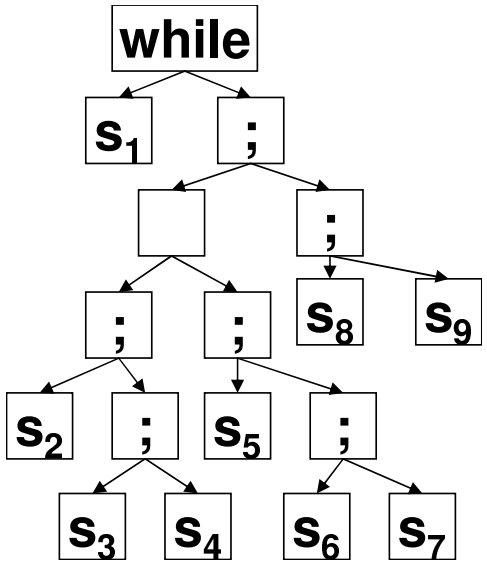


Figure 4.20: Abstract syntax tree

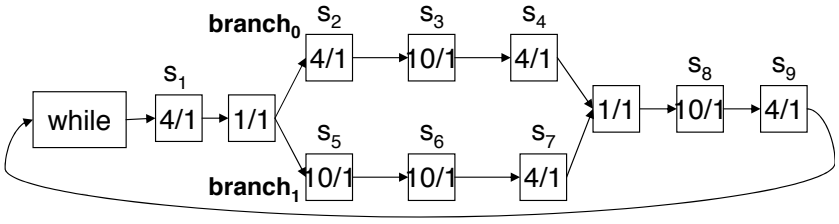


Figure 4.21: Pipelined implementation of Diffeq

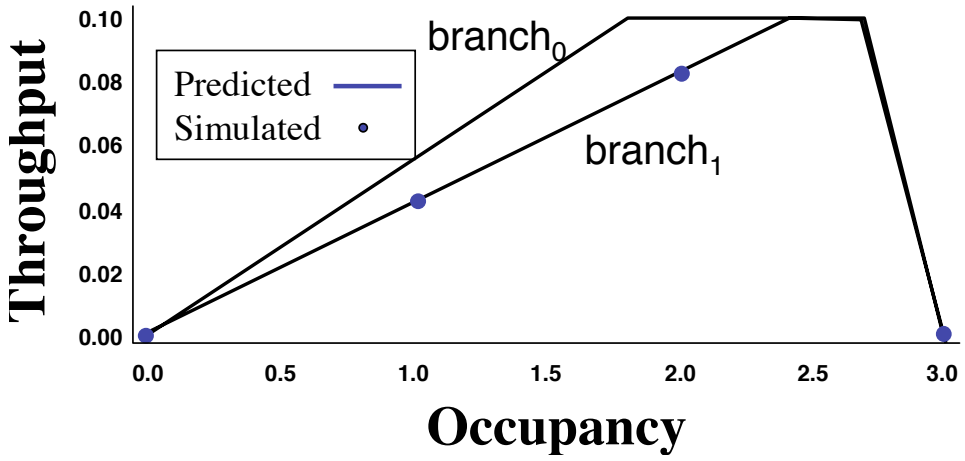


Figure 4.22: Parallel composition of $branch_0$ and $branch_1$

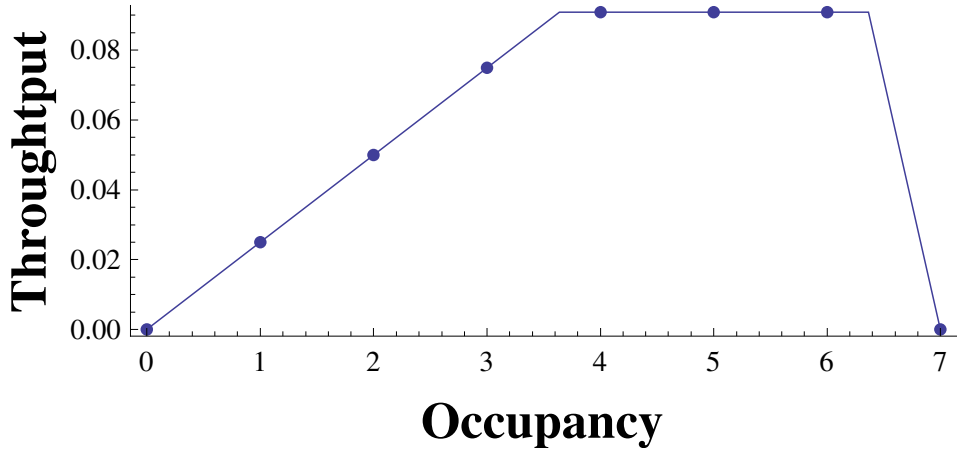


Figure 4.23: Canopy graph for loop body

graph shown in Figure 4.23. Finally, the method for finding the canopy graph of a loop, given in Section 4.3.4, produces the canopy graph shown in Figure 4.24. In this way, our algorithm can handle any system with arbitrary levels of nesting, by working from the bottom up.

4.5 Results

We have implemented the algorithm given in Section 4.4 in Java, and run it on several examples using a Pentium 4 2.4GHz CPU. Section 4.3 gave several examples of our method for predicting the throughput of pipelined architectures, many of which were part of larger algorithms. In this section, we summarize those results, and also add results for larger systems.

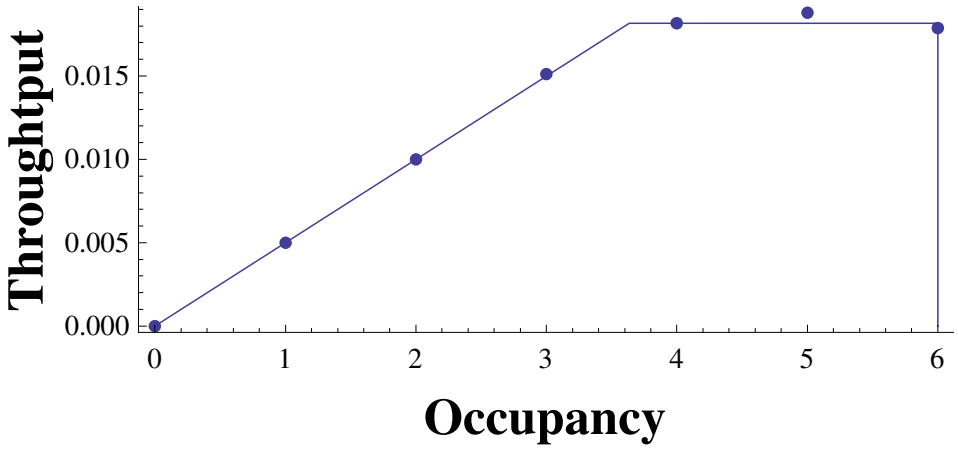


Figure 4.24: Canopy graph for pipelined loop

We also add two new examples. Raytracing—a pipelined implementation of a ray-sphere intersection algorithm—performs an iterative (*i.e.*loop pipelined) square root based on some data-dependent condition. The other new example is multiplier that does a shift instead of a full multiply if the multiplicand is a power of 2. The seven examples and the type of hierarchical composition they use are the following: **1)** CORDIC: parallel and sequential **2)** CRC: conditional and sequential **3)** GCD: nested parallel and loop **4)** Sequential composition from Figure 4.5: parallel, sequential **5)** DIFFEQ: parallel, sequential, and loop **6)** MULT: parallel **7)** Raytracing: Conditional with nested loop

Table 4.1 compares the throughput found through switch-level Verilog simulation to the throughput predicted by our method and gives the percent error. In the simulation, each individual pipeline stage is modeled at the gate level, and delays are assigned based on the complexity of the logic present. Delays are normalized such that an individual stage has a forward latency of 1; stages with more complex logic are modeled with commensurately higher delays.

In the table, the throughput of CORDIC and CRC examples are reported for a section of the pipeline, and the throughput for the entire system. For CRC, all results are shown for $p1 = .3$. We find that our predictions are within 2.2% of simulated results for all examples. Most notably, our method accurately predicts the maximum throughput of the large, complex

raytracing example with negligible error. The largest error occurs in the sequential example from Section 4.3.2, although the estimate is still quite accurate, as shown in Figure 4.7.

In addition to reporting the accuracy of our method, Table 4.1 also lists the size of each example in number of pipeline stages and the approximate run time. Even for large examples, the run time never exceeds 10 ms.

These results indicate that our method is sufficiently accurate and fast for use within a system-level optimization loop.

4.6 Conclusion

This chapter presented a fast, accurate analytic method for estimating the throughput of pipelined asynchronous systems. The analysis method derives its speed from the observation that many circuits have a naturally hierarchical structure consisting of a hierarchical composition of sequential constructs, parallel (fork-join) constructs, conditional constructs (if-then-else), and iteration (pipelined 'for' and 'while' loops). This class of systems is expressive enough that we could apply our technique to pipelined implementations of several non-trivial examples, yet predictable enough in structure and operation that an analytic (rather than simulation-based) analysis method is possible. The analysis method was accurate to within $\tilde{2}\%$ for the examples presented in this chapter.

Table 4.1: Predicted throughput compared to simulation

Example	Simulated	Predicted	Size	Time	%error
CORDIC cond	0.167	0.167	31	$\sim$ 10 ms	$\sim$0
CORDIC cond mis	0.0909	0.0909	23	$\sim$ 10 ms	$\sim$0
CORDIC full	0.0909	0.0909	44	$\sim$ 10 ms	$\sim$0
CRC cond	0.352	0.357	27	$\sim$ 10 ms	1.4
CRC cond mis	0.292	0.286	23	$\sim$ 10 ms	2.0
CRC cond cluster	0.305	0.300	27	$\sim$ 10 ms	1.6
CRC full	0.333	0.333	67	$\sim$ 10 ms	$\sim$0
DIFFEQ	0.0183	0.0182	10	$\sim$ 0 ms	.5
GCD	0.0490	0.0500	21	$\sim$ 10 ms	2.0
raytracing	.222	.222	168	$\sim$ 10 ms	$\sim$0
MULT	0.0387	0.0384	13	$\sim$ 0 ms	1.9
sequential	0.110	0.107	109	$\sim$ 10 ms	2.2

Chapter 5

Optimization and Trade-off Exploration

5.1 Introduction

Optimization and design space exploration are critical to any design flow that aims at producing high-performance systems. In the absence of a fast automated optimization tool, a designer typically must contend with costly simulation to search for bottlenecks, a time-consuming and error-prone procedure that can make the typical design-analyze-optimize cycle in the design flow quite inefficient. The analysis method presented in Chapter 4 is fast and accurate enough to be used repeatedly within the optimization or design space exploration portion of a toolflow. Even with the full analysis information, however, optimization is not trivial. The component or set of components that limit the speed of the system, also known as the system bottlenecks, can be difficult to find and correct, because their causes are diverse and often subtle. Merely finding local cycle times or analyzing canopy graphs of sub-systems individually is not sufficient, because this will not always account for complex system-level interactions (*e.g.*, systems with choice, feedback due to algorithmic loops, etc.).

For an optimization system to work well with the canopy-graph-based analysis method, optimizing transforms must first be characterized in terms of their effects on the canopy graphs. Section 5.3 lists and categorizes a set of optimizing transformations and describes qualitatively how each will affect the canopy graph of a circuit. In addition, this section goes

into detail on two of the optimizations: loop unrolling and buffer insertion. In particular, It introduces loop pipelining, which is a new microarchitectural optimization for reducing the bottleneck caused by algorithmic loops, and discusses slack matching in depth.

The key to fast optimization is to keep and use, rather than discard, all performance information generated during intermediate steps of the analysis approach. Bottleneck detection consists of pinpointing the precise portion of the architecture that is limiting the achievable throughput. Once a bottleneck has been found, our technique systematically assists the designer to apply certain structural transformations to alleviate the bottleneck. Section 5.4 explains my approach for using all the information generated during one analysis pass to locate the stage or sets of stages that cause a bottleneck in the system.

Once the bottlenecks have been identified, the difficult task of choosing how to remove them remains. Section 5.5 details two methods for removing bottlenecks once they have been identified. The first is a user-guided method, which suggests optimizations using information from the analysis method but relies on the user to choose and perform the optimizations. It can be seen as a way for a designer to gain intuition and get expert advise about the best way to complete hand optimization. The second method is automated by casting the problem as one of constrained optimization: reach some throughput goal while minimizing some secondary criteria such as energy or area.

5.2 Background on Optimizing Asynchronous Systems

5.2.1 Previous Work on Optimizing Transformations

Buffer Insertion

The slack matching methods presented by Beerel [3] and Prakash [47] give provably optimal solutions, both in terms of fewest buffer stages added and maximum throughput attained. However, since these methods are based on mixed integer linear programming (MILP) for-

mulations, they can be slow for large examples. Moreover, neither of these methods handles architectures with conditional computation.

The work by Venkataramani [66] does handle conditional execution in the form of conditional Petri nets. The method is simulation-based and requires a gate level circuit as input. The strategy is to simulate repeatedly to find bottlenecks, adding buffer stages each time. It is not proven to find the optimal solution, either in terms of fewest buffer stages or maximum throughput attained.

Loop Pipelining

Several approaches in hardware design have attempted to optimize iterative computation. However, their main objective is to reduce the *latency* of the loop, as opposed to improving its throughput. One such approach is the CASH compiler by Budiu and Goldstein [8], which translates an ANSI C program into data-flow hardware. A different approach by Theobald and Nowick [63] targets generation of distributed asynchronous control from control-data-flow graphs, with the objective of optimizing communication between the controllers, and between a controller and its associated datapath object.

The above approaches offer very limited throughput benefit, since they are mainly targeted to improving a loop's latency. In particular, the optimizations introduced by these approaches focus on shaving off some delays from each loop iteration, and offer modest concurrency increases: the tail-end of an iteration for a given data item is allowed to overlap somewhat with the start of that same item's next iteration, or the tail-end of an item's last iteration overlaps somewhat the start of the next item's first iteration. Neither approach allows multiple distinct data items to be truly processed concurrently. Hence, the throughput benefits of these approaches are modest, and nowhere in the 2–10x range targeted by loop pipelining.

Like loop pipelining approach, work by Kapasi, Dally et al. [24] targets efficient implementation of stream algorithms. However, it does not address the critical challenge of pipelining loops and only addresses the problems related to conditional branches.

5.2.2 Previous Work on Optimization and Bottleneck Removal

While there has been much work on asynchronous performance analysis, the problem of bottleneck analysis has so far not been adequately addressed. A recent approach by Venkataramani *et al.* [65] introduces the notion of a *global critical path* in a system, and uses simulation and profiling to help the designer identify targets for optimization. Although the approach can be useful in identifying bottlenecks, its reliance on simulation can make it time-consuming.

Other prior approaches do not directly target bottleneck identification, but focus instead on finding a system's peak achievable throughput. These include (i) simulation-based approaches [9, 37, 70]; (ii) Markov analysis methods [29, 36, 71]; (iii) methods based on graph unfolding [22, 10]; and (iv) closed-form analytical solutions [67, 18, 44, 31]. The simulation, Markov analysis, and graph unfolding methods all tend to require long running times. The closed-form solutions, on the other hand, only apply to a limited set of architectures (*e.g.,* rings, meshes, linear and simple fork-join pipelines). Recently, a graph-theoretic approach was proposed that avoids graph unfolding to achieve quite fast runtimes [35]. However, all of the aforementioned approaches that are not simulation-based cannot handle systems with choice, thereby limiting their applicability to systems without conditionals or data-dependent loops. Simulation-based approaches generally are able to handle choice, but require long runtimes.

5.3 Transformations for Bottleneck Alleviation

This section describes optimizing transforms in terms of their effect on the canopy graph of the system as a whole and introduces the notion of a TRIC: a TRansformation for Increasing the Canopy. A circuit transformation is can be classified as a TRIC if it has the potential to expand the canopy in some way (*i.e.*the transformation increases the throughput at some occupancy.) Although many of the TRICs described in this section are in common use, this is the first time that their effects on system canopy graphs has been described. Subsection 5.3.1 details eight

different TRICs that can be used in system optimization. In addition, this section presents novel work on two kinds of TRICS: buffer insertion and loop unrolling. Subsection 5.3.2 describes an automated method for applying the buffer insertion TRIC to improve throughput in a circuit while Subsection 5.3.3 addresses the challenges of loop pipelining and describes the benefits of loop unrolling.

5.3.1 Bag of TRICS

Each of these TRICs raises the throughput for some range of occupancies. Although many of these TRICs has been used before, they have not previously been analyzed relative to their effects on the canopy graph. This section describes how each TRIC affects the canopy graph and lists the type of hierarchical component that each TRIC can apply to. This list is not intended to be exhaustive but rather illustrative of describing common optimization techniques in terms of their effects on the canopy graph.

Coalescing stages

Two adjacent, sequential pipeline stages can be grouped together into one stage, thereby reducing the total forward latency by removing some latches from the forward path. As indicated in Figure 5.1, the reduced forward latency leads to expansion of the canopy graph to alleviate Type I bottlenecks. However, this transformation may also lead to an increased maximum cycle time and also reduces the total effective occupancy. Additionally, this TRIC applies only to adjacent leaf nodes composed in sequence.

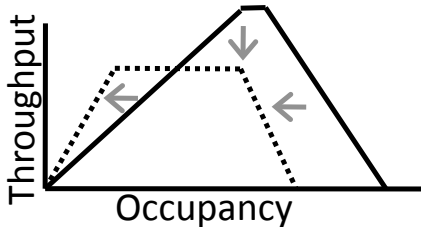


Figure 5.1: Coalescing

Parallelization

As described in [20], sequential computations can sometimes be changed to occur in parallel, based on the dependency graph of the computations. As indicated in Figure 5.2, this reduces the forward latency by splitting one sequential path into two parallel paths. However, it also reduces the occupancy of the system. Additionally, this TRIC applies only to nodes that are currently composed in series and has the further restriction that the data dependencies between the nodes must allow for parallel execution.

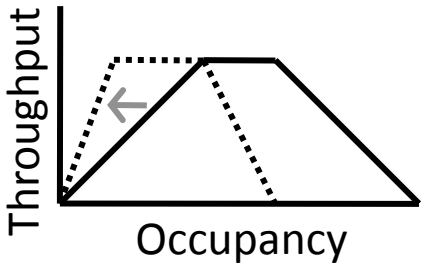


Figure 5.2: Parallelization

Stage Splitting

Stage splitting increases the level of pipelining by taking a high-latency stage and splitting it into two, lower latency stages. As indicated in Figure 5.3 this leads to an increase in throughput due to the decreased cycle time of the lower latency stages. It also increases the occupancy of the system by adding a new stage. However, it also increase the forward latency because the stage overhead (*i.e.* latches and stage controllers) increases due to the fine-grained pipelining. Additionally, this TRIC applies only to leaf nodes.

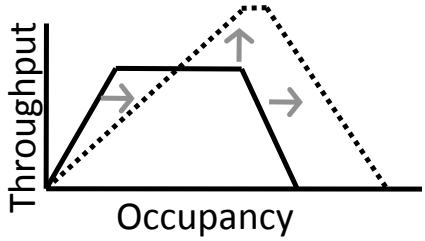


Figure 5.3: Stage Splitting

Loop Pipelining

Loop pipelining is a method for breaking up a high-latency loop, which essentially acts as a single slow stage within the system. The details of the technique of loop pipelining are a contribution of this thesis, and will be further detailed in Subsection 5.3.3. Although the details are more complex than stage splitting, the result is similar and the illustration of Figure 5.3 that was used for stage splitting applies for loop pipelining as well. In particular, loop pipelining will worsen the forward latency of the loop improving the throughput and increasing the maximum occupancy. This TRIC, of course, applies only to loop nodes.

Duplication with Wagging

A wagging buffer [5] improves throughput by sending data alternatingly along one path and then the other. A node and all its children must be fully duplicated in order to apply the wagging buffer method. As illustrated in Figure 5.4, applying this TRIC increases the throughput as well as the occupancy. It has a second order effect of increasing the forward latency due to the overheads of the wagging buffer, but this effect is constant regardless of the amount of logic duplicated. Though this is not captured in the canopy graph, it should be noted that this TRIC has a significant area overhead and therefore cannot be repeated limitlessly in the presence of area restrictions. This TRIC can be applied at any type of node.

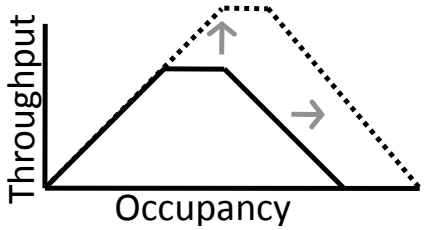


Figure 5.4: Duplication with Wagging

Loop Unrolling

Loop unrolling [16] improves throughput by increasing the number of algorithm iterations that a data item completes during one trip through the ring, thereby decreasing the total number of times that each item must cycle through ring. Loop unrolling is a type of duplication, and has the same effects on the canopy graph as illustrated in Figure 5.4. Specifically, the throughput and occupancy increase and a small overhead for forward latency occurs. Additionally, this TRIC can be applied only to loop nodes.

Buffer Stage Insertion

Buffer stage insertion is a common technique for improving throughput, because it is used to

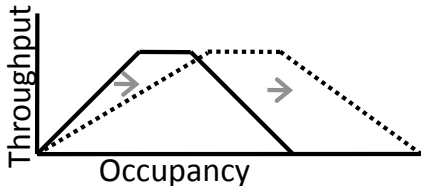


Figure 5.5: Buffer Insertion

accomplish slack matching [3] [47]. As illustrated in Figure 5.5 adding buffers increases the occupancy but also worsens the forward latency. It also increases the reverse latency, but as long as the buffer has a revere latency that is no worse than the other stages in the pipeline, adding this stage will not result in a decrease in throughput anywhere in the hole-limited region. This TRIC can be applied at any type of node. Buffer insertion and its use to improve throughput are further elaborated on in Section 5.3.2.

5.3.2 Buffer Insertion Details

When the TRIC buffer insertion is the only one made available to an optimization algorithm, the problem reduces to the one of *slack matching*. Slack matching is the problem of determining the number and positioning of additional pipeline buffers in a pipelined design in order to

reach some performance goal. Both a mixed integer linear programming solution, which is NP-hard but optimal, and a heuristic algorithm, which is faster but not provably optimal, can be created based on the analysis method of Chapter 4.

Slack Matching Problem

Buffer insertion is able to improve the throughput of the system if a lack of space for data tokens, or a lack of slack, creates a bottleneck within the system. The essence of the problem behind slack matching can be illustrated in a simple example. Consider the fork/join microarchitecture shown in Figure 5.6. Figure 5.7 shows the canopies of $path_0$ and $path_1$ plotted together. The maximum throughput possible is the lower of the two peak throughputs, as indicated on the graph. Since the throughput is constrained to lie under *both* canopies, the actual throughput achieved is the point of intersection of the two canopy graphs; this point is lower than the ideal maximum.

In this simple case, it is clear that one way to improve throughput is to add slack (*i.e.* one buffer stage) to $path_1$. Figure 5.8 illustrates how adding one buffer stage accomplished slack matching.[1] Adding an extra buffer stage adds to both the forward and the reverse latencies of that pipeline, thereby decreasing the slope of both the forward and the reverse lines in its canopy graph. More importantly, adding buffer stages lengthens the base of the canopy because it allows greater occupancy. As a result, the two canopy graphs are now able to intersect at a point which corresponds to the peak throughput of the slower pipeline, thereby eliminating the overhead of slack mismatch.

For larger systems, the solution is not always as clear as it was for this small example. In particular, adding a buffer stage increases the forward latency and the reverse latency along at least one path. An increase in latency equates to shallower slopes in the resulting canopy graphs, which can in turn create a new slack mismatch situation with a different part of the circuit. Because fixing a local slack mismatch can have negative global consequences to sys-

[1] Calculations assume a fast buffer stage with forward and reverse latencies of 1.

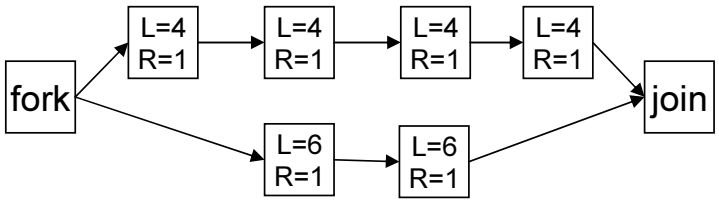


Figure 5.6: Slack mismatch in a fork/join construct

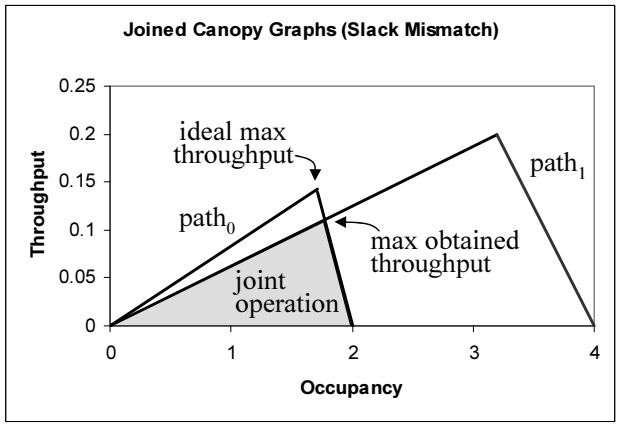


Figure 5.7: Canopy graphs showing slack mismatch

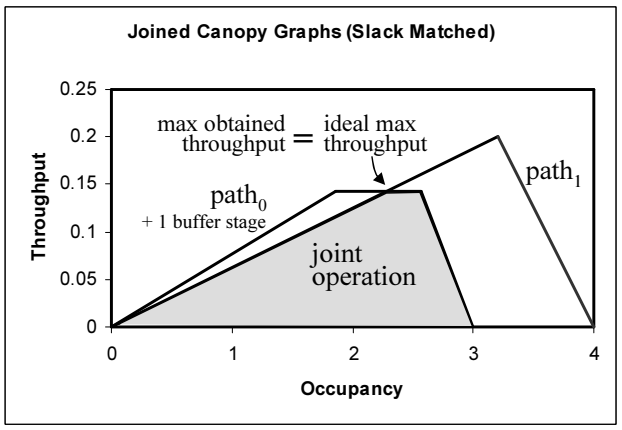


Figure 5.8: Canopy graphs with slack matching

tem performance, finding the optimal placement for buffers that minimizes the total number of buffers placed while meeting some goal throughput is a challenging problem.

MILP Formulation

Slack matching using canopy graph analysis seems to lend itself quite naturally to a linear programming solution. The formulation of a linear programming problem requires a function to minimize: in this case, the total number of buffer stages added (*i.e.* $s_1 + s_2 + s_3 \ldots + s_n$). A linear programming formulation also requires a set of constraints to satisfy, in the form of inequalities. In the case of slack matching the throughput is limited to be beneath the lines of the canopy graphs. This section defines canopy graphs as a set of inequalities and formalizes the composition functions that operate the canopy graphs. These representations explicitly include the effects of slack matching buffers.

Although equations for calculating the sequential and parallel dynamic slack are given in

the previous work by [31], they are reformulated here to do all of the following: (i) represent non-trapezoidal canopy graphs, (ii) explicitly include the effects of slack matching stages, and (iii) cast them in a form suitable for an MILP solver.

Canopy Graph Equations Inequalities of the following form can represent any canopy graph, cg_i. In particular, each canopy graph representation consists of one inequality in the form of 5.1 that forms the horizontal cutoff line, one in the form of 5.2 that forms the positive sloped forward line, and one or more in the form of 5.3 that represent the negative sloped reverse lines.

$$tpt_i \leq \frac{1}{T_i} \tag{5.1}$$

$$tpt_i \leq \frac{k}{F_i + s_i \cdot f_s} \tag{5.2}$$

$$tpt_i \leq \frac{-k}{R + s_i \cdot r_s} + \frac{N_i + s_i}{R_i + s_i \cdot r_s} \tag{5.3}$$

where T_i is the highest local cycle time of the stages in cd_i, F_i is the sum of the forward latencies of stages in cg_i, R_i is the sum of the reverse latencies of stages in cg_i, N_i is the number of stages in the pipeline, s_i is the number of slack stages to add to cg_i, and f_s and f_r are the forward and reverse delays of a slack stage.

Composition Functions Canopy graph composition functions operate on the inequalities that represent the canopy graphs. The conditional and loop transforms consist of scaling and line intersection. Parallel composition consists of line intersection while sequential composition is equivalent to adding the inverse of two line functions. The following gives composition functions in terms of forward and reverse delays, number of stages, and number of slack stages added.

Conditional Composition. Conditional composition consists of scaling two canopy graphs based on their probabilities, as discussed in Section 4.3.3. In order to represent canopy graph cg_3 that is formed by composing two canopy graphs, cg_1 and cg_2, in a conditional construct

scale the canopy graph inequalities for cg_1 and cg_2 by dividing the right side of the inequalities of Eq. 5.1–5.3 by the probability of each branch being chosen.

In addition, if clustering is present, as discussed in Section 4.3.3 further modifications are needed: a new maximum throughput is added and the effective occupancy changes based on the expected occupancy. In particular, the new maximum throughput is determined using the probabilities of each branch being chosen along with the current maximum throughput of the two branches, as given in Eq. 4.2. Also, the effect that new slack stages have on the increase in occupancy will be changed based on the expected occupancy. That is, inequalities of the form 5.3 should be re-written as follows to take into account the expected occupancy equation.

$$tpt_i \leq \frac{-k}{R + s_i \cdot r_s} + \frac{expected(p_i, N_i + s_i))}{R_i + r_i \cdot r_s} \tag{5.4}$$

Note that while the effective occupancy increase caused by adding slack stages decreases based on the expected occupancy, the effect on the reverse latency remains the same. The modified inequalities can be used in further hierarchical composition steps in the same way as the original equations.

Loop Composition. Loop composition, discussed in Section 4.3.4, is accomplished by scaling the canopy graph, cg_1, of the loop body and applying a cutoff token capacity. To scale the canopy graph inequalities, simply divide the right side of the inequalities that represent canopy graph cg_1, as shown in the inequalities of Eq. 5.1–5.3, by the expected number of loop iterations. In addition, a cutoff line indicating the limited occupancy, based on the designer specified loop capacity, must be added.

$$N_i \leq K_i \tag{5.5}$$

where K_i is the maximum occupancy of the loop represented by canopy graph i.

Parallel Composition. As shown in Section 4.3.1 two pipelines structures in parallel have a combined canopy graph that in the area under both canopy graphs. In order to represent

canopy graph cg_3 that is formed by composing two canopy graphs, cg_1 and cg_2, in parallel, it is sufficient to simply retain all of the inequalities of Eq. 5.1–5.3 and 5.5 of both cg_1 and cg_2.

Sequential Composition. As discussed in Section 4.3.2, putting two pipelined structures in series causes the total number of tokens at each throughput to be added. The following inequalities—Eq. 5.6, 5.7, and 5.8—represent part of the canopy graph, cg_3, that is formed by composing two canopy graphs, cg_1 and cg_2, in sequence. The are formed using the canopy graph inequalities of Eq. 5.1–5.3 respectively.

$$tpt_3 \leq \frac{1}{max(T_1, T_2)} \tag{5.6}$$

$$tpt_3 \leq \frac{k}{F_1 + s_1 \cdot f_s + F_2 + s_2 \cdot f_s} \tag{5.7}$$

$$tpt_3 \leq \frac{k}{R_1 + s_1 \cdot r_s + R_2 + s_2 \cdot rs} + \frac{N_1}{R_1} + \frac{N_2}{R_2} \tag{5.8}$$

Note that any occupancy cutoff line of form 5.5 introduced through loop composition can actually be handled using inequality 5.8. In particular, the cutoff line has no reverse latency R and is not affected by the number of slack stages s, so both of those terms are zero within the inequality. To find all of the inequalities that represent canopy graph cg_3, it is necessary to perform these operations for each set of inequalities that represent cg_1 and cg_2.

Heuristic Algorithm

The heuristic algorithm performs slack matching by starting at the lowest level of the hierarchy and then moving on to successive levels. At each level, it determines which of two branches needs additional stages and then adds stages somewhere along that path, possibly descending down the hierarchy. Because the algorithm uses no backtracking or exhaustive search, it is faster than the MILP solution and other past slack matching algorithms [47, 3].

Heuristic. In order to understand the heuristic for adding stages, note that the area under a canopy graph represents a set of conditions under which the circuit can operate. Any buffer

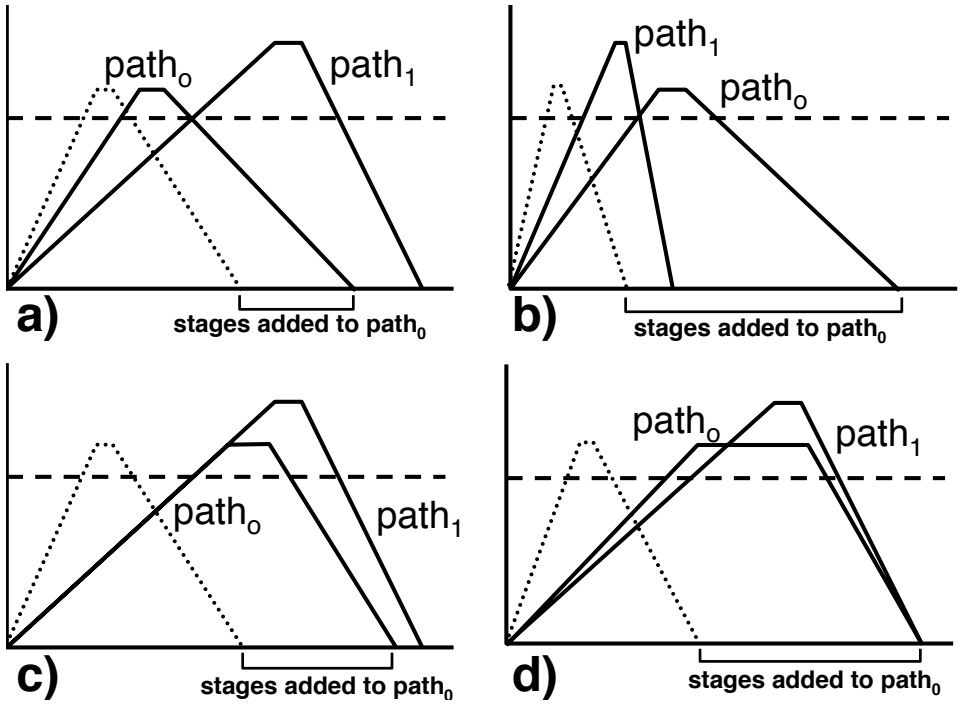


Figure 5.9: Key canopy graph intersections

stage addition that expands the area under the canopy graph—such that the resulting canopy graph is a superset of the previous canopy graph—will never be detrimental to the performance of the system. The heuristic for the placement of a buffer stage, therefore, is to compare choices for stage placement based on which will expand the canopy graph most.

Four types of intersection calculations are important in determining the number of stages to add and finding which stage placement will expand the canopy graph most. To describe these intersection scenarios throughout this section, the terms "forward" and "reverse" generically refer to the boundary lines of the canopy graphs in the data limited and hole limited regions respectively.

The first interesting canopy graph intersection point is at the minimum number of buffer stages needed to achieve a goal throughput for two paths, $path_0$ and $path_1$. Figure 5.9(a) shows how adding slack stages to $path_0$ creates an intersection between the two canopy graphs at the desired throughput level. To find this number of stages, use the reverse line of $path_0$, the forward line of $path_1$ and solve for the number of stages, s_0, to add to $path_0$.

At some point, however, placing additional stages on $path_0$ will cause the throughput to degrade. Figure 5.9(b) shows the point at which placing additional buffer stages will decrease throughput. Finding the number of stages is identical similar to the previous intersection point, except the forward line of $path_0$ and the reverse line of $path_1$ are used.

At some number of added slack stages, the *forward degradation* point, the slopes of the two forward lines will be identical, as illustrated in Figure 5.9(c). To find this number of stages, set the two forward lines equal to each other and solve for the number of stages to add to $path_0$. Note that adding further slack stages will cause the combined canopy graph to have a forward line with a shallower slope, thereby reducing the maximum throughput possible *at some occupancies.* This condition will lead us to avoid placing stages along this path, if possible, because it will cause the canopy graph to get smaller, which goes against the stage placement heuristic.

Finally, Figure 5.9(d) shows that adding stages can cause the effective occupancies of the two canopy graphs to be identical. At this point, the *reverse degradation* point, adding further stages does not increase the maximum occupancy of the resulting canopy graph. This condition will lead us to avoid placing stages along this path, if possible, because it will not expand the canopy graph.

Algorithm. The slack matching algorithm uses the canopy graph expansion heuristic to decide on slack stage placement. Pseudocode for this algorithm is shown in Figure 5.10. At each level the algorithm calculates and stores the minimum and maximum number of stages for one of the branches.

When placing stages at each level, it first descends down to the lowest level along that path. It will add stages at that level in order to slack match *unless* adding a stage will cause it to reach the maximum number of stages, the forward degradation point, or the reverse degradation point. It repeats this at each level, up to the current level, until the minimum number of stages has been added. [2]

[2] See Appendix A for a proof of existence of integer solutions for this method.

```
For each level of the hierarchy
  Calculate min and max for current level
  Descend to lowest level of hierarchy
  While min is not yet reached
    Add stages till reaching max or degradation points
    Move up to next level
  Store max stages and degrade point for current level
```

Figure 5.10: Slack matching algorithm

5.3.3 Loop pipelining: a novel transformation for bottleneck alleviation

Loop pipelining and loop unrolling are a TRICs that can be applied when algorithmic loops are the bottleneck for a system. I illustrate the challenge to be overcome in pipelining loops with the simple code example shown in Figure 5.11. This example represents a generic code fragment that iterates on every data set using a **for** loop. In the example, the symbols $s1$ through $s8$ represent statements.

A direct translation of this code into data-driven hardware typically yields the schematic structure shown in Figure 5.12. Each statement in the code—$s1$ through $s8$—becomes a pipeline stage. Statements $s3$ through $s6$ along with the **for** statement compose the *loop block*. During operation, the environment supplies data sets to the system via the read block and receives the outputs as they are completed via the write block.

A key observation is that the *loop block* has the same external interface as an individual pipeline stage, even though internally it contains an entire ring for iterative computation. Specifically, the *loop block* accepts one data set from the predecessor stage, performs a calculation, and passes the computed result on to a successor stage. It does *not* accept new data until the results of the calculation have been accepted by the successor stage.

Interestingly, the presence of a loop in the *compute* body has the same effect on performance as *a single pipeline stage with long latency*—it limits the throughput of the pipeline as a whole. Figure 5.12 illustrates this bottleneck scenario by indicating the presence of data with a dot. Stages downstream from the *loop block* are idle while the stages upstream from

```
func compute(input)
  s1; s2;
    for i = 1 to N
      s3; s4; s5; s6;
    end
  s7; s8;
  return(result)
```

Figure 5.11: Sample code for a *for* loop

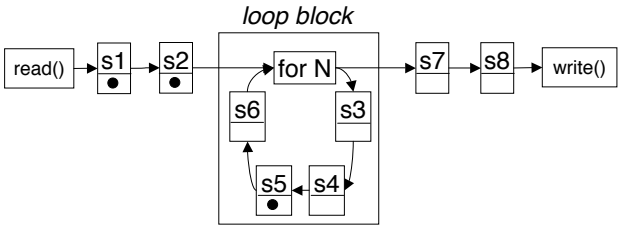


Figure 5.12: A simple implementation of the *compute* function

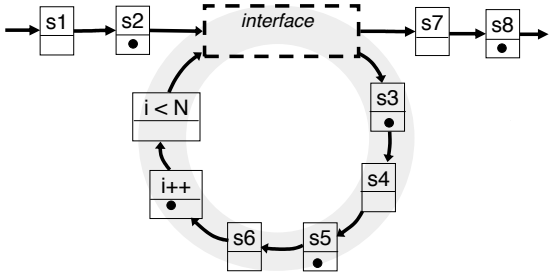


Figure 5.13: A loop pipelined implementation of the *compute* function

the *loop block* are stalled.

Loop Pipelining Approach

The key difference between my approach and previous work is to allow multiple data sets into the ring concurrently. Allowing multiple data sets to enter reduces pipeline stalls in the stages before the loop and reduces starvation in the stages after the loop. The net effect is increased throughput for the system as a whole. Figure 5.13 shows the basic concept behind transforming the code of Figure 5.11 into a pipelined loop that can hold multiple data sets. In this example, the loop body holds three data sets, thereby reducing stalling before the loop and starvation after the loop.

Allowing multiple data sets within the ring concurrently presents three challenges. i) Flow control - managing the entry and exit of data sets in the ring and preventing collisions; ii) congestion prevention - preventing the ring from becoming over full, which leads to slowing down or stalling; iii) Data hazard prevention - allowing concurrency while preventing separate data sets from overwriting each other's values.

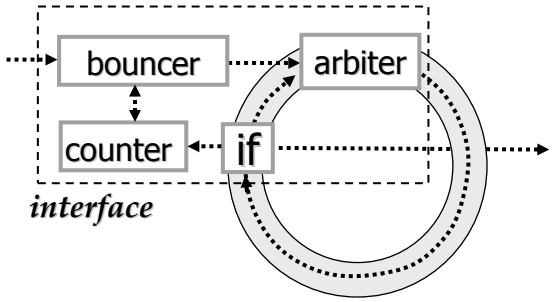


Figure 5.14: My method of loop flow control.

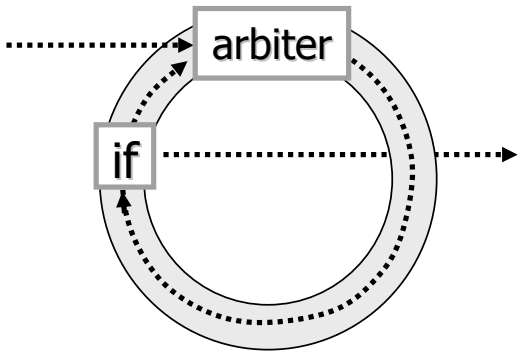


Figure 5.15: Congestion prevention for pipelined loops.

Flow Control. I resolve challenge of flow control by adding two stages to the interface of the loop. A conditional split (*i.e.* a pipelined if construct) and an arbitration stage. The arbiter prevents newly entering data from colliding with data already cycling in the ring. The **if** checks the boolean condition and sends the data out of the ring or back in through the arbiter. Together, these two stages of the ring interface manage the entry and exit of data sets and prevent collisions. Figure 5.14 shows the use of these two stages.

Ring Congestion. Addressing the challenge of ring congestion requires some information about the performance of the ring. In particular, as described in Section 2.2.2, every ring has some ideal occupancy (*i.e.* number of data sets) at which it attains maximum throughput. This ideal occupancy can be found through timing analysis of the ring or through simulation.

Two stages in the loop interface are used to enforce the ideal occupancy : the bouncer and the counter. The counter dynamically keeps track of the number of data sets within the ring; it is notified whenever a dataset leaves and whenever one enters. The bouncer uses the information from the counter to determine if the ring is congested. It only allows a new data set to enter when the ring is not congested. Figure 5.15 shows these two stages added to the interface with arrows to indicate communication between them.

Data Hazards. In order to prevent data sets from interfering with each other (*i.e.* data hazards) I replicate some of the variables to ensure sufficient storage when operating on many concurrent data sets. In particular, each stage must store its own copy of the *context*—the

```
func compute (b,c)
var a, i;
 for i = 1 to N
   a = b*2;
   b = a+c;
 end
  return t6
```

Figure 5.16: Sample code with feedback

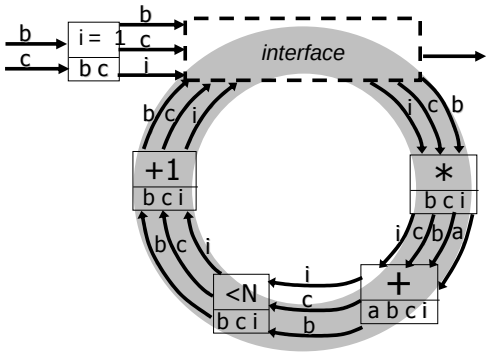


Figure 5.17: Preventing data hazards in pipelined loops.

collection of all variables needed by that stage or one of its successor stages. For example, the code in Figure 5.16 contains a loop that uses four different pieces of data: a, b, c, and i. Figure 5.17 shows a loop pipelined implementation of this code. Notice that since the counting variable, i, can have a different value for each data set, the value is stored in every stage to prevent collisions. Notice also that the value a is not stored in every stage; it is produced in stage 2, consumed in stage 3, and not in any subsequent stages.

More formally, I solve the problem of data hazards by storing the context for each data set locally in each stage's storage element, and not in any centralized memory location. I determine which variables need to be latched in each stage using *live variable data flow analysis*—a method for determining which values may be used later and which can be discarded. In particular, the IN set of each stage is the set of data values that are required to go into a code fragment either because they may be used inside, or because they may need to be relayed to a successor of that fragment. Each stage must store its IN set, so all variables which are live at that stage have a storage space.

Loop Unrolling

Unrolling the loop body to form a ring with a greater number of stages can greatly improve performance. Intuitively, this improvement results from the duplication of hardware inside the loop and a corresponding increase in the ring occupancy. Thus, the unrolled loop is able to

perform more "work" per unit time. More formally, the ring frequency (at ideal occupancy, *cf.* Equation 2.7) remains fairly unchanged when the loop is unrolled, but every "tick" at the loop's interface now represents a greater amount of work completed. In particular, if the loop is unrolled u times, every time a data set crosses the interface stage, it indicates that u iterations have just been completed on that data set, rather than just one. Therefore, ignoring overheads, the loop's effective computation throughput increases by a factor equal to the number of times it is unrolled, u.

As a second-order effect, loop unrolling actually also has the benefit of somewhat reducing the overhead of the special-purpose "helper" stages: the bouncer, the counter and the arbiter. That overhead is now amortized over a larger ring. As a result, the latency of each data set will tend to somewhat decrease and hardware utilization will slightly increase. One possible negative effect of loop unrolling is that it can cause some data sets to be iterated over more times than necessary, thereby requiring extra checks within the unrolled loop to preserve the semantics of the computation.

Although loop unrolling is a common technique in both software and hardware optimization, goals and performance effects are generally different. In software compilers, the primary benefit of loop unrolling is to introduce more room to allow instructions to be reordered, with the purpose of reducing stalls due to branch and data hazards. In hardware translation approaches, such as [8, 63], loop unrolling is used in conjunction with compaction to increase concurrency within the loop body. However, these approaches do not allow an increase in the occupancy of the loop, thereby obtaining limited throughput improvement. In contrast, unrolling pipelined loops increases the loop occupancy by the same factor as the number of times it is unrolled, thereby obtaining dramatically higher speedup.

Performance Benefit and Overheads

Previous work on performance analysis of rings allows us to predict the speedup obtained by the use of our method. As discussed in Section 2.2.2 and shown in Figure 2.6, the *ring*

frequency is proportional to the total number of data sets that are revolving inside the ring, as long as the ring is not congested (*i.e.,* "hole limited"). Therefore, the maximum speedup of our approach is proportional to the ideal ring occupancy.

The speed improvement of our method is largely due to improved hardware utilization. A ring that holds only one data set has a high amount of unused hardware at any given time. By allowing multiple data sets, we are able to obtain high hardware utilization from the components within the ring.

Our approach adds some overhead that decreases the actual speedup and increases total area. Certain "helper" stages—such as the bouncer, counter and arbiter—increase the latency of the ring and add a constant area overhead for each loop. Also, any counter implementation must have some latency and therefore will not allow new data to enter immediately after old data leaves. The effect is that the ring may be operating at an occupancy that is, on average, slightly lower than the nominal occupancy. This results in a decrease in throughput only if this slightly lower occupancy has a throughput lower than the maximum. The most notable increase to area is the extra storage elements that are required in order to hold the entire context at each ring stage. This overhead is necessary to allow each data set to hold its own copy of the loop's context, thereby enabling multiple data sets to coexist independently within the ring.

5.4 Bottleneck Identification

Finding the causes for a system's throughput limits is challenging because the throughput is determined by complex interactions between its constituent stages. Using canopy graphs as the basis for analysis can help expose the full range of causes for a bottleneck. The pipeline identification algorithm takes the following inputs: a system's hierarchical description, the performance metrics associated with individual stages (*i.e.*, forward latency, reverse latency, cycle time, and capacity; see Section 3.2), and probability estimates for any choices within the

system. It gives as output an expression that represents the parts of the system that limit the throughput. More specifically, the output is an AND-OR expression of hierarchical nodes that indicates, using AND-OR causality, which parts of the circuit work together to limit throughput.

The bottleneck detection method consists of three basic steps. The first step is to determine the canopy graph for the entire system, hierarchically from the bottom up, by applying the analysis method of Chapter 4. Next, for each node in the tree that represents the system's hierarchy, the algorithm determines which of its child nodes is responsible for limiting its canopy graph. Finally, the algorithm reports the set of nodes responsible for limiting the overall system's throughput.

5.4.1 Classification of Bottlenecks

To aid in alleviating bottlenecks, this section introduces categories based on the type of underlying problem within the system that needs to be solved.

Type I: Forward Latency Dependent Bottlenecks. Latency dependent bottlenecks are caused by a part of the system having a forward latency that acts as a bottleneck. If a forward segment has been indicated as a limiting segment, this implies that a Type 1 bottleneck exists at that system node.

Type II: Cycle Time Dependent Bottlenecks. Cycle time dependent bottlenecks occur when the cycle time of one part of the system limits throughput. In terms of canopy graphs, if a top segment is indicated by the bottleneck identification method, this implies that a Type 2 bottleneck exists at that system node.

Type III: Occupancy Dependent Bottlenecks. Occupancy dependent bottlenecks are caused by part of the system having insufficient buffering or a high reverse latency. If a reverse segment is indicated as a limiting segment, this implies that a Type 3 bottleneck exists at that system node.

5.4.2 Step 1: Compute Overall Canopy Graph

The first step in finding the bottlenecks is to find the canopy graph of the entire system by applying the analysis method presented in Chapter 4. The analysis begins at the leaf nodes of the tree and progresses upwards. Specifically, the canopy graph for a leaf node is determined by the forward and reverse latencies of the individual stage at that node. Subsequently, the canopy graph at each non-leaf node is found by composing the canopy graphs of its child nodes. During analysis, the algorithm stores in each tree node the canopy graph computed at that node; this information will be used in the next step of the algorithm.

5.4.3 Step 2: Find Limiting Segments

Once the canopy graphs have been computed for each node in the system's hierarchical tree representation, the next step is to determine which stages are responsible for the limiting lines of the canopy graphs at each node. When more than one child limits its parents canopy graph, the relationship between contributing children is one of two possibilities: 1) AND-causality: the child is part of a group of stages which jointly limit the parent's canopy and must all be structurally modified to alleviate the bottleneck; or 2) OR-causality: though several children contribute to limiting the parent's canopy, changing any one of them will alleviate the bottleneck. This method is applied to all the nodes in the system's tree representation. The result is an AND-OR tree that identifies the bottlenecks present throughout the system. This method is now described in detail.

Given a node in the hierarchy tree, the method for computing which of its children is causing the parent's canopy to be limited depends on the type of composition represented by the parent node. Each of the four cases is now discussed.

Parallel Nodes

The forward segment of a canopy graph at a parallel node is limited by the child canopy graph in which the forward segment has the most shallow slope. If the forward segments of the two children have the same slope, both segments are added to the AND-OR tree using an AND operator, which indicates AND-causality, *i.e.*, both children must be structurally modified in order to change the forward segment of their parent's canopy. If only one child limits the forward segment of the parent's canopy, then that child alone is added to the AND-OR tree (using a degenerate single-input AND operator).

Similarly, each reverse segment of a canopy graph is limited by one or both of its children. If it is limited by both, the appropriate reverse segments of both are added to the AND-OR tree using an AND operator. If only one child limits the reverse segment of the parent's canopy, then only that child is added to the AND-OR tree.

The top segment of the canopy graph at a parallel node is limited in one of two possible ways. First, if any of its children have a top segment with the same limiting throughput value, then that node is added (or those nodes are added) to the AND-OR tree using an AND operator.

Second, the throughput may instead be limited by a slack mismatch. Specifically, the forward segment of one child intersects one of the reverse segments of the other child, thereby limiting throughput of the composition, even though each child individually could have supported higher throughput. In this case, the offending reverse segment of one child and the offending forward segment of the other child are both added to the AND-OR tree using an OR operator. This case is one of OR-causality because, as shown later in Section 5.5, the bottleneck can be removed by modifying either of the two children.

Conditional Nodes

The method for identifying bottlenecks at a conditional node is similar to the method for identifying bottlenecks at a parallel node, for all segment types. Specifically, first the canopy of each child is scaled by dividing it by the probability of that branch, as explained earlier in

Section 4.3.3. Then, the conditional is treated as a parallel node, and the method described above applied to all segments of the conditional's canopy graph.

Sequential Nodes

The forward segment of a canopy graph at a sequential node is affected by the forward segment of every child, so the forward segment of every child of the sequential node is added to the AND-OR tree with an OR operator.

Similarly, each reverse segment of the canopy graph is affected by one reverse segment of every child. Specifically, the reverse segment of the node spans some range of throughputs; any child reverse segment that also falls within this range affects that segment of the parent's canopy. Each of these reverse segments is added to the AND-OR tree with an OR operator.

The top segment of sequential node is limited by the top segment of that child that has the lowest throughput among all children. If more than one of the child nodes have top segments at the same throughput, each of them is added to the AND-OR tree with an AND operator.

Loop Nodes

As described in Section 4.3.4, the canopy graph of a loop node is simply the canopy graph of the loop's body scaled down vertically by the expected iteration count. Thus, the forward, top and reverse segments of the canopy of the loop node are limited, respectively, by the forward, top and reverse segments of its child's canopy graph. Hence, the child's (*i.e.*, the loop body's) segments are added to AND-OR tree (using a degenerate AND operator).

5.4.4 Step 3: Compute And-Or Bottleneck Formula

The AND-OR tree for the limiting segments represents the segments in the entire system that affect the overall throughput. Figure 15 depicts an AND-OR tree for the simple example of Figure 12. The nodes are annotated with the type of limiting segment and the operator of the and-or tree. At each node in the tree, the responsibility can be placed upon the current node

OR it can be passed on to the child nodes of the AND-OR tree. The final expression for this system's bottlenecks can be found by recursively combining the expressions for each node.

Figure 5.18 shows pseudocode for the algorithm that determines this final expression. At each node, the expression for its child nodes are found and combined with the operator at that node. The expression for the node itself is then added. Using the example of Figure 5.19, at node n_0 first the expressions for n_1 and n_2 are found and then the top segment of n_0 is appended to the expression with the OR operator. The final expression for the bottlenecks in the system is $top_{n_0} + (top_{n_1} \cdot (top_{n_2} + (top_{n_3} \cdot top_{n_4})))$.

```
Expression limitingSegments( node, segment )
    for each child of node
        expression += limitingSegments(child, childsegment)
        expression += operator
    expression = segment "OR" expression
    return expression
```

Figure 5.18: Pseudocode for generating limiting segment expression

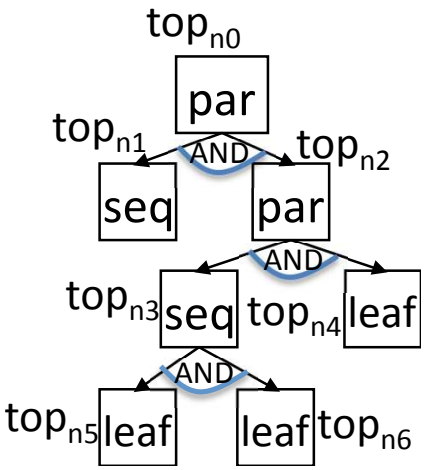


Figure 5.19: AND-OR tree for the system of Figure 5.22

5.5 Bottleneck Removal

5.5.1 User-Guided Method for Bottleneck Alleviation

Determining which transform will alleviate a bottleneck in the system is non-trivial, because each TRIC applies to only some types of bottlenecks and some types of nodes. Based on the AND-OR tree of bottlenecks, as described in Section 5.4, our system reports a sum-of-products form of possible TRICs. It then relies on the user's knowledge of the design domain to choose the TRIC that is most compatible with their needs (*e.g.* a designer might avoid excessive duplication if area is a concern.)

Determining TRIC Applicability. Table I summarizes the rules that our tool uses when determining which TRICs to suggest to the user. In the table, a check mark indicates that the TRIC is a good candidate for removing that type of bottleneck, an *X* indicates that it will exacerbate a given type of bottleneck, and a dash indicates that the TRIC will make little to no change in that type of bottleneck.

	Type 1	Type 2	Type 3
Coalescing	√	X	X
Parallelization	√	-	X
Stage Splitting	X	√	√
Loop Pipelining	X	√	√
Duplication	-	√	√
Loop Unrolling	-	√	√
Buffer Insertion	X	-	√

Table 5.1: TRICs applicability to the bottleneck types

Iterative Algorithm for Bottleneck Alleviation. The use of our tool for iterative bottleneck elimination is described by the very high-level algorithm of Figure 5.20. First, the user picks some goal throughput for the system. Next, the algorithm given in Section 5.4 is used to identify the bottlenecks. Then choices of TRICs and the nodes on which to apply them are

listed for the user. After applying one or more TRIC, the target throughput might not yet have been met. Often, removing one bottleneck reveals another one, so the method must be repeated until the goal throughput is met or surpassed.

```
while( throughput < goal )
    Find bottlenecks
    List possible fixes
    Apply user design choice
```

Figure 5.20: Iterative algorithm for bottleneck alleviation

5.5.2 Automated Bottleneck Removal

Problem Domain

In order to find an optimized circuit, the search method needs the following inputs: a hierarchical representation of a circuit, a goal throughput, a set of TRICs, and a cost function. The desired output is a sequence of TRICs that, when applied to the circuit in the given order, yield a new circuit that meets or exceeds the throughput goal while minimizing the given cost function.

Input Circuit The bottleneck alleviation method presented in this Chapter requires a hierarchical representation of the circuit to be optimized. The hierarchical representation consists of individual pipeline stages—each of which has a forward latency, a reverse latency, a cycle time, and a capacity—and operators that combine those stages. The operators supported are sequential, parallel, conditional, and iterative composition operators. The supported hierarchical constructs are described in Section 3.3. Figure 5.21 depicts a block diagram of a circuit to be analyzed and optimized. Each block represents a single pipeline stage. The hierarchical regions are marked in order to highlight the nested, hierarchical structure of this circuit.

Within the framework, a tree represents each hierarchical circuit; the leaf nodes represent individual stages and all other nodes are either parallel, sequential, conditional, or loop oper-

ators. Figure 5.22 depicts the tree structure of a hierarchical system. Parallel and conditional nodes have exactly two children, sequential nodes have two or more children, and loop nodes have one child.

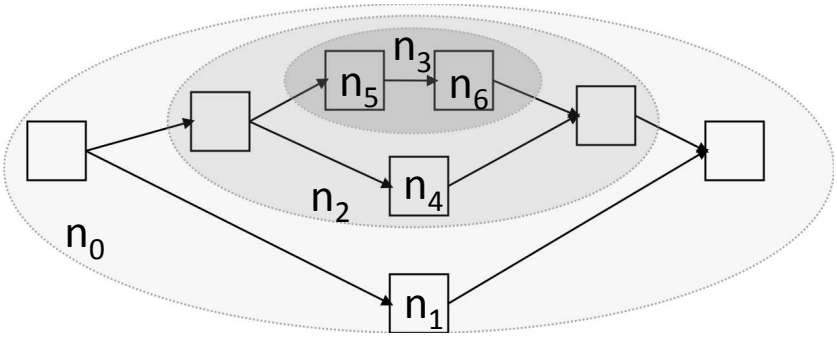


Figure 5.21: Block level representation of a hierarchical system

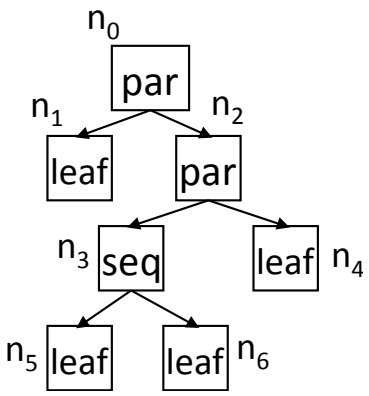


Figure 5.22: A tree representing system of Figure 5.21

Throughput Goal The framework requires a throughput goal for the circuit. The throughput of the system as a whole will be computed using the method described in Chapter 4, and then compared with the goal throughput.

Circuit Transformations (TRICs) In order to make changes to the given circuit, the framework needs access to a set of TRICs (TRansformations for Increasing the Canopy graph), such as the one presented in Section 5.3. For use in the solution framework, a TRIC must specify both how it changes the circuit tree and the criteria for its applicability.

TRIC Applicability to a Bottleneck Type Each TRIC can alleviate some set of bottleneck types. Section 5.4 gives three different categories of bottlenecks that can occur; these are given the labels Type I, Type II, and Type III. Specifically, if a TRIC increases the area under the forward segment of the canopy graph, that TRIC could potentially alleviate a Type I bottleneck. Similarly, TRICs that affect the top and reverse segments of the canopy graph can be applied to Type II and Type III bottlenecks respectively. Note that the framework itself does not determine which bottleneck type each TRIC applies to, but instead requires that this information be part of the TRIC specification.

TRIC Circuit Changes Each TRIC must specify which set of circuit node types it can be applied to. For example, the coalescing TRIC of Section 5.3, which combines the functionality of two stages into one stage, can be applied only to two leaf nodes that are composed sequentially.

Each TRIC specifies its changes to the circuit in terms of changes to the circuit tree, and the resulting circuit must always be representable as a hierarchical circuit tree. Examples of possible changes that a TRIC can make to the tree include removing nodes, replacing nodes with nodes of a different type, and adding nodes to a sequential construct. TRICs are not allowed to generate any nodes with more than one parent, thereby preserving the tree structure of the system specification.

Cost Functions Often, there are many different ways to modify a circuit to reach a throughput goal. Therefore, the solution framework requires a cost function that includes some metric other than throughput. Common cost metrics include the energy area product and energy times time squared, or $E\tau^2$ [32]. For a cost function to be compatible with the solution framework, it must be able to take the tree representing the circuit as input and output the total cost of the circuit. In addition, the cost function must be meaningful when applied to any node in the circuit tree. For example, applying the cost function to node n_3 in Figure 5.22 should give the cost of the sub-circuit including nodes n_3, n_5, and n_6.

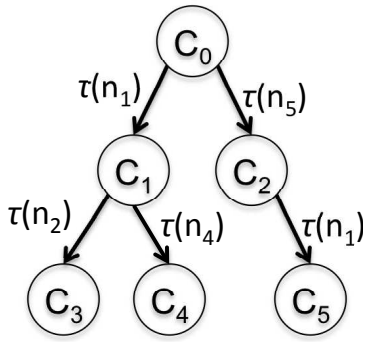


Figure 5.23: A sample search tree

Output The framework outputs the set of TRICS to apply; this includes information about which nodes each TRIC applies to and the order in which to apply the TRICS. It also outputs the cost of the circuit for the given cost function, and the throughput prediction for the circuit.

Solution Space

Search Tree Structure The search space can be represented by a tree structure in which each node corresponds to a circuit with the same behavior as the original input circuit. As an example, Figure 5.23 shows an example search tree for the circuit in Figure 5.21. The root vertex of the tree, C_0 represents the original circuit specification input to the system. Each of its children represent the result of applying a TRIC to that original specification. Specifically, for every vertex C_i there is some set of transformations, $TRIC_i$, that potentially alleviate a bottleneck in the circuit.

An edge from vertex C_i to vertex C_j exists if there is some transformation in the set $TRIC_i$ that changes the circuit corresponding to C_i to the circuit corresponding to C_j. The notation $C_i \xrightarrow{\tau} C_j$ denotes that some TRIC, τ, when applied to the circuit represented at node C_i will yield the circuit at C_j. The weight of the edge is determined from the given cost function. Specifically, the weight of an edge from C_i to C_j is the change in the cost function between the two circuits.

A path through the tree is a series of TRICs starting at the root vertex C_0 and ending at some vertex C_i. The notation $C_0 \xrightarrow{p_i} C_i$ indicates that some path p_i exists from node C_0 to

node C_i. The tree consists of a set of vertices such that for every vertex there is some path from the root node to to that vertex. There is an edge from one vertex to another if there is a TRIC that could be applied to a bottleneck in the first circuit to produce the second. More formally, the set of vertices, V, and edges, E, can be defined as follows:

$$\begin{aligned} V &= \{C_i \mid \exists p_i C_0 \xrightarrow{p_i} C_i\} \\ E &= \{(C_i, C_j) \mid \exists \tau C_i \xrightarrow{\tau} C_j s.t. \tau \in TRIC_i\} \end{aligned}$$

Terminating Conditions Edges in the graph can have either positive or negative weights, depending on the specific TRIC applied and the cost function being used. As a result, the search space is non-monotonic, which makes it difficult to find heuristics for pruning the search tree. Further, the search space can be infinite, because there can be arbitrarily long sequences of TRICs to consider in the absence of methods to effectively bound the search space.

The solution framework currently uses two terminating search conditions: reaching some maximum depth and finding a *prime solution.* Specifically, the setting the maximum depth places the restriction that path length from the root vertex C_0 to any graph vertex be less than d, that is $||p|| < d$.

The second possible terminating condition is the *prime* solution. Intuitively, a prime solution is the first solution on a given path that reaches the given throughput goal, g. This limits the vertex set V by placing a restriction on path p that no other solution path exists that is a subset of p. More formally, C_i is a vertex in the search space iff $\forall p_x | C_0 \xrightarrow{p_x} C_x \xrightarrow{p'} C_i, tpt(C_x) < g$. That is, C_i is a reachable vertex in the search space only if all the vertices that lie along the path from C_0 to C_i *do not* meet the goal. Note that since the search space is non-monotonic, stopping the search at the prime solution can potentially prevent the search from finding a lower cost solution further past a prime solution. The trade-off, which is worth this potential loss in solution quality, is that using the prime solution prunes the search space to a more

manageable, non-infinite size.

Solution Methods

Exhaustive Search An exhaustive search of the solution space is quite time consuming. In an exhaustive search, every possible path is explored until reaching the terminating condition. Exhaustive search uses both solution depth and finding the prime solution as terminating conditions, thereby pruning the search tree at some maximum depth and at every prime solution. Figure 5.24 is a graphical representation of this search. A search along a given branch terminates at the prime solutions, shown as shaded circles.

Pseudocode for a recursive implementation of this algorithm is shown in Figure 5.25. The exhaustive search method finds all the solutions within the search tree, and then the lowest cost solution that meets the throughput goal is reported as the final solution.

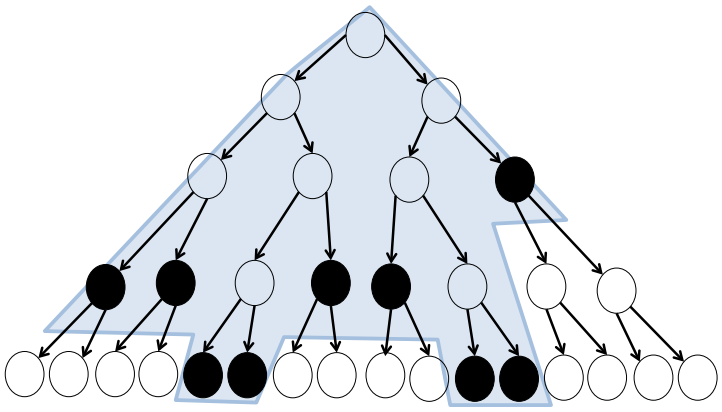


Figure 5.24: Exhaustive search with prime solutions as the terminating condition

Greedy Search A greedy search of the solution space prioritizes the vertices of the search tree based on which of the next vertices has the lowest heuristic function. If the node that has the lowest heuristic function is a solution, (*i.e.* the throughput is greater than the goal throughout) the search ends and the lowest cost solution found is returned. Figure 5.26 is a graphical representation of a greedy search. Lightly shaded nodes are analyzed but not taken

```
SolutionPath ExhaustiveSearch( TreeNode t, double goal )
    for each t.getChild
      if child.tpt >= goal
            if lowCost > child.tpt
                  lowCostNode = child
      else
            ExhaustiveSearch( child, goal )
    return lowCostNode.getPath
```

Figure 5.25: Code description of exhaustive search

while dark nodes are the nodes along the chosen path. The final node is the prime solution along that path.

Pseudocode for a the greedy algorithm is shown in Figure 5.27. Greedy search can be faster than an exhaustive search because fewer vertices are visited. However, the solution obtained may not be the lowest cost solution. The greedy search is also bounded by prime solutions: it continues until it finds the first solution along a path.

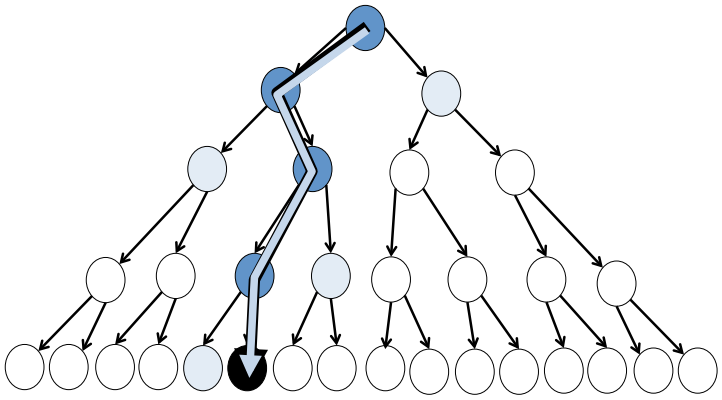


Figure 5.26: Greedy search evaluates a small number of nodes

Lookahead Search A lookahead search prioritizes vertices based on cost information from nodes some depth, d, away from each given vertex. Greedy search can be called a special case of lookahead search: lookahead 0.

Figure 5.28 is a graphical representation of three steps in a lookahead-2 search. The dark shaded nodes represent nodes with the minimum value for the heuristic function at a given

```
SolutionPath greedy( TreeNode t, double goal )
 for each t.getChild
       if child.heuristic < lowHeuristic
            lowHeuristicNode = child
       if child.tpt >= goal AND child.cost < lowCost
            lowCostNode = child

 if lowHeuristicNode.tpt >= goal
      return lowCostNode.SolutionPath
 else
      return greedy( lowHeuristNode, goal )
```

Figure 5.27: Code description of greedy search

depth. The algorithm takes one step in the direction of the shaded node, and evaluates another set of nodes in that direction. Pseudocode for a recursive implementation of this algorithm is found in Figure 5.29. During the search, algorithm keeps track of the minimum cost solution that it has found at any point in the search space. When the it finally terminates, it will return this minimum cost solution, which may be different from the minimum heuristic solution.

A lookahead search can produce a more optimal solution than a greedy search while still visiting a smaller number of nodes then an exhaustive search. Therefore, it can offer a speed advantage over exhaustive search while still returning a result that is close to optimal.

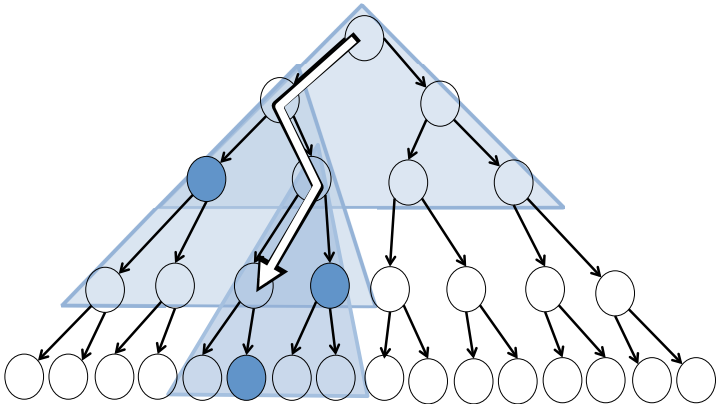


Figure 5.28: Three steps in a lookahead-2 search

```
SolutionPath LookAhead( TreeNode t, double goal, double depth )
  for each t.getChild
      expand Child lookDistance nodes ahead
      record minCostSolution
      record minHeuristicNode

  if( minHeuristicNode.tpt >= goal )
        return minCostSolution
  else
        return Lookahead( minHeuristicNode, goal, depth)
```

Figure 5.29: Code description of lookahead search

5.5.3 Advanced Tree Pruning Approaches

This section details two further enhancements to the techniques for searching the design space. The first involves recognizing which sets of TRICS are commutative (*i.e.* the order of the operations does not matter.) The second is identifying when one TRIC effectively undoes the action of a previously applied TRIC; these TRICs are *reciprocals*. Both enable additional pruning of the search space to eliminate redundant search paths, which leads to faster optimization runtimes.

Commutative Transformations

Definition of Commutativity for TRICS Two TRICS are commutative if the order in which they are applied does not affect either the structure or the performance of the resulting circuit. Consider two TRICS τ_a and τ_b which transform graph vertices such that $C_0 \overset{\tau_a\tau_b}{\rightarrow} C_x$ and $C_0 \overset{\tau_b\tau_a}{\rightarrow} C_y$. If C_x and C_y are equivalent for all possible initial nodes C_0, then the two TRICS are commutative.

Commutativity of the Bag of TRICS The TRICS currently implemented in this tool are coalescing, stage splitting, duplication, buffer insertion, and loop unrolling. When determining if the TRICS are commutative, it is necessary to consider the set of circuit hierarchy nodes that is affected by the TRIC. TRICS that affect individual leaf nodes in the circuit hierarchy

will be commutative, since their effects on the circuit are strictly local.

Loop unrolling and duplication, however, have more global affects and are potentially not commutative. Both create copies of some set of circuit nodes, thereby resulting in a change to a large part of the circuit and limiting commutativity of subsequent TRICS. For loop unrolling, I address this issue by marking the circuit nodes that have been affected by the loop unrolling. When subsequent transformations are applied, the marking enables the framework to judge that the transformation taking place after the loop unrolling is distinct from one taking place before the loop unrolling.

In duplication, I address this issue by modifying the set of TRICS that can be applied after duplication in order to restore the commutativity of the operation. In particular, once a portion of the circuit has been duplicated, all TRICS that are applied to one duplicated circuit component must also be applied to its matching duplicated component. This is not likely to result in loss of solution quality because applying a TRIC to only one half of a duplicated node will not improve overall throughput.

This conclusion is reached based on knowledge of the bottleneck identification method used by the framework, as presented in Section 5.4. Specifically, after duplication takes place, the canopy graphs (*i.e.*the throughput plotted vs. the occupancy) of the two duplicated paths will henceforth be combined using the parallel operator. The parallel operator prescribes that the canopy graphs of the two components are intersected with each other, as described in Section 4.3.1.

If two different TRICS, τ_a and τ_b are applied to the duplicated nodes respectively, then the new canopy graph becomes $\mathbb{C}_a \cap \mathbb{C}_b$. On the other hand, if the same TRIC, τ_a, is applied to both of the duplicated components, the resulting combined canopy graph is simply $\mathbb{C}_a$; similarly if the same TRIC, τ_b, is applied to both of the duplicated components, the resulting canopy graph is equal to $\mathbb{C}_b$. Based on the definition of intersection, it is impossible that $\mathbb{C}_a \cap \mathbb{C}_b$ contains any points that are not contained in both $\mathbb{C}_a$ and $\mathbb{C}_b$. Therefore, applying either τ_a or τ_b to both nodes will result in a canopy graph that is a superset of the one obtained

by applying τ_a to one of the duplicated components and τ_b to the other.

For this reason, the commutativity of duplication with other operations is preserved by enforcing that TRICS applied to one duplicated component must always be applied in kind to the other. Preserving commutativity allows the search space to be more effectively pruned and therefore further reduces the execution time of the algorithm.

Exploiting Commutativity for Runtime Improvement Figure 5.30 shows a search tree composed of the permutations of three TRICS: τ_a, τ_b, and τ_c. If the three operations are commutative, the grayed nodes are removed from the search, assuming a left-to-right traversal order for the tree.

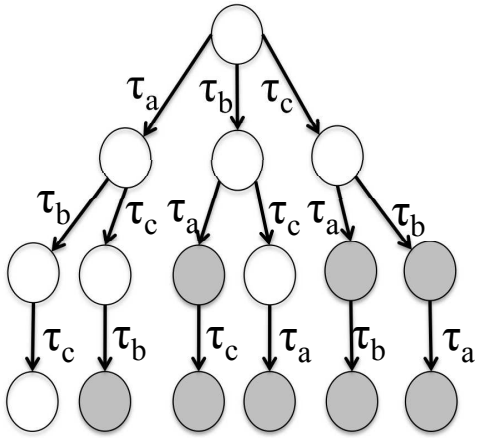


Figure 5.30: Recognizing commutative operations to prune the search space

To implement this pruning, a canonical representation is created for each path already explored. The canonical representation is a lexicographical reordering of the search path such that two paths that are equivalent under commutativity will also have the same canonical paths. Pseudocode for the pruning algorithm using canonical paths representations to prune the search space is shown in Figure 5.31. After the tool determines a set of possible next TRICS, it creates a canonical representation for each one and adds the TRIC to the queue of TRICS to be applied only if the canonical representation is not in the set of paths already explored.

```
CanonicalTest( CompositionNode node )

  TRIC[] choices = bottleneckAnalaysis( node )
  for each choice in choices
    path += choice
     canonical_path = lexicographical_order( path )
     if( !previous_paths.contains ( canonical_path ) )
          previous_paths.add( canonical_path )
          TRICqueue.add( choice )
```

Figure 5.31: Exploiting commutativity to prune the tree

Reciprocal Transformations

Definition of Reciprocal for TRICS Applying one TRIC can potentially be the reciprocal of another; applying it will undo all the changes made by an earlier TRIC such that the resulting circuit has the same structure and performance characteristics as the original circuit. More formally, consider two TRICS τ_a and τ_b which transform vertices such that $C_0 \stackrel{\tau_a \tau_b}{\longrightarrow} C_x$. If C_x always equals C_0 for all possible initial vertices C_0, then τ_b is the reciprocal of τ_a.

TRICS as Reciprocal Transformations Of the TRICS currently used, there are two that are reciprocals of each other. Specifically, coalescing and stage splitting, when applied successively to the same pipeline stage, will net no result change to the circuit. Although this set of TRICS does not include any other reciprocal transforms, introducing new TRICS (*e.g.* transformations to increase or decrease the level of parallelism) will likely create new pairs of reciprocal transformations that, when taken together, reduce to an identity transformation (i.e., cancel each other out).

Avoiding Identity Transforms to Improve Runtime Figure 5.32 shows a search tree pruned to avoid identity transforms. In this example, τ_a and τ_b together form an identity transform. As a result, the tree vertex labeled as C_x is equivalent to the vertex labeled at C_0, and the shaded vertices will not be visited during a search. Notice that the path consisting of the series of TRICS τ_a, τ_b, τ_c is no longer explored. This will not result in the loss of a

possible solution, however, because the path consisting of only τ_c is equivalent.

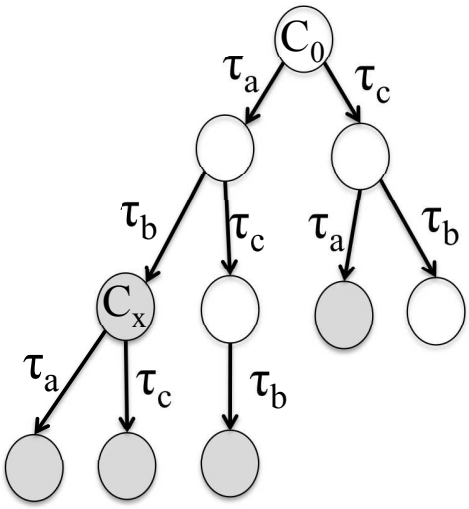


Figure 5.32: Pruning search space to avoid identity transforms

This optimization flags nodes within the hierarchical circuit that have been coalesced or split by past TRICs. These flags are then checked when determining the set of possible TRICs at a search node vertex. For example, if the bottleneck identification method lists coalescing among the possible optimizing transformations on a circuit node that has already been split, when that suggestion is checked against the flags it will be removed and not explored during the search.

5.6 Results

This section presents the results of applying the optimization methods described in this chapter. First Section 5.6.1 introduces the example circuits that are used to obtain these results. Sections 5.6.2 and 5.6.3 describe the benefits of using the loop pipelining and buffer insertion TRICs. Section 5.6.4 shows the use of bottleneck detection. Finally, Sections 5.6.5 and 5.6.6 respectively show the results for manual and automated application of optimizing transforms.

5.6.1 Benchmarks

The following is a description of all the benchmarks used for the results in this section:

- BISECT: An iterative implementation of a bisection algorithm for finding the zero of a polynomial that uses the POLY algorithm as a nested, internal loop.
- BTREE: The BTREE algorithm searches a binary tree residing in ROM for a given input key. A key match returns the data value associated with the node that matches the key. Behavior is highly data-dependent: the number of loop iterations depends on how deep the input key's node is in the tree.
- CORDIC: COordinate Rotation DIgital Computer is a a rotation algorithm for computing trigonometric functions.
- CRC: The cyclic redundancy check (CRC) algorithm calculates an 8-bit checksum for a given block of input data. This algorithm has a fixed iteration count (16 iterations).
- GCD: An implementation of Euclid's iterative algorithm for computing the GCD of two integers.
- JPEG: A JPEG encoder that takes an 8x8 image and uses two nested loops to iterate over the entire image and output a stream of encoded data. This particular implementation comes from the work of Hansen [21].
- MULT: A multiplier that includes a conditional because it performs a shift instead of a full multiply if the multiplicand is a power of 2.
- ODE: The ODE example is the ordinary differential equation solver from [73, 74, 63], based on the Euler method. It receives as input the coefficients of a third degree ordinary differential equation, along with an interval over which to integrate, initial conditions, and a step size. Its output is the final value of the dependent variable. The number of loop iterations depends on the size of the interval and the step size.

- POLY: An iterative polynomial evaluation method based on Cramer's rule.
- Raytracing: a pipelined implementation of a ray-sphere intersection algorithm—performs an iterative (*i.e.*loop pipelined) square root based on some data-dependent condition.
- TEA: Tiny Encryption Algorithm (TEA) encrypts a 64-bit block with a 128-bit key. The number of loop iterations depends on the number of encryption rounds chosen. For this example, the number of rounds was fixed at 32 (16 loop iterations).

5.6.2 Results for Loop Pipelining and Unrolling

Experimental Setup. Since loop pipelining targets systems with iterative constructs, only those benchmarks that include iteration are considered in these results. In particular, this section presents results for the examples BISECT, BTREE, CRC, GCD, ODE, POLY, and TEA. The benchmark examples were synthesized using the Haste/TiDE synthesis flow from Handshake Solutions [45], a Philips subsidiary. Haste (formerly "Tangram") is the only mature automated asynchronous synthesis flow available at present. While the control-dominated architectures that Haste currently produces are not an ideal match for pipelined dataflow applications, this experimental setup allows the relative performance benefit of the loop pipelining approach to be accurately estimated.

For each of the benchmark examples, three different implementations were synthesized: (i) a baseline version ("Original") that did not use loop pipelining approach; (ii) a second version ("Pipelined") that uses loop pipelining approach, but does not use loop unrolling; and (iii) a final version ("Unrolled") that employs a twofold unrolling for its loop along with the loop pipelining approach. Simulation and area estimation tools from the Haste suite were used to quantify the latency, throughput and area of the resulting implementations. Since the performance of some of the examples is data-dependent, input streams containing as many as 100 data sets were used, and latency and throughput results were averaged over them.

Results. Tables 5.2–5.3 summarize the experimental results. Table 5.2 presents the

Table 5.2: Synthesis Results: Performance Benefit

Algorithm/ Approach	Area (μm^2)	Latency (ns)	Throughput (Mega items/s)	Normalized Throughput
BISECT				
Original	28928	1946	0.51	1
Pipelined	98420	15960	1.03	2.0
Unrolled	184400	16000	1.76	3.4
BTREE				
Original	2900	40	24.65	1
Pipelined	7335	110	41.91	1.7
Unrolled	10840	75	79.62	3.2
CRC				
Original	4405	66	14.99	1
Pipelined	10730	193	19.79	1.3
Unrolled	15080	137	40.21	2.7
GCD				
Original	1770	108	9.11	1
Pipelined	4998	390	12.24	1.3
Unrolled	6574	277	23.51	2.6
ODE				
Original	8931	571	1.75	1
Pipelined	15610	1338	3.61	2.1
Unrolled	25630	1156	7.07	4.1
POLY				
Original	23661	367	2.71	1
Pipelined	53880	1300	5.19	1.9
Unrolled	99280	1226	8.80	3.2
TEA				
Original	30390	1205	0.83	1
Pipelined	96720	2529	4.04	4.9
Unrolled	166500	1704	8.07	9.7

area, latency and throughput obtained for each of the benchmarks. The final column presents the normalized throughput (relative to the "Original" version), to illustrate the performance benefit of loop pipelining. Finally, Table 5.3 summarizes the area and latency overheads of loop pipelining by normalizing with respect to the "Original" version.

Discussion. The results demonstrate that a substantial impact on throughput is achieved by loop pipelining: up to 9.7x speedup. Without the use of loop unrolling, the loop pipelining approach obtains a throughput improvement by a factor of 1.3 to 4.9. When a twofold loop unrolling was applied along with loop pipelining, the speedup obtained was even higher: a

factor of 2.6 to 9.7x.

In greater detail, algorithms that were pipelined into a relatively small number of stages (CRC and GCD) had a limited potential for loop pipelining because the maximum capacity of the self-timed ring structures in these cases was low. As a result, the throughput benefit was around 1.3x (without unrolling). On the other hand, algorithms that were highly pipelined (BISECT, ODE, and TEA), had larger ring structures which could accommodate a greater number of data sets concurrently. For these benchmarks, the throughput increase was substantially higher: a factor of 2 to 4.9x (without unrolling).

As expected, the twofold unrolling led to a throughput increase in each case by a factor of 1.7 to 2.0x, yielding an overall combined throughput benefit of 2.6 to 9.7x relative to the original implementation.

Although loop pipelining approach results in a significant boost in throughput, there are costs associated with the performance improvement. Shown in Table 5.3 are the increases in total area consumed, and in the average latency per data set.

In terms of area, the pipelined version adds the overheads of loop control discussed in Section 5.3.3. Each pipeline stage must latch data from the previous stage, so algorithms with many stages or large contexts (*e.g.* BISECT) will see a large increase in area. By unrolling the loop twofold, the total area of the implementation increased by a factor of 1.3–1.9x.

The average latency for a data set also increased when loop pipelining was used. This was expected because, like most traditional pipelining approaches, loop pipelining approach increases throughput at the expense of latency. The latency overhead in most of the benchmarks was in the 1.4–3.6x range, except for the example that contained a nested loop, BISECT. For BISECT, the latency overhead reported is substantially higher (8.2x) due to the compounded overheads of its nested loops.

While the area and latency overheads may seem daunting, for most applications the main performance measure is overall execution time. By increasing latency and chip area, a dramatic improvement in throughput results, reducing execution time by a large factor.

Table 5.3: Area and Latency (Relative Overheads)

Algorithm	Pipelined		Unrolled	
	Area (Norm.)	Latency (Norm.)	Area (Norm.)	Latency (Norm.)
BISECT	3.4	8.2	6.4	8.2
BTREE	2.5	2.7	3.7	1.9
CRC	2.4	2.9	3.4	2.1
GCD	2.8	3.6	3.7	2.6
ODE	1.8	2.3	2.9	2.0
POLY	2.3	3.3	4.2	3.3
TEA	3.2	2.1	5.5	1.4

Table 5.4: Slack matching results

Example	Goal Tpt	Stages Added		Simulated tpt		% speedup
		MILP	Heuristic	before	after	
CORDIC cond	0.167	8	8	0.0909	0.167	83.72
CORDIC	0.083	0	0	0.083	N/A	N/A
CRC	0.357	3	3	0.292	0.352	20.55
ODE	0.0182	0	0	0.0183	N/A	N/A
GCD	0.05	0	0	0.049	N/A	N/A
raytracing	0.222	145	145	0.161	0.222	37.89
MULT	0.167	8	8	0.0387	0.167	331.52

5.6.3 Results for Slack Matching

Table 5.4 highlights the effectiveness of slack matching by comparing the throughput of the systems before and after slack matching. In each case, the throughput increases as expected due to the addition of slack stages. The table also compares the number of stages added using the MILP and Heuristic algorithms. The heuristic algorithm placed stages optimally in each of these examples, even though they were chosen to represent a wide range of hierarchical structures, including nestings of parallel and sequential constructs as well as conditionals and loops. This result emphasizes the assertion that the heuristic will often lead to a completely optimal solution.

Note that some examples do not require any additional slack stages. These examples are actually quite interesting, because they highlight the fact that the heuristic algorithm will successfully avoid adding unnecessary stages. For example, the inner conditional of the CORDIC algorithm, when analyzed separately from the rest of the system, requires 8 slack stages to op-

erate at its maximum speed. However, when this conditional is nested within the rest of the CORDIC system, the algorithm no longer chooses to place these stages. It successfully determines that other stages within the system act as a bottleneck and that placing additional stages will not improve performance.

5.6.4 Results for Bottleneck Identification

The bottleneck identification method was applied to a set of examples that represent a range of different topologies, as described below: **1)** CORDIC: parallel and sequential **2)** CRC: conditional and sequential **4)** ODE: parallel, sequential, and loop **5)** MULT: parallel In addition, these results report two different versions of the CORDIC example, one of which contains a conditional and another which does not.

TRIC	Bottleneck Type	Node Type
Coalescing	Type I	Leaf node
Stage Splitting	Type II & III	Leaf node
Duplication	Type II & III	Any node
Buffer Insertion	Type III	Sequential node
Loop Unrolling	Type II & III	Loop nodes

Table 5.5: TRIC applicability

TRICs Used Section 5.3.1 gives examples of several different TRICS that can be used to alleviate bottlenecks. For the experiments given here, the tool uses a set of five TRICs: coalescing, stage splitting, duplication with wagging, buffer insertion, and loop unrolling. Table 5.5 shows the bottleneck types and circuit node types that each of the TRICs applies to. While this is not an exhaustive set of possible circuit transformations, it is sufficient in that each type of bottleneck is handled by at least one of the TRICs.

Bottleneck Identification. Table 5.6 shows the results of bottleneck identification. The table reports the size of the example in terms of number of nodes in the hierarchical tree

representation of the system. It also shows the predicted throughput of the system based our analysis method, and for comparison shows Verilog simulation results for the same example; this indicates that the analysis is quite accurate for these examples. Stage latencies were chosen such that a FIFO stage has a forward and reverse latency of 1 ns each (*i.e.*, a cycle time of 2 ns). More complex stages had correspondingly longer latencies. The table further reports the number of limiting segments found for each example, and the runtime on a 2.1 GHz Intel Core 2 Duo machine.

		Throughput (MHz)		# Limiting Segments Found			
Example	Nodes	Simulated	Analysis	Forward	Top	Reverse	Time (ms)
CRC	26	286	292	16	1	12	42
Cordic Cond	30	90.9	90.9	12	2	9	21
Cordic	43	83.3	83.3	0	3	0	40
ODE	13	18.2	18.3	0	8	0	27
MULT	29	38.5	38.7	10	1	3	20

Table 5.6: Bottleneck identification: finding limiting segments

5.6.5 Results for Bottleneck Alleviation

Example	Throughput			# iterations	Type			TRICS
	orig	goal	final		I	II	III	
CRC	286	342	345	4	1	0	3	coalesce; add buffers
Cordic cond	90.9	109	111	2	0	0	2	add buffers
Cordic	83.3	100	101	2	0	1	2	split stages
ODE	182	218	267	1	3	0	0	split stages; duplicate
MULT	38.4	46.2	62.5	6	5	0	1	coalesce; add buffers

Table 5.7: Iterative bottleneck alleviation

Bottleneck Alleviation. Table 5.7 shows the bottleneck alleviation results attained through iterative application of the bottleneck identification method to reach a goal throughput. Each of the examples presents a different challenge to removing bottlenecks. In order to highlight

how examples of similar sizes can require a different number of iterations and different TRICS to reach the same throughput, the goal throughput for the iterative algorithm shown in Figure 5.20 is set to to be the same for each example and iterate till the goal is reached.

In each example, the goal is 20% higher than the original throughput and performed some number of TRICs to reach that goal throughput. For each example, there are many different choices for eliminating bottlenecks; the results here represent one possible set of choices for each example.

In every case, the system was able to reach the desired throughput through iterative application of the manual bottleneck alleviation method method. The table reports the number of iterations that completed before reaching the throughput goal, the type of bottlenecks that were targeted, and the TRICs used for each example. Although the bottleneck identification method greatly aids the designer in performing these optimizations, choosing from the set of possible optimizations and then applying the optimization to the circuit is still in the hands of the designer. Therefore the total amount of optimization that can take place is limited by designer effort, and improving these examples much more than the 20% goal could be time prohibitive for most design flows.

5.6.6 Results for Automated Constrained Optimization

Experimental Setup

Circuit Examples This section uses five examples of varied topologies and sizes. For each example, Table 5.8 indicates which type of circuit constructs make up the specification, the initial throughput as analyzed by the method of Chapter 4, and the number of nodes in the hierarchical representation of the circuit.

Delay Models, and Area, Energy For these experiments, the estimates for delay, area, and energy for each stage are based on an estimate of its complexity. The delay model estimates the forward and reverse latency of a buffer stage at 1 ns, for a cycle time of 2 ns. Stages with

Name	Circuit Type	Nodes	Throughput
ODE	parallel, sequential, and loop	22	0.018
MULT	parallel	10	0.038
CORDIC	parallel and sequential	39	0.083
CRC	conditional and sequential	23	0.029
JPEG	Loop, conditional, and sequential	40	0.00047

Table 5.8: Information about five circuit examples

logic have longer forward latencies, based on a rough estimate of the complexity of the logic (*e.g.* one full adder has twice the forward delay of the buffer stage). Reverse latencies are not affected by the presence of logic. Similarly, each stage is modeled with a normalized area of one for each buffer stage and an energy of two for each buffer stage.

Experimental Results

Comparing solution methods. The running time and solution quality for any example varies based on the type of search method used. Table 5.9 compares the the runtime of the basic method described in Section 5.5.2 on a 2.1 GHz Intel Core 2 Duo machine. For each example, the throughput goal is low enough to obtain meaningful results from the exhaustive method while being high enough to highlight the differences between the methods. CORDIC, CRC, and ODE have a 50% throughput improvement goal while MULT has a 2x throughput goal and JPEG has a 20x throughput goal. This experiment uses the cost function of energy over the square of throughput, which is equivalent to the common $E\tau^2$ metric [32]. It also uses $E\tau^2$ as its search heuristic during the greedy and lookahead searches.

Table 5.9 shows results for each solution method. For easy comparison of relative solution quality, the table shows costs normalized such that the cost of the exhaustive solution is 1 unit. The lookahead method appears twice with two different lookahead depths, 1 and 3. For every example, the exhaustive method is the most time consuming while the greedy method is the least time consuming. In addition, the exhaustive method gives a lower cost solution than the greedy solution for every example. Note that for some examples, the exhaustive method did

not fully finish after running for over 2 hours. The result shown is the best out of the partial set of results.

Based on these results, the lookahead 1 search seems to be the best tradeoff between runtime and solution quality. It found the best solution possible for CRC and JPEG and is within 4% of the best solution for ODE and MULT. On CORDIC, however, it yields a cost that is 80% higher than the best solution. In addition, as the throughput goals get higher, the error in the lookahead 1 search will likely increase as well.

Runtime Improvement with Tree Pruning Table 5.10 shows the runtimes with the addition of the pruning methods described in Section 5.5.3. Greedy is not included because it will not benefit much from additional tree pruning. All the other search methods find the same quality of solutions as those without pruning, but with at a much faster runtime. For easy comparison of relative solution quality, the table shows costs normalized such that the cost of the exhaustive solution is 1. The quality of the results, in terms of throughput and cost, indicates that the advanced pruning methods introduced in Section 5.5.3 do not eliminate good solutions from the search space.

Lookahead 1 search has the least amount of runtime benefit from the use of additional pruning, since it was already searching a much smaller number of vertices than the other methods. Both lookahead 3 and the exhaustive search method show large improvement in the execution time with additional tree pruning. JPEG shows the most improvement, with the exhaustive and lookahead 3 searches improving by 2662x and 428x respectively. Although ODE shows the least amount of improvement, the speed improvement is still quite high: exhaustive and lookahead 3 search runtimes improve by 5x and 8.7x respectively.

Based on these results, the search method with the best tradeoff between execution time and solution quality is lookahead 3. In all but one of the examples, it reached a solution with the same cost as the exhaustive solution. In ODE, it was just 3.7% higher. For these reasons, lookahead 3 used in the following experiments to gather all further results.

	ODE			MULT			CORDIC			CRC			JPEG		
Search Method	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)
Exhaustive	0.036	1.00	4.95	0.151	1.00	883	0.250	1.00	>7200	0.5	1.00	>7200	0.015	1.00	>7200
Greedy	0.036	1.07	0.07	0.103	2.09	0.07	0.167	2.12	0.05	0.5	1.02	0.07	0.013	1.04	0.20
Lookahead 1	0.036	1.04	0.06	0.154	1.02	0.22	0.182	1.80	0.14	0.5	1.00	0.58	0.015	1.00	15.3
Lookahead 3	0.036	1.04	2.45	0.151	1.00	15.7	0.250	1.00	28.2	0.5	1.00	507	0.015	1.00	495

Table 5.9: Comparison between search methods

	ODE			MULT			CORDIC			CRC			JPEG		
Search Method	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)	Throughput	Normalized Cost	Time (s)
Exhaustive	0.036	1.00	0.99	0.151	1.00	21.5	0.250	1.00	66.9	0.5	1.00	17.6	0.015	1.00	2.70
Lookahead 1	0.036	1.04	0.06	0.154	1.02	0.20	0.182	1.80	0.14	0.5	1.00	0.43	0.015	1.00	0.39
Lookahead 3	0.036	1.04	0.28	0.151	1.00	4.22	0.250	1.00	11.7	0.5	1.00	5.11	0.015	1.00	1.16

Table 5.10: Speed improvement with additional tree pruning

				$E\tau^2$		$E\tau \cdot a$		$Energy$		$area$		$Energy \cdot area$	
	Initial throughput	Increase	Goal throughput	Final throughput	Cost	Final throughput	Cost	Final throughput	Cost	Final throughput	Cost	Final throughput	Cost
ODE	0.018	1.5	0.027	0.036	21175	0.031	17160	0.030	24	0.030	28	0.031	528
	0.018	2	0.036	0.036	21175	0.038	26640	0.036	26	0.038	36	0.038	1008
MULT	0.038	2	0.077	0.151	2151	0.100	20700	0.077	44	0.077	44	0.077	1936
	0.038	5	0.192	0.414	327	0.400	21725	0.200	50	0.200	100	0.200	4704
CORDIC	0.083	1.5	0.125	0.250	1392	0.128	57215	0.143	81	0.128	88	0.128	7304
	0.083	2	0.167	0.333	801	0.274	61136	0.186	81	0.167	152	0.186	12636
	0.083	3	0.250	0.500	364	0.404	55259	0.333	86	0.255	204	0.255	17748
CRC	0.286	1.5	0.429	0.500	196	0.500	7056	0.500	49	0.500	144	0.476	3360
	0.286	2	0.571	0.571	266	0.571	19488	0.571	82	0.571	220	0.571	9676
JPEG	4.73E-04	20	9.47E-03	1.53E-02	420445	1.28E-02	1044108	1.28E-02	97	1.28E-02	136	1.28E-02	13386
	4.73E-04	40	1.89E-02	2.22E-02	212625	2.22E-02	1096200	2.22E-02	105	2.22E-02	232	2.22E-02	24360

Table 5.11: Results using different cost functions and goal throughputs

Varying Cost Metrics and Goals Based on the execution time and solution quality of the examples in the previous sections, lookahead 3 is a efficient method for finding a high-quality solution, if it is not necessary that the solution be completely optimal. Table 5.11 therefore uses the lookahead 3 search method to find the solution for two different cost functions. Each example is tested for two different throughputs: ODE, CORDIC, and CRC have goals of 50% and 2x throughput improvement while MULT has goals of 2x and 5x throughput improvement and JPEG has goals of 20x and 40x.

For each example, five different cost metrics are used with the search method. The search heuristic used for each cost was simply the cost over throughput. In every case, the search-based bottleneck alleviation method was able to find a solution that meets the throughput goal. This indicates its ability to work with a variety of different cost functions.

5.7 Conclusion

This chapter presented a framework for iterative bottleneck removal. The results show the optimization framework can identify a series of circuit transformations that reach the desired throughput goal while maintaining a low value for the given cost function. The framework was able to achieve between 50% and 20x throughput improvement when using the $E\tau^2$ cost metric on five example circuits without advanced pruning techniques. With the addition of advanced pruning and searching techniques, it was able to achieve between 2x and 40x throughput improvement on all example circuits. When comparing the different search methods, the results also indicate that lookahead 3 search offers a good trade-off between algorithm run time and solution quality compared to greedy search and exhaustive search. The framework can handle a variety of different cost metrics, and the results show successful optimization for five different cost functions: $E\tau^2$, energy·area·, energy alone, area alone, and the energy-area product. All execution times using advanced pruning techniques were less than two minutes for all examples over all cost functions and throughput goals. The results as a whole indi-

cate that our framework is capable of quickly providing high-quality solutions for minimizing various cost functions while meeting a throughput goal.

Chapter 6

Testing and Design for Testability

6.1 Introduction to Testing Asynchronous Pipelines

Most asynchronous high-speed pipeline styles achieve high performance by making timing assumptions, thereby sacrificing some timing robustness. Even if these timing assumptions are verified during design, they may be violated in practice due manufacturing variations and defects. Therefore, the ability to test for these timing violations in a fabricated chip containing high-speed pipelines is critical to inspire confidence in the use of asynchronous hardware. In short, the analysis and optimization methods for pipelined asynchronous systems presented in Chapters 4 and 5 are of little practical use if the resulting systems cannot be tested for errors.

Timing violations are quite challenging to test, both in terms of test application and test pattern generation. Many timing violations occur only under certain operating conditions that seem difficult to create using typical low-speed testing equipment. In addition, a single timing violation may cause errors on one or several wires at once, leading to non-deterministic error behavior that is difficult to test.

Several approaches [26, 46, 53] for testing timing violations have been proposed earlier, but they are intrusive, *i.e.,* they require additional test circuitry to be added to the pipeline. The extra circuitry has an area overhead typically proportional to the number of stages in the pipeline. In addition, the added circuitry is typically on the critical path of each stage's cycle,

thereby negatively impacting the pipeline's performance.

This chapter presents a new approach to expose timing violations of asynchronous pipelines that directly addresses the acknowledged challenges to exposing timing violations: activating "at-speed" errors using low speed testing equipment and handling non-deterministic error behavior. Moreover, my approach applies not only to linear pipelines, but also to any hierarchical composition of forks, joins, conditional constructs, and data-dependent loops. Finally, the pattern generation approach maps timing violation testing to stuck-at-fault testing of a dual circuit, thereby allowing us to leverage existing stuck-at ATPG tools.

A key distinguishing feature of this approach is the ability to handle errors on *multiple* wires caused by a single timing violation, unlike [53] that assumes such errors mostly occur singly. In particular, unlike stuck-at faults where the likelihood of multiple errors is usually quite low, timing constraint violations due to setup or hold time violations can easily cause errors in multiple data bits simultaneously. Therefore, the fault model of this chapter is significantly more general. A key contribution of this approach is a method of test pattern generation that can expose any combination of multiple errors due to such timing violations.

A significant benefit the testing approach presented in this chapter is that it is minimally-intrusive, and hence low-overhead, yet it can create the operating conditions necessary to expose timing violations using only low speed testing equipment. In fact, my approach is non-intrusive for linear pipelines; no additional circuitry is required to be added in order to test them. Although some additional circuitry must be added for full fault coverage of pipelines that have forks and joins, the total number of extra gates added is proportional to the number of forks and joins present, not to the total number of stages.

The remainder of the chapter is organized as follows. Section 6.2 discusses previous work on asynchronous pipeline testing, including stuck-at-fault testing for MOUSETRAP. It also provides background on three asynchronous pipeline styles which illustrate my testing approach: MOUSETRAP [55], GasP [59], and high-capacity (HC) pipelines [54]. Section 6.3 proposes a classification of timing constraints into two key categories—*forward* and *reverse*—

and proposes test strategies for timing constraint violations that occur when these constraints are violated. Section 6.4 extends these testing strategies to paths containing forks and joins. Section 6.5 applies the test strategy to linear MOUSETRAP pipelines, and introduces a test pattern generation method which can test for timing constraint violations that cause errors on an undetermined number of wires, and Section 6.6 then generalizes this work to handle pipelines containing forks and joins. Finally, Section 6.7 offers conclusions and directions for future work.

6.2 Previous Work and Background on Asynchronous Testing

This section discusses previous work in the testing of asynchronous pipelines and presents background on three high-speed asynchronous pipeline styles—MOUSETRAP, GasP, and high-capacity (HC)—which are used in the remainder of the paper as illustrative examples.

6.2.1 Previous Work

Several methods have been proposed for testing asynchronous micropipelines. Pagey *et al.* [43] proposed methods for generating test patterns to test stuck-at faults in traditional micropipelines, but do not consider violations of the timing constraints within pipeline stages. A more recent approach to testing micropipelines is by [27], which focuses on test sequence generation for testing both stuck-at and timing violations inside C-elements.

Several authors [26, 46] have proposed full-scan approaches for testing micropipelines. Though these full-scan methods test both stuck-at faults and violations of the bundled delay constraint, they are high-overhead in both area and speed. Roncken *et al.* [49, 50] proposed a more optimal approach that employs partial scan. Their approach targets faults in both the control and datapath, and covers not only stuck-at fault testing, but also bridging faults as well as IDDQ testing.

Recently, van Berkel *et al.* [64] have proposed adding synchronous as well as LSSD modes of operation to asynchronous circuits in order to make them testable. In addition, te Beest *et al.* [62] present an approach that uses only multiplexers, instead of latches, to break feedback loops, and thus leverage combinational test pattern generation without the overhead of scan latches. Both of these approaches were proposed mainly in the context of the Tangram/Haste design flow. Finally, Kondratyev *et al.* [28] have proposed a test approach for NCL circuits.

None of the above approaches, however, addresses the specific test needs of fine-grain high-speed asynchronous pipeline styles. In particular, they are either specific to a particular circuit style, or are intrusive and can cause unacceptable performance overheads.

An interesting case study of non-intrusive testing was the approach developed at Intel to test the RAPPID asynchronous instruction length decoder [52, 51]. This approach had minimal area and performance overheads, yet provided 94% stuck-at fault coverage. Of the faults that were missed, a detailed analysis was made to classify them into benign and potentially catastrophic faults. While this work included an initial analysis of timing constraint violations, it did not provide a comprehensive timing constraint violation test approach, focusing instead mostly on stuck-at faults.

Very recently, Shi *et al.* [53] have presented an approach specifically targeted to testing high-speed asynchronous pipelines. Testing of both stuck-at and timing constraint violations is addressed. There are two limitations of their approach to exposing timing constraint violations: (i) it is intrusive and therefore has area and performance overheads, and (ii) it uses a limited timing constraint model that assumes the fault will only affect a single bit.

The approach of this paper overcomes the limitations of [53] and provides a minimally-intrusive fault testing strategy. It also handles a larger class of timing constraint violations, in which a single fault can cause errors on one or more bits simultaneously.

6.2.2 Background: Asynchronous Pipeline Styles

Throughout this paper, three different pipeline styles serve as illustrative examples: MOUSETRAP, GasP, and high-capacity (HC) pipelines. These pipeline styles represent a range of implementation design decisions. This subsection highlights the main features of each pipeline style.

MOUSETRAP [55] is a two-phase pipeline style for static logic datapaths. Although it uses transition signaling on the request and acknowledge lines, it uses level signaling to control the latches. During the operation of a MOUSETRAP pipeline, all the latches begin as transparent and only become opaque once data has been captured. Two simple one-sided timing constraints must be satisfied for correct operation: *setup time* and *hold time*. The former simply requires that the data arrive at least one setup time before the latch closes; otherwise the data may not be latched properly. On the other hand, the latter constraint requires that, once data enters a stage, it should be securely captured at least a hold time before new data is produced by the previous stage; otherwise, the data may be overwritten by new data.

GasP [59] is a static logic pipeline style that aggressively exploits timing assumptions in order to provide very high performance. A key distinctive feature of GasP is that a single wire, called the state conductor, is used to transmit *both* the request and acknowledge signals between a pair of adjacent stages. This feature eliminates the redundant return-to-zero event present in typical four-phase pipeline styles, and GasP is thereby able to combine some of the benefits of four-phase and two-phase signaling. However, for correct operation, GasP requires several timing constraints to be satisfied, some of which are complex 2-sided requirements.

The final pipeline style discussed in this paper is the high-capacity (HC) pipeline [54]. It is a four-phase pipeline style that targets dynamic logic datapaths. It is *latchless*. Instead, the dynamic logic of each stage has an "isolate phase", in which its output is protected from further input changes. A timing constraint must be satisfied to avoid premature termination of a stage's evaluation; this is similar to a setup constraint. Similarly, a hold time requirement must also be met.

The next section tabulates some of the constraints of the three pipeline styles, introduces categories for their classification, and proposes a strategy for the testing of their violation.

6.3 General Approach for Testing Asynchronous Pipelines

This section introduces the test strategy for high-speed asynchronous pipelines. It focuses on testing for timing constraint violations, since test methods for stuck-at faults have already been proposed in [53].

Section 6.3.1 begins by identifying two key categories of timing constraints that must hold for correct operation: *forward* and *reverse* constraints. Section 6.3.2 then gives functional test methods for loading and unloading a pipeline that are most likely to expose these timing constraint violations.

The test approach is generalized in Section 6.4 to handle forks and joins, which represents a first step toward handling systems with arbitrary topologies. Test pattern generation is deferred till Section 6.5.

The test strategy is illustrated using examples of timing constraints from three different pipeline styles: MOUSETRAP, GasP, and high-capacity (HC) pipelines.

6.3.1 Classification of Timing Constraint Violations

For the purposes of designing fault tests, I classify timing constraints into two key categories: *forward* and *reverse* constraints.

A *forward* timing constraint is one that requires a downstream event to occur after an upstream event, *i.e.,* in the same direction as the flow of data. More specifically, the pipeline operates correctly only if a particular event in stage N occurs before a particular event in stage $N + 1$. Typical examples of forward constraints include *setup time requirements* of many pipelines: the latch in stage $N+1$ must "capture" a data item and be disabled only after the output of stage N has been generated and stabilized. Such forward constraints typically

Pipeline Style	Timing Constraint
MOUSETRAP	$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{Latch}_N} + t_{\mathrm{XNOR}_N\downarrow}$
GasP	$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{inva}_N} + t_{\mathrm{NAND}_N} + t_{\mathrm{reset}_N} + t_{\mathrm{INVc_N}\downarrow}$
HC	$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{aC}_N} + t_{\mathrm{delay}_N} + t_{\mathrm{INV}_N}$

Table 6.1: Forward Constraint examples

prevent situations where a data item is not correctly transmitted from one stage to next. Violations of forward timing constraints are likely to manifest when a data item is allowed to travel through an uncongested or empty pipeline; such a scenario allows downstream events to occur unimpeded, thereby exposing violations of the forward constraint.

A *reverse* timing constraint, on the other hand, is one that requires an upstream event to occur after a downstream event, *i.e.,* counter to the flow of data. More specifically, the pipeline operates correctly only if a particular event in stage N occurs before a particular event in stage $N-1$. Typical examples of reverse constraints are *hold time requirements* of many pipelines: once a data item is received in stage N, the latch in stage N must be disabled before new data is generated by stage $N-1$. Such reverse constraints typically prevent current data from being erroneously overwritten by new data. Therefore, violations of reverse timing constraints are likely to be exposed when one data item follows closely behind another.

In some pipeline styles, timing constraints exist that do not fit into either of these categories. For example, GasP has a timing constraint that relates two events within a single pipeline stage. I have not yet devised a standard method for testing for violations of these types of timing constraints.

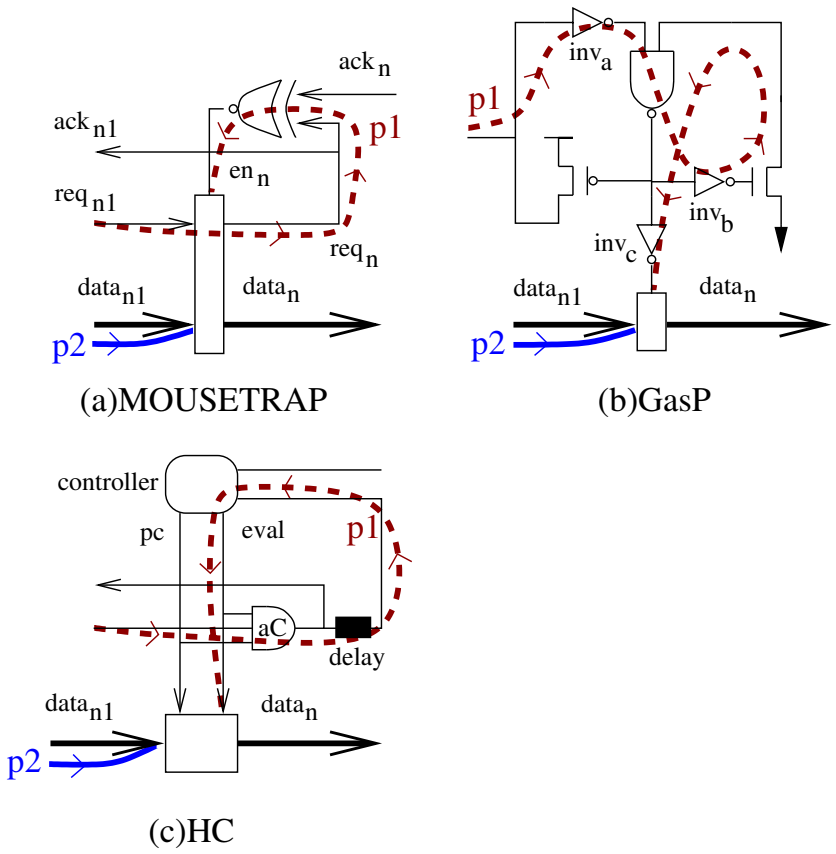


Figure 6.1: Examples of forward timing constraints.

6.3.2 Test Strategy for Linear Pipelines

Forward Timing Violations

Examples. Many pipeline styles have forward timing constraints, most commonly in the form of the bundled data constraint. Table 6.1 gives the forward timing constraints for MOUSETRAP, GasP, and HC pipelines. In this table, $t_{\text{data}_{N-1}}$ and t_{req_N} refer to the arrival times of data and request from stage $N-1$; t_{setup_N} is the setup time for the latch (or dynamic gate) in stage N; t_{reset_N} is the time for the self-resetting NAND in GasP to reset itself; all other terms are the delays associated with components identified in their labels. For clarity, Figure 6.1 represents these constraints graphically. For each pipeline, two paths are highlighted: $p1$ and $p2$. For correct operation, $p1$ must be *longer* than $p2$.

The constraints shown are closely related to the bundling assumption in that they require

that the data bits do not arrive much after their associated request signal. These constraints are actually less strict than the bundling constraint since there is some delay between the time the request arrives and when the data is actually needed. In particular, strictly speaking, the bundled data constraint requires that $t_{\mathrm{data}_{N-1}} < t_{\mathrm{req}_{N-1}}$. However, the setup time constraints shown in the table and the figure are somewhat more relaxed: in both MOUSETRAP and GasP, the data must arrive at stage N one setup time before the latch closes; in HC pipelines, the data must arrive one setup time before the evaluation phase ends (*i.e.,* isolate phase begins). Thus, the constraints identified are more properly referred to as *setup time* constraints.

Violations of these timing constraints actually manifest themselves as pipeline malfunction only when there is a bubble in stage N, *i.e.,* stage N is ready to process the new data from stage $N-1$. This is so because the above setup time constraints were derived assuming the worst-case scenario that stage N is ready to capture the data arriving from the previous stage. In each of the three pipeline styles, if stage N is not ready to receive new data, then more time is available for that data to arrive from stage $N-1$ and stabilize at stage N's inputs before it must be captured. In particular, in MOUSETRAP and GasP, if stage N is not empty, its latch remains disabled. Similarly, in HC pipelines, if stage N is not empty it will not commence its next precharge-evaluate cycle.

Testing Approach. The observation that stage N must be empty for a setup time violation to manifest itself helps us construct a test for these forward timing violations. I first present a scenario that exposes these faults, and then present a functional test strategy to efficiently test an entire pipeline.

Figure 6.2 illustrates a scenario in which setup time violations are likely to be exposed. Assume the scenario begins with the snapshot of the pipeline shown in Figure 6.2(a): stage N is empty and stage $N-1$ has data item $d1$. If the pipeline operates correctly, the data item $d1$ flows to the right, as shown in snapshot (b). However, if the setup time constraint between the two stages is violated, the data item $d1$ will not be properly captured (*i.e.,* latched

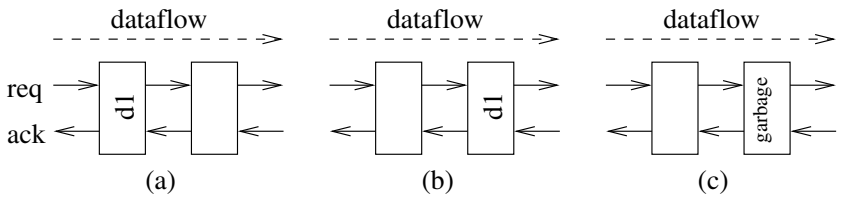


Figure 6.2: a) Beginning state b) Correct end behavior c) Behavior with forward timing violation

for MOUSETRAP and GasP, or evaluated for HC) by stage N. In the case of MOUSETRAP and GasP, the data item $d1$ will be corrupted by any stale data that was previously in stage N, since some or all of the bits of $d1$ could not be properly latched. For HC pipelines, the result in stage N could be all zeros (*i.e.,* the result of the previous precharge), or partial evaluation of some of the bits before evaluation was prematurely interrupted.

In order to test the entire pipeline for setup time violations, the above scenario must be created at each stage in the pipeline. Once again, the recent test approach of [53] accomplishes this goal by added circuitry to the pipeline for greater controllability, but the intrusiveness of that approach causes loss of performance during actual operation.

The strategy, once again, is to create this error-exposing situation functionally, without the need for intrusive circuitry. We start with an empty pipeline. Next, we feed one data item to the pipeline. As the data item travels forward, it creates the condition shown in 6.2 for each stage. Specifically, the setup time constraint between stages 1 and 2 is tested first, between 2 and 3 next, and so on all the way to the rightmost stages in the pipeline.

Reverse Timing Violations

Examples. Examples of reverse constraints can be found in many pipeline styles. Table 6.2 gives the reverse timing constraints for MOUSETRAP, GasP, and HC pipelines. In this table, t_{hold_N} is the hold time for the latch (or dynamic gate) in stage N; all other terms are the delays associated with components identified in their labels. For clarity, these constraints are graphically shown in Figure 6.3. Once again, for each pipeline, two paths are highlighted: $p1$

Table 6.2: Examples of reverse timing constraints: Analytical expressions.

Pipeline Style	Timing constraint
MOUSETRAP	$t_{\mathrm{XNOR}_N\downarrow} + t_{\mathrm{hold}_N} < t_{\mathrm{XNOR}_{N-1}\uparrow} + t_{\mathrm{Latch}_{N-1}} + t_{\mathrm{logic}_{N-1}}$
GasP	$t_{\mathrm{reset}_N} + t_{\mathrm{INVc_N}\downarrow} + t_{\mathrm{hold}_N} < t_{\mathrm{pull\_up}_N} + t_{\mathrm{NAND}_{N-1}} + t_{\mathrm{INVc}_{N-1}\uparrow} + t_{\mathrm{Latch}_{N-1}} + t_{\mathrm{logic}_{N-1}}$
HC	$t_{\mathrm{delay}_N} + t_{\mathrm{INV}_N} + t_{\mathrm{hold}_N} < t_{\mathrm{NAND}_{N-1}} + t_{\mathrm{aC}_{N-1}} + t_{\mathrm{delay}_{N-1}} + t_{\mathrm{INV}_{N-1}} + t_{\mathrm{Eval}_{N-1}}$

and $p2$. For correct operation, $p1$ must be *shorter* than $p2$.

Each of the constraints shown in Table 6.2 and Figure 6.3 are *hold time* constraints. In particular, the constraint for MOUSETRAP states that, once a data item enters stage N, that stage's latch must be disabled at least a hold time (t_{hold_N}) before the previous stage can perturb the data at stage N's inputs. The constraint for GasP similarly ensures that stage N's latch is disabled at least a hold time before new data arrives from stage N. Finally, the constraint for HC ensures that stage N, upon evaluating a data item, enters its *isolate phase* (during which the dynamic logic in that stage is neither precharging nor evaluating) before stage $N-1$ goes through a complete new cycle of precharge and evaluation to generate a new data item.

Violations of these timing constraints actually manifest themselves as pipeline malfunction only when a new data item is readily available for processing by stage $N-1$. This is so because the above hold time constraints were derived assuming the worst-case scenario of a steady stream of input data. For instance, in MOUSETRAP, a violation of the above timing constraint causes an actual pipeline malfunction only if there is actually a new data value or request (or both) on the input to stage $N-1$. Similarly in GasP and HC, a timing violation causes an error only if there is new data available on the input side of the datapath of stage $N-1$.

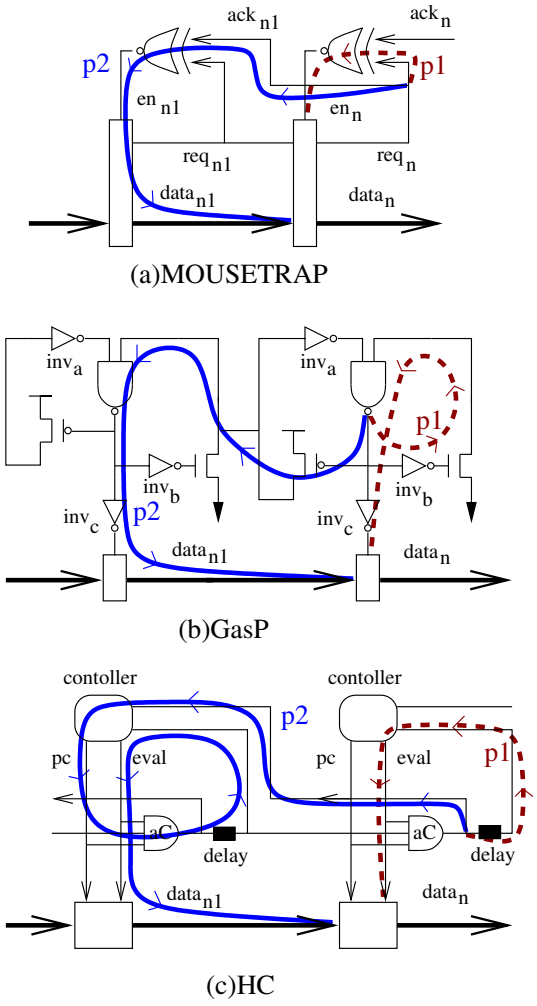


Figure 6.3: Examples of reverse timing constraints

Testing Approach. The observation that a new data item must be waiting at the input side of stage $N-1$ in order for a hold time violation to manifest itself helps us construct a test for these reverse timing violations. I first present a scenario that exposes these faults, and then present a functional test strategy to efficiently test an entire pipeline.

Figure 6.4 illustrates a scenario in which hold time violations are likely to be exposed. Assume the scenario begins with the snapshot of the pipeline shown in Figure 6.4(a): there are two data items, $d1$ and $d2$ in stages $N-1$ and $N-2$, respectively, and there is a bubble in stage N. If the pipeline operates correctly, the two data items remain distinct as they flow to the right, taking the pipeline through the snapshots (b) and (c). However, if the hold time

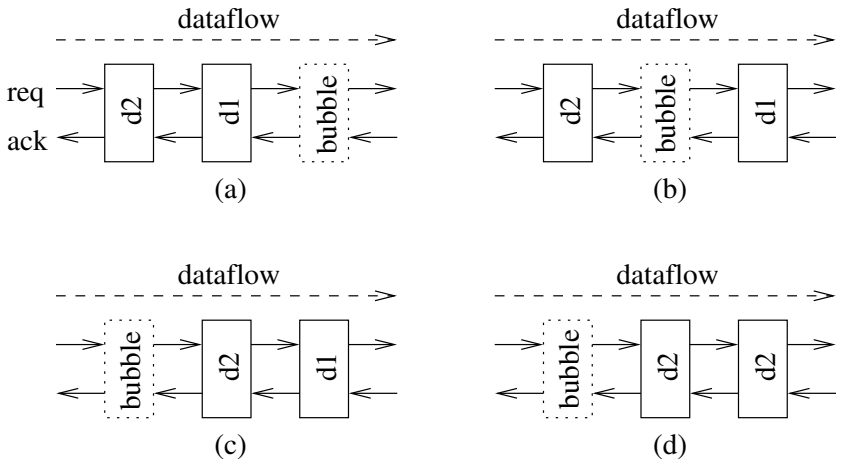


Figure 6.4: a) Beginning stage b) bubble propagates backwards c) Correct end behavior d) Behavior with reverse timing violation

constraint between the two rightmost stages is violated, the data item $d1$ will be overwritten by the item $d2$, as shown in snapshot (d).[1]

In order to test the entire pipeline for hold time violations, the above scenario must be created at each stage in the pipeline. The recent test approach proposed in [53] accomplishes this goal by adding circuitry to the pipeline to provide adequate controllability. However, that approach is intrusive and creates overheads to the normal functioning of the pipeline, thereby resulting in a loss of performance. Moreover, the intrusiveness of that approach also modifies the timing constraint itself, thus perturbing the very constraint it is trying to verify.

The strategy instead is to create this error-exposing situation functionally, without the need for intrusive circuitry. We start with an empty pipeline. Next, we fill the entire pipeline with data, while withholding acknowledgments to the rightmost stage. We then add one bubble to the right end of the pipeline by sending a single acknowledgment. As this bubble propagates leftward through the pipeline, it sequentially creates the situation shown in Figure 6.4 for each stage. Specifically, the hold time constraint between stages $N-1$ and N is tested first, followed by the constraint between stages $N-2$ and $N-1$, and so on all the way to the

[1]For simplicity of presentation, we assume that when a hold time violation occurs, a data item is completely overwritten by the next data item. Strictly speaking, it is possible that only some bits of a data item are overwritten. Section 6.5 addresses this more general scenario.

leftmost stages in the pipeline.

There are several key benefits of this new approach. First, unlike [53], this new approach is non-intrusive, with no overhead to steady-state performance. Second, this approach can test hold time faults for an entire pipeline in a single sweep, *i.e.,* by propagating a single bubble backward through the pipeline. In contrast, the approach of [53] must test for hold time violations at each pipeline stage individually, thereby requiring significantly greater test effort. Finally, much like the approach of [53], this new approach only requires low-speed ATE, yet provides at-speed testing for timing constraint violations.

6.4 Test Strategy for Non-Linear Pipelines: Handling Forks and Joins

Thus far, I have only discussed testing strategies for faults in straight pipeline paths. Here, I describe extensions to the test strategy to handle pipelines with forks and joins, which is a first step toward handling systems with arbitrary topologies.

6.4.1 Forward Timing Violations

Challenges. Testing forward timing violations offers some challenges. Specifically, setup time violations between the join element and all of its immediate predecessors are difficult to test. If data from one branch arrives at the join much after data from the other, only the later-arriving data will expose setup time violations. The earlier-arriving data will have had sufficient time to stabilize before the join reacts, thereby masking any setup time violations between that branch and the join stage.

Testing Approach. To ensure testing of setup time faults between a join stage and each of the branches that feed into it, one must be able to control which branch provides a data item first. The strategy is to modify the pipeline by inserting extra circuitry immediately before

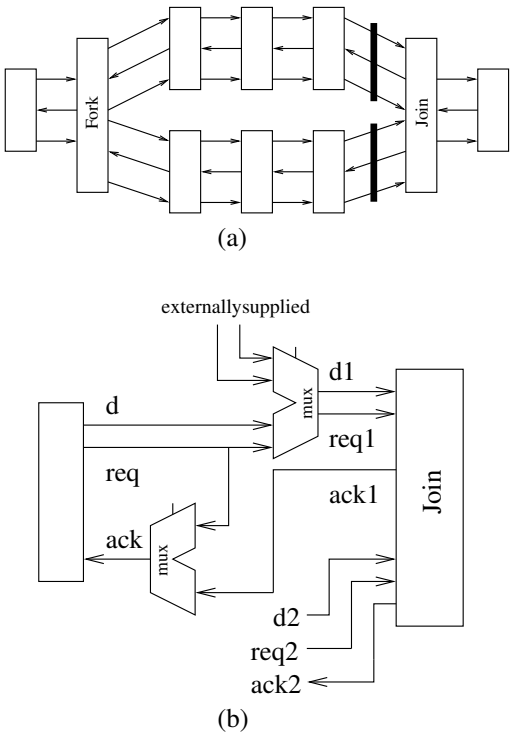


Figure 6.5: a) Extra circuitry for forward delay. b) Generic implementation

each join, which allows the testing environment to directly provide data items to the input side of the join, as shown in Figure 6.5.[2] Specifically, multiplexors are added to both the control and the data input of each join stage. This allows data to be fed into the join externally during testing. They key idea is to provide data externally for each input interface of a join stage, *except* for the input interface that is to be tested for setup time violations.

Testing takes place as follows. Assume we are testing the interface between the join stage and the lower branch of the pipeline shown in Figure 6.5(a). First we initialize the pipeline to be empty. Then we set the multiplexors in the circuitry of Figure 6.5(b) on the upper branch of the join so that data is read from the testing environment and sufficient time is allowed for this data to be read and stabilized. At this point, the setup time constraint between the upper branch and the join stage is trivially satisfied because the join will not latch this data until

[2]In order to conserve the number of input pins used by the test environment, external data is typically supplied bit-serially, and converted to bit-parallel on chip before being supplied to the circuit under test.

data also arrives from the lower branch; therefore, a potential setup time violation between the lower branch and the join stage is now exposed and testable. The test proceeds by sending a single data item into the pipeline from its left interface, which flows rightward through the pipeline, effectively exposing setup time faults throughout the lower branch, through the join and all the way through the right end of the pipeline. Finally, we read the output from the pipeline and examine it for errors. The test is then repeated for the upper branch that feeds into the join.

6.4.2 Reverse Timing Violations

Challenges. Testing reverse delays presents two challenges: (i) "unbalanced" branches, *i.e.,* forked paths whose branches have unequal number of stages, are not fully testable using only functional methods; and (ii) fork stages themselves are difficult to test robustly for hold time violations. I discussed these challenges briefly before describing the testing approach I use.

Unbalanced branches: It is challenging to functionally test pipelines with forked paths that reconverge if the branches have unequal number of stages, *i.e.,* if the branched paths are not "slack matched" [33]. In particular, the approach for exposing reverse timing constraint violations, presented above for linear pipelines, relies on filling the entire pipeline with data items, and then propagating a bubble backward through it. If unbalanced reconvergent forks exist in the pipeline, then not all branches can be completely filled with data, and therefore this functional test strategy cannot be directly applied to the longer of the branches.

In more detail, assume that a fork creates two branches, $B1$ and $B2$, which reconverge. Assume branch $B1$ has $N1$ stages and branch $B2$ has $N2$ stages. Then, if $N1 > N2$ then the first $N1 - N2$ stages of branch $B1$ cannot be made to hold data while the test pattern is applied, rendering them functionally untestable using the method presented thus far. However, it is important to note that the shorter of the two branches, $B2$, is still fully testable for reverse timing violations, a fact that will be exploited by the new test approach.

Fork stages: If all fork branches are balanced, then the test approach for linear pipelines

can be easily adapted to test for hold time violations throughout the non-linear pipeline, *except* to test for faults between the fork stage and its immediate successors. This difficulty is because of the unpredictability of the order of arrival of bubbles at the fork stage from each of its successors. In particular, if one branch of a 2-way fork propagates a bubble towards stage N faster than the other branch, then only the later-arriving bubble can actually expose a timing constraint violation upon reaching the fork stage. The earlier arriving bubble will not be able to expose a hold time violation because the fork stage, which is waiting for the other bubble to arrive as well, cannot immediately generate a new data item. Therefore, to fully test for hold time violations, greater controllability of pipeline operation is required in order to control the order of arrival of bubbles at a fork stage from different fork branches.

Testing Approach. I now present the test strategy, which addresses both of the above challenges.

Unbalanced branches: This new approach avoids the problem due to unbalanced branches by requiring that all reconvergent paths are balanced (*i.e.,* slack matched). This is easily done by either inserting buffer stages in the shorter branch, or by pipelining the longer branch more coarsely using fewer stages. Fortunately, slack matching is already considered good design practice for performance reasons [33], so requiring it for ease of timing constraint violation testability is not too onerous a requirement.

Fork stages: If full testability is required for hold time violations between a fork stage and all of its immediate successors, one then needs to modify the circuit in order to control the order of arrival of bubbles at that fork stage. The strategy is to add a small amount of circuitry to the pipeline, as shown in Figure 6.6. Specifically, on each of the branches, one adds an extra pipeline stage, which is externally configurable to act as either a regular pipeline stage (*i.e.,* capable of storing one data item or bubble) or as pass-though logic with no storage capability.

Essentially, we are configuring one branch to have N stages and configuring the other branch to have $N + 1$ stages. Thus, we have deliberately made the branches *unbalanced*

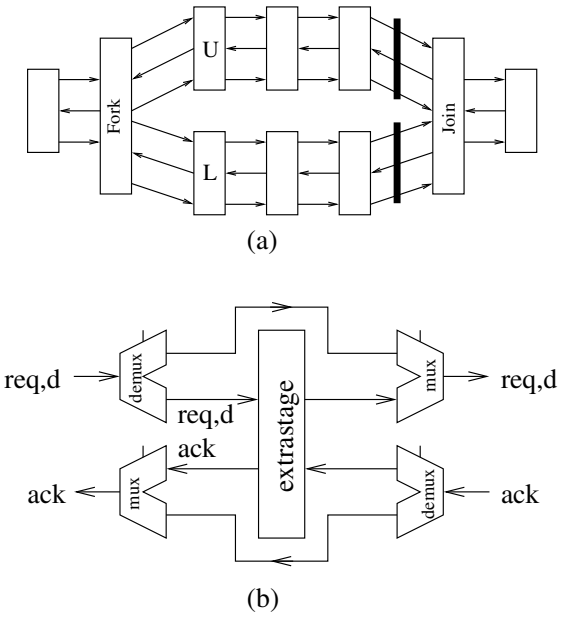


Figure 6.6: a) Extra circuitry for reverse delay. b) Generic implementation

during testing, a technique that allows the shorter branch to be fully testable. This procedure is then repeated to test the other branch.

For example, assume we are testing the for timing violations between the fork stage and the stage U in Figure 6.6. First we initialize the pipeline to be empty. Then, we set the external control signals such that the extra circuitry in the upper branch behaves as pass-through logic, but the extra circuitry on the lower branch behaves as a regular pipeline stage. Thus, during this test, the upper branch is shorter than the lower one.

Next, we load a test sequence of eight data elements through the left interface of the pipeline, without providing acknowledgments at the right interface, and wait for a sufficient time for the pipeline to stabilize. At this point the upper branch is completely filled, whereas the lower branch has one bubble in stage L.

Then, we remove one data item from the right end of the pipeline, thereby creating a bubble that propagates leftward through the pipeline. Since there is already a bubble in stage L, the fork will be ready to react quickly when the bubble reaches stage U, thereby exposing any reverse timing violations between stage U and the fork stage.

Errors are easily detected by reading out all of the data from the pipeline and examine it for functional correctness. This testing procedure is then repeated to similarly test for reverse timing violations between stage L and the fork stage.

6.5 Test Example I: Linear MOUSETRAP

This section applies the testing strategies for linear pipelines given in Section 6.3 to MOUSETRAP. For each of the possible timing violations in a MOUSETRAP pipeline, either the forward or reverse fault testing strategy is applied. A key contribution is a test pattern generation approach which can help expose these timing violations.

MOUSETRAP has two distinct timing constraints that must be satisfied for correct operation, one forward (setup time) and one reverse (hold time), as were listed in Tables 6.1 and 6.2. A violation of the first constraint can lead to data corruption because the data fails to arrive a setup time before the latch is disabled. A violation of the second constraint can lead to *two* distinct failure scenarios: "control overrun" (req_N is overwritten by req_{N+1}) and "data overrun" ($data_N$ is overwritten by $data_{N+1}$). Test methods and pattern generation for testing these timing constraint violations for linear MOUSETRAP are now presented.

6.5.1 Forward Timing Violations

Setup Time Fault. Violations of the setup time requirement are tested using the test approach for forward timing constraint violations introduced in Section 6.3.2. Specifically, we initialize the pipeline to be empty, then feed test data to the pipeline from the input side, and read the data from the output side of the pipeline to determine if a timing constraint violation exists. Determining test data patterns to expose such faults, however, is not trivial.

Recent work [53] has introduced a method for test pattern generation to expose this type of fault. Their method separately tests for setup time violations for each data bit for each pipeline stage. Specifically, it suggests sending one test pattern to initialize that bit to 0, and

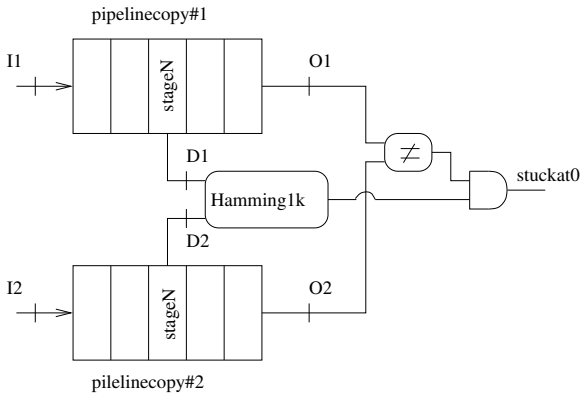


Figure 6.7: ATPG for testing setup time violations.

then sending a second test pattern that checks for a stuck-at-0 fault on that bit. If there is a setup time violation at that bit location, then the bit value will remain at 0 and behave as if it were stuck at 0; otherwise, the bit is correctly set to 1 by the second test pattern. Thus, their work nicely leverages stuck-at ATPG tools to help test for setup time violations.

There is, however, a significant limitation in the test pattern approach of [53]: it relies on the assumption that the setup time fault will affect a single bit in the stage being tested. In practice, however, if one of the lines coming out of a stage is corrupted due to a setup time violation, the other lines coming out of it are likely to be corrupted as well. The combination of multiple errors in one stage could easily lead to the fault's not propagating to the end of the pipeline.

This new approach overcomes this limitation of [53] and can correctly test for setup time faults that may affect multiple bits simultaneously. In particular, this test pattern generation strategy exposes a setup time violation for a particular bit (say, bit k) in a given stage (say, stage N) regardless of whether or not other bits are also affected by that fault. The key idea is to generate pairs of test data items (say, I_1 and I_2) which satisfy the following criteria: (i) they generate values (say, D_1 and D_2) at stage N which *differ in only the k-th bit,* (*i.e.,* other bits receive values that are identical for I_1 and I_2) and (ii) they produce different outputs at the right end of the pipeline. Thus, this test pattern suite is more constrained than the suite

of [53]; this restriction guarantees that simultaneous errors on multiple bits, which are likely in practice, are correctly handled.

Test approach. This test method consists of two steps: sending the first test data item (I_1) through an empty pipeline and removing it from the other end, followed by sending the second test data item (I_2). If criteria (i) is met, a setup time violation can *only* cause the bit k to change, since all other bits are the same between I_1 and I_2 If criteria (ii) is met, then a fault for bit k in stage N is propagated to the output, and is therefore functionally testable. This procedure is repeated for all bits in all pipeline stages.

ATPG setup. This approach to generating the test pattern sequence (I_1, I_2) leverages existing ATPG tools for stuck-at fault testing. Specifically, we map the problem of finding the test patterns I_1 and I_2 that will expose a setup fault at bit k in stage N in the pipeline into an equivalent problem of testing for a stuck-at-0 fault at the output of the dual circuit shown in Figure 6.7. The block labeled "Hamming-1_k" produces 1 if its inputs differ in only the k-th bit; otherwise it generates 0.[3] Therefore, any input pattern $I_1 I_2$ that tests for output stuck-at-0 fault also satisfies the conditions required for the sequence (I_1, I_2) to be a test pattern for exposing a setup time violation in bit k in stage N. In particular, the inequality tester ensures that the two input patterns will generate different pipeline outputs, and the Hamming distance checker ensures that the intermediate values in stage N differ in only bit k. Finally, a test for stuck-at-0 at the output of the AND gate effectively ensures that the test patterns I_1 and I_2 that are generated satisfy the conjunction of these conditions.

Example. Figure 6.8 shows a two-stage pipeline with one level of combinational logic. To check setup time faults for the bit *o1*, ATPG yields a test sequence consisting of the following two data items: $i_1 i_2 i_3 = 011$ followed by $i_1 i_2 i_3 = 010$. Table 6.3 shows the correct and faulty behaviors.

[3]The block tests if its inputs have a Hamming distance of 1, and the only different bit is bit k.

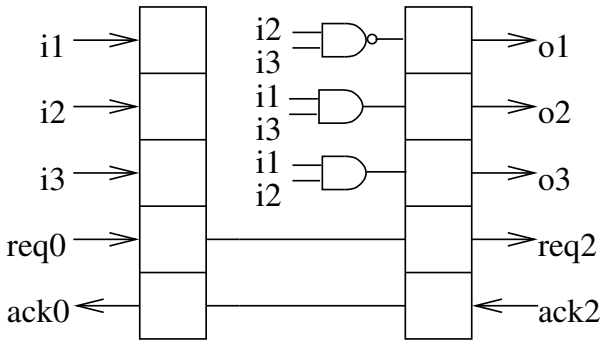


Figure 6.8: A two-stage pipeline with one level of combinational logic.

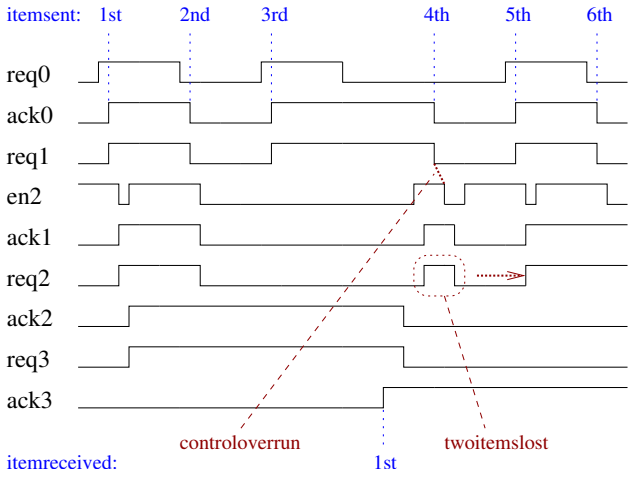


Figure 6.9: Timing diagram showing control overrun

6.5.2 Reverse Timing Violations

The reverse timing violation in MOUSETRAP occurs due to a violation of the hold time requirement. This faults can manifest itself in two flavors: *control overrun* and *data overrun.*

Control Overrun. Control overrun occurs when the hold time constraint for MOUSETRAP (see Table 6.2) is violated for the incoming *req* signal. That is, once a stage receives a data item along with a request, it fails to disable its latch before a new request overwrites the current request. Since this is a reverse timing violation, we use the testing strategy outlined in 6.3.2. Specifically, we fill the pipeline with data and then insert one bubble at the right end of the pipeline, which exposes faults as it propagates backwards.

	S.No.	1	2
Test Pattern	req0 ack2 (i1,i2,i3)	1 0 (0,1,1)	0 1 (0,1,0)
Correct Behavior	ack0 req2 (o1,o2,o3)	1 1 (0,0,0)	0 0 (1,0,0)
Faulty Behavior	ack0 req2 (o1,o2,o3)	1 1 (0,0,0)	0 0 (0,0,0)
	Comment	send inputfor o1=0	send inputfor o1=1

Table 6.3: Test patterns for checking setup time fault for line $o1$ in Figure 6.8.

The error behavior for control overrun is that two requests, and therefore two pieces of data, are lost. This is because MOUSETRAP uses transition signaling. For example, Figure 6.9 indicates how six requests will be interpreted as only four requests in the presence of a fault.

One can test for this fault quite easily in a linear pipeline. First we fill the pipeline with N data items, where N is the length of the pipeline, while withholding acknowledgments from the right end of the pipeline. Then we present the $N + 1$-th data item to the pipeline, although at this point it will not be accepted by the pipeline. Now, every stage in the pipeline has new data available to its left. Next, we remove one item from the right end of the pipeline, effectively introducing one bubble which flows leftward to expose any hold time faults. When this bubble reaches the leftmost pipeline stage, the $N + 1$-th data item is accepted by the pipeline. Thus, if there is no fault, the pipeline will once again contain N data items at this point. If there were any hold time violations, then two items will have been lost for each violation, and therefore only $N - 2$ or fewer items would remain in the pipeline. The correct

S.No.	Test Pattern (req0,ack3)	Correct Behavior (ack0,req3)	Faulty Behavior (ack0,req3)	Comment
1	(1,0)	(1,1)	(1,1)	apply 1st req
2	(0,0)	(0,1)	(0,1)	apply 2nd req
3	(1,0)	(1,1)	(1,1)	apply 3rd req
4	(0,0)	(1,1)	(1,1)	apply 4th req
5	(0,1)	(0,0)	(0,0)	add one bubble
6	(1,1)	(0,0)	(1,0)	apply 5th req, observe ack1

Table 6.4: Test pattern for control overrun

and faulty behaviors are easily distinguished: the correct pipeline is full and will not accept any new data, but the faulty pipeline will accept two or more data items.

As an example, Figure 6.9 shows the behavior of a 3-stage linear pipeline that has a control overrun fault in $stage_2$. First we supply four data items along with their requests; then we remove one item to cause error behavior. Specifically, *req2* falls while *en2* is still asserted, thereby allowing an incorrect request through. This error is exposed when two subsequent requests are acknowledged by the pipeline. Table 6.4 shows the test pattern that exposes this error. In general, the test sequence for a control overrun fault in a linear N-stage pipeline contains $N + 3$ test patterns.

Data Overrun. Unlike control overrun, data overrun does not result in the loss of requests. Instead, data becomes corrupted. Specifically, this hold time violation causes data item D_N to be overwritten partially or entirely with data item D_{N+1}. In MOUSETRAP, data overrun occurs when the hold time constraint is violated only for the latches that capture the data bits and not for the tiny latch that captures the associated request; this scenario is quite possible because data can arrive from the previous stage much earlier than its associated request.

ATPG setup. The test for data overrun is more complex than that for control overrun because test patterns must be generated carefully to expose data corruption, as opposed to simply detecting loss of data items. As with setup time faults, if a data overrun fault affects one data bit of a stage, it is likely to affect other bits of the same stage as well. Therefore, testing

S.No.	Test Pattern			Correct Behavior			Faulty Behavior			Comment
	req0	ack9	d0	ack0	req9	d9	ack0	req9	d9	
1	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 1st item, recv 1st item
2	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 2nd item
3	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 3rd item
4	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 4th item
5	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	send 5th item
6	0	0	(0,0,0,0)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 6th item
7	1	0	(1,1,1,1)	0	1	(1,1,1,1)	1	1	(1,1,1,1)	send 7th item
8	1	1	(1,1,1,1)	1	0	(0,0,0,0)	1	0	(0,0,0,0)	recv 2nd item
9	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	1	(1,1,1,1)	recv 3rd item
10	1	1	(1,1,1,1)	1	0	(0,0,0,0)	1	0	(0,0,0,0)	recv 4th item
11	1	0	(1,1,1,1)	1	1	(1,1,1,1)	1	**0**	(1,1,1,1)	recv 5th item
12	1	1	(1,1,1,1)	1	0	(0,0,0,0)	empty: req9 same			recv 6th item
13	1	0	(1,1,1,1)	1	1	(1,1,1,1)	empty: req9 same			recv 7th item

Table 6.5: Test pattern for control overrun on a forked path

sequences must be generated for data overrun faults in a manner similar to those generated for setup time faults in Section 6.5.1.

Interestingly, the ATPG approach of Figure 6.7 also directly applies to generating tests for the reverse timing violation. Suppose we are testing stage N for a data overrun fault. In order to begin testing, we must fill the pipeline up to and including stage N with some data; these values do not affect the testing of stage N. The next two data items fed into the pipeline are the same patterns generated by the method of Figure 6.7: I_1 followed by I_2. Then, one item is removed from the right end of the pipeline, and a bubble propagates backward exposing the error caused by the timing violation for bit k in stage N. If the pipeline operates correctly, stage N will correctly latch the value D_1; otherwise it will latch the value D_2. The correct and faulty scenario are therefore easily distinguished at the output of the pipeline.

6.6 Test Example II: Non-Linear MOUSETRAP

This section describes how to test for timing constraint violations in MOUSETRAP in the presence of forks and joins. Setup time faults and data overrun faults use the same test pattern generation methods as their linear counterparts described in Section 6.5. For control overrun

faults, however, we must use a different test pattern than the one given for linear pipelines.

6.6.1 Setup Time Fault

Since the setup time fault is a forward timing violation, we use the testing strategy introduced in Section 6.4.1. Specifically, for full fault coverage we add logic that allows the join stages to be tested. Test patterns for forked paths can be generated in the same way as for straight paths, as described in Section 6.5.1.

6.6.2 Control Overrun

Since control overrun is based on a reverse timing constraint, testing forked paths faces the challenges given in Section 6.4.2. Specifically, we must ensure that forked paths are balanced and add logic to allow testing of the fork stages.

The input patterns for testing control overrun faults for straight pipelines given in Section 6.5.2 will not work in the presence of forks. Recall the the linear pipeline test relied on being able to detect if the pipeline has lost two (or more) data items after being filled. However, if the fault occurs on only one branch of a fork, the branch *without* the fault will not lose any data items, and will thereby prevent the loss of data in the other branch from being externally observable. In particular, the correctly functioning branch will not allow more data items from being accepted from the environment even though the faulty branch contains bubbles.

For better fault coverage in pipelines with forks, I propose a different strategy. Instead of determining if the pipeline has lost any data items due to timing constraint violations by trying to feed it more items, we determine any losses by reading out the entire contents of the pipeline and counting the number of items read. If fewer than N items emerge from an originally filled pipeline, then some data has been lost due to control overrun.

Figure 6.10 shows an example of a forked pipeline to be tested. Table 6.5 shows a test pattern that we use to test for control overrun faults. In the correct scenario, all N of the acknowledges (*ack9*) sent to the right interface of the pipeline lead to further requests coming

out of the pipeline (*req9*). In the faulty scenario, only $N - 2$ of the acknowledges lead to new requests being generated.

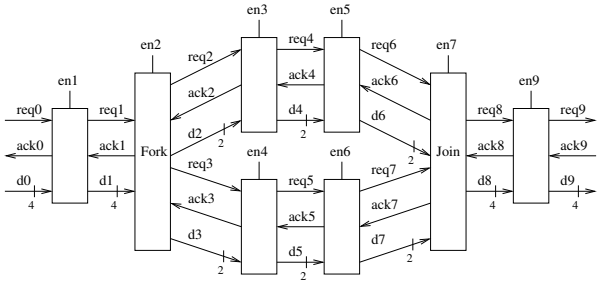


Figure 6.10: A non-linear MOUSETRAP pipeline

6.6.3 Data Overrun

Since data overrun is based on a reverse timing constraint, we use the test setup given in 6.4.2. Specifically, we must ensure that forked paths are balanced and add logic to allow testing of the fork stages. Test patterns for forked paths can be generated in the same way as for straight paths, as described in Section 6.5.2.

6.7 Conclusions and Possible Extensions

In this chapter, I presented a strategy for exposing timing constraint violations in high-speed asynchronous pipelines, including pipelines with forks and joins. I showed that our testing strategies are applicable by giving examples from three different pipeline styles: GasP, MOUSETRAP, and HC. This testing strategy is non-intrusive for linear pipelines and minimally-intrusive for pipelines containing forks and joins. In addition, all of these tests can be conducted using low-speed testing equipment.

I also demonstrated one specific application of this strategy. Using MOUSETRAP as an example, I showed how our testing strategies work for both straight paths and paths with forks and joins. We also showed how to generate test patterns for MOUSETRAP that are robust to

the non-deterministic nature of errors caused by timing constraint violations. This method leverages past ATPG tools.

There are several natural future opportunities to this work. Although many delay constraints are either forward or reverse constraints, there are still other kinds of timing constraints to identify and develop test strategies for. By testing pipelines with forks and joins, this work took one step towards testing pipelines with arbitrary topologies. We still need to incorporate testing of more complex pipeline stages, such as data dependent conditional forks. In addition, one could apply the testing strategies presented here to circuits with non-trivial combinational logic.

Finally, there is a subtle limitation of the current test approach for MOUSETRAP: the setup fault test approach will actually not be able to expose a setup violation in a stage if data has accumulated positive margins in prior stages. In particular, positive timing margins in prior stages sometimes rescue a small timing violation in a later stage, very much like time borrowing in synchronous systems, allowing the pipeline to still operate correctly. While this feature makes MOUSETRAP even more robust, if the intent of the test strategy, however, is to expose even these types of violations, then this setup fault test strategy needs to be enhanced further.

Chapter 7

Circuit Designs

7.1 Introduction

In previous chapters, all the pipelined circuits introduced and discussed were assumed to have efficient and robust implementations. Moreover, these implementations were also assumed to be capable of modular composition with other stages (*i.e.* preceding and succeeding stages need not be aware of the internal behavior or implementation). This chapter introduces the implementation of five types of special-purpose pipeline stages: conditional split, conditional select, conditional join, merge without arbitration, and an arbitration stage.

Together with pipeline stage designs introduced in previous literature [55], these stages form a comprehensive suite of circuits that can be combined modularly to create any of the hierarchical circuits described in previous chapters. The circuits presented here represent one possible set of circuits that can be used to implement any of the hierarchical, dataflow circuits that are the focus of this thesis.

Section 7.2 first presents background work on the MOUSETRAP pipelines that the new circuits are designed to work with. Section 7.3 then goes on to describe the behavior of components that are needed to implement each type of construct used by analysis algorithm of Chapter 4: sequential, parallel, conditional, and iterative constructs.

The remaining sections detail each of the five new components. In particular, each com-

ponent is described using gate-level diagrams, and the operation of the component is given in detail. In addition, many of the components have several different implementations that represent tradeoffs between speed and robustness. Each circuit includes a synchronization point performance analysis, which will yield the performance metrics necessary to complete the type of analysis presented in Chapter 4. The description of each implementation also includes the forward and reverse timing constraints, which are an important part of the testing method introduced in Chapter 6.

7.2 Background

The work of [55] presents the MOUSETRAP pipeline style. MOUSETRAP is a two-phase pipeline style that uses transition signaling. A transition from high to low or from low to high on the request line indicates that a new data is present, and a similar transition on an acknowledge line indicates that the data has been received and stored.

7.2.1 Linear MOUSETRAP Pipeline Stage

The MOUSETRAP linear pipeline has a very simple implementation, which uses only one *xnor* as the control element. Figure 7.1 shows a linear pipeline stage implementation, as first described in [55].

The forward and reverse delay metrics for a MOUSETRAP stage are given below.

Forward Latency: $t_{latch} + t_{logic}$

Reverse latency: $t_{xor} + t_{latch}$

7.2.2 MOUSETRAP Fork

Figure 7.2 shows a MOUSETRAP stage that is capable of sending requests to and processing acknowledges from two successor stages. The MOUSETRAP fork stage takes one incoming

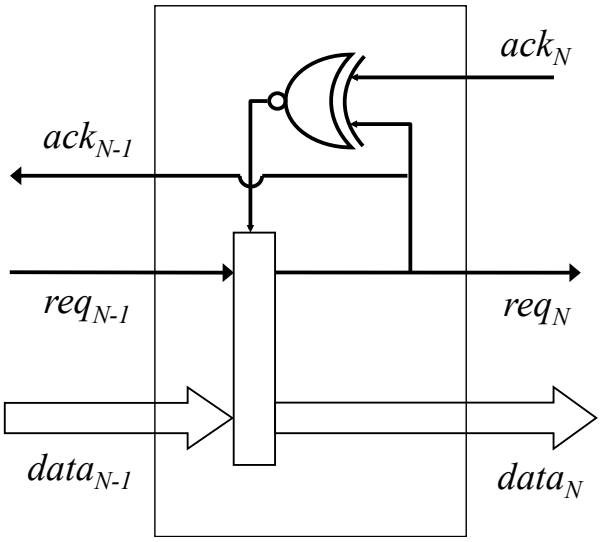


Figure 7.1: A basic MOUSETRAP **stage from [55]**

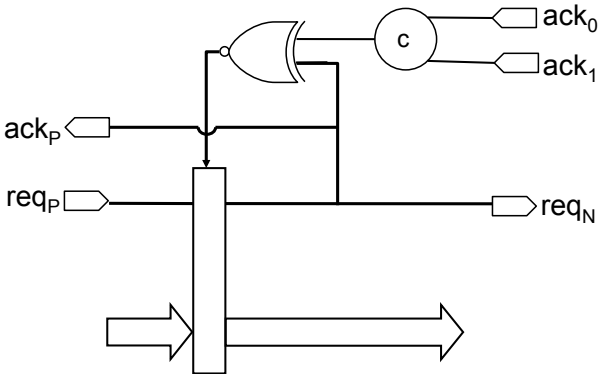


Figure 7.2: A MOUSETRAP **fork stage from [55]**

request from its predecessor stage and forks the outgoing requests to its successor stages. In the figure, the same request signal req_N is sent to both of the successor stages.

The fork combines the incoming acknowledges from the successor stages with a c-element. As a result, the latch controller will not signal the latch to become transparent until after both acknowledges have arrived.

7.2.3 MOUSETRAP Join

Figure 7.3 shows a MOUSETRAP stage that can receive requests from two predecessor stages and send a combined request on to a successor stage. The MOUSETRAP join stage combines

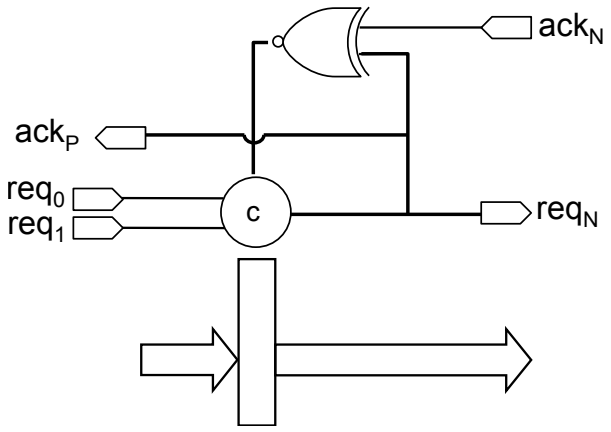


Figure 7.3: A MOUSETRAP join stage from [55]

together the incoming requests from two separate stages using a c-element. The outgoing request does not change until *both* of the incoming requests have changed. The logic within a particular join stage will determine what calculations take place using the joined data.

7.3 Circuit Usage

7.3.1 Stage Description

This chapter introduces a set of five new pipeline stages which, together with the MOUSETRAP stages presented in Section 7.2, create a full suite of pipeline stages capable of implementing all the hierarchical constructs presented in Section 3.3. In particular, these new stages are needed to implement both speculative and non-speculative conditionals as well as pipelined loops. The names and a brief description of the behavior of each stage is listed below.

- conditional split: A conditional split stage is analogous to a router: it receives a data item from an input channel, and a Boolean value (select) from a second input channel. Based on the value of select, the data item is sent to one of the two output channels. In some applications, the Boolean select value and the data value could be bundled together onto the same channel (i.e., the data itself includes the routing information). In yet other applications, the select could be simply be a value provided by the system

without any handshaking (e.g., a global or external input, or a local value that changes infrequently). In these special cases, our proposed circuit implementation for the more general case can be simplified/optimized.

- conditional select: A conditional select stage is similar to an event multiplexer: it has two data input channels, and a third input channel that supplies a Boolean value (select). It has one output channel. The behavior is to first read the select channel; then, based on the value of select, read one of the two data channels, and send the result to the output channel. A value that is not selected is stored to be later sent out when the corresponding Boolean value arrives.

- conditional join: A conditional join is similar to a conditional select, except that all input channels are read even though data from only one of them is forwarded; data from the remaining input channels is discarded. Thus, the handshake behavior is identical to a simple 3-way pipeline join stage. The datapath operation is identical to a combinational multiplexor. This simple implementation waits for both inputs to arrive before sending out one and discarding the other.

- merge without arbitration: This pipeline stage has two input channels and one output channel. Data is read from whichever input channel has new data, and then sent to the output. No arbitration is provided; it is assumed that the input channels are mutually exclusive.

- arbitration stage: This pipeline stage performs arbitration between two input channels, and produces results on two output channels. Only one input channel is read at any time, and its value is sent to its corresponding output channel. This circuit can be combined with the merge without arbitration circuit above to produce a merge with arbitration.

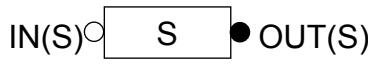


Figure 7.4: A hierarchically composable single pipeline stage

7.3.2 Implementing Hierarchical Constructs

With the introduction of these new stages, all of the hierarchical constructs of Section 3.3 can be realized. This section lists the supported hierarchical constructs and describes their implementation using MOUSETRAP stages. Figures from Section 3.3 are repeated here for the reader's convenience.

Single Stage

The single stage construct

Figure 7.4 shows a single pipeline stage. A single MOUSETRAP stage [55], as described in Section 7.2, implements the single stage hierarchical component.

Sequential

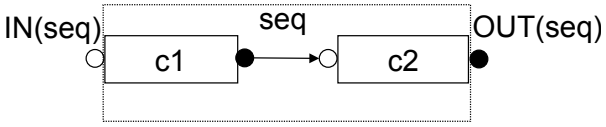


Figure 7.5: A hierarchically composable sequential component

Figure 7.5 shows a sequential component. This component does not need any special-purpose stages as part of its implementation, because all pipeline stages can potentially be composed together in sequence.

Parallel

Figure 7.6 shows a parallel component. The fork and join stages displayed in the Figure are implemented by the MOUSETRAP fork and join stages [55] presented in Section 7.2.

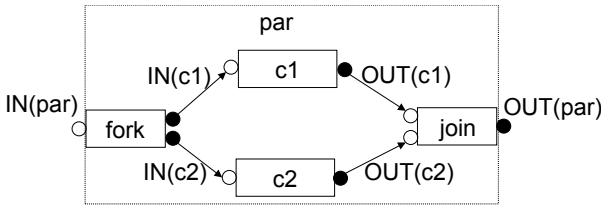


Figure 7.6: A hierarchically composable parallel component

Conditionals

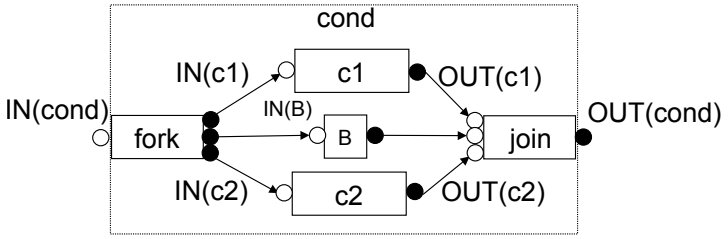


Figure 7.7: A hierarchically composable parallel component with speculative conditional

The MOUSETRAP fork components [55] along with the new conditional join component can be used together to form the speculative conditional found in Figure 7.7. In particular, the MOUSETRAP fork stage [55] needs to be modified to have three rather than two parallel outputs, so it can be used as the fork stage of the speculative conditional. In addition, the conditional join—which was designed for use in a speculative conditional—will take signals from all three paths as input and send out the data that is selected by the Boolean value.

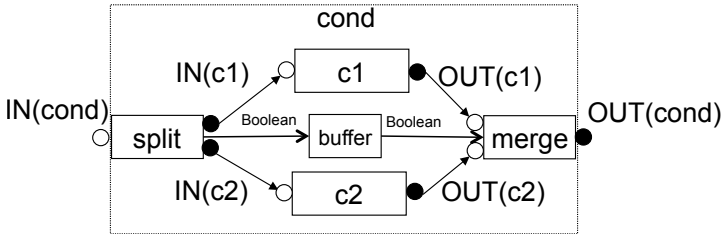


Figure 7.8: A hierarchically composable non-speculative conditional component

The implementation of a non-speculative conditional, as seen in Figure 7.8, requires two of the newly designed stages. In particular, the conditional split and conditional select are designed to work together to implement an if-then-else construct without speculation. The conditional split first sends data along one of two paths, based on a Boolean value, thereby splitting one data stream into two. Subsequently, based on that Boolean value, the conditional select receives data from the correct path, thereby recombining the two data streams into one.

Iteration

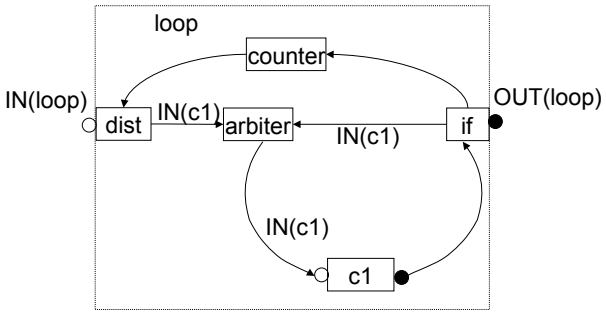


Figure 7.9: A hierarchically composable data-dependent loop component

As seen in Figure 7.9, the loop interface is made up of several stages, each of which requires a different implementation. A new data item entering the loop first arrives at the distributor stage, which sends the new item into the loop if the counter indicates that it is not already full. This joining of information between the incoming data and the counter can be implemented as a MOUSETRAP join stage.

The arbiter of the loop interface prevents collisions between data items that are entering the loop and data items that are already within the loop. It is implemented using two of the new stages that this section introduces: the arbitration stage and the merge without arbitration. In particular, the arbiter first chooses whether the new data arriving into the loop or the old data cycling through the loop arrived first. It then sends a signal out on exactly one of its outputs. The merge without arbitration takes in this signal and sends the data into the loop.

Finally, a conditional split implements the behavior of the *if* block in Figure 7.9. Specifically, it takes in a Boolean conditional that determines whether the data item is ready to leave and sends that item either out of the loop or back to the arbiter. Additionally, every time a data item leaves the ring, a signal should be sent to the counter that acts as a decrement.

7.4 Conditional Split

7.4.1 Behavior

A conditional split stage is similar to a router: it receives a data item from an input channel, and a Boolean value (select) from a second input channel. Based on the value of select, the data item is sent to one of the two output channels. This stage waits for both the Boolean select input and the data input to be ready. It toggles only one of the outgoing request lines, depending on the Boolean value. The data input is simply copied to all of the output channels.

7.4.2 Non-optimized Implementation

A basic non-optimized implementation is shown in the figure 7.10. In this implementation, a C-element combines the incoming data and Boolean requests. The Boolean input value, B, is used to invert one of the incoming acks, ack0 or ack1, to produce the appropriate requests on req_0 and req_1. Negative edge triggered flip flops are used to latch outgoing requests; this prevents changes on the ack_0 and ack_1 lines from producing spurious requests. The latches in the data path (not shown) can be controlled with the same signal, *enable*, that controls the incoming request latch.

Performance Metrics

For performance analysis, the forward latency path of the conditional split included the C-element and the latch, then the *xnor* controller that changes the enable signal, and finally one

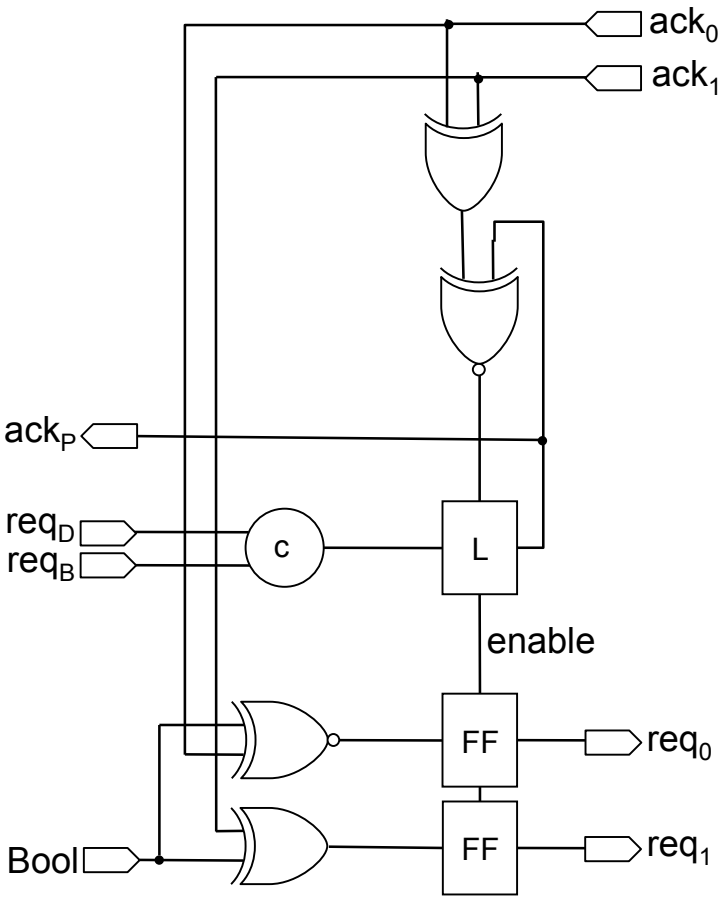


Figure 7.10: Conditional split circuit diagram

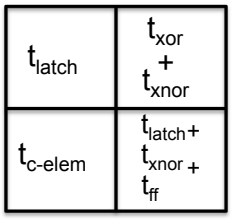


Figure 7.11: Conditional split abstract analysis model

of the two flip flops. The forward latency can be expressed as follows:

Forward latency: $t_{c\text{-}elem} + t_{latch} + t_{xnor} + t_{ff}$

The reverse path goes through the controller *xor* and *xnor* and finally the latch.

Reverse latency: $t_{xor} + t_{xnor} + t_{latch}$

Using the synchronization-point notation of Section 3.2 gives the values of Figure 7.11. These values can be used in the analysis and optimization techniques presented in this thesis.

One simple way to improve the cycle time, and therefore improve the throughput, of this component is to break the conditional into two separate stages. The first stage handles combining the incoming signals, and the second stage handles the outgoing signals. The overhead of this optimization is increased latency and area, but there is no penalty in terms of robustness.

Timing Constraints for Testing

The setup time constraint is less restrictive than that of a typical MOUSETRAP stage, as given in Section 6.3.2, because the extra C-element delay decreases the risk of a timing violation. $t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{c\text{-}elem}_N} + t_{\mathrm{Latch}_N} + t_{\mathrm{XNOR}_N\downarrow}$

The hold time constraint remains identical to the one for a typical MOUSETRAP stage, given in section 6.3.2, since the path for closing the latch remains identical to that of the original stage design. $t_{\mathrm{XNOR}_N\downarrow} + t_{\mathrm{hold}_N} < t_{\mathrm{XNOR}_{N-1}\uparrow} + t_{\mathrm{Latch}_{N-1}} + t_{\mathrm{logic}_{N-1}}$

Reverse Path

The remaining three designs optimize the forward path of the circuit, i.e., the logic that controls the outgoing requests. However, the reverse path—which handles the acknowledgments

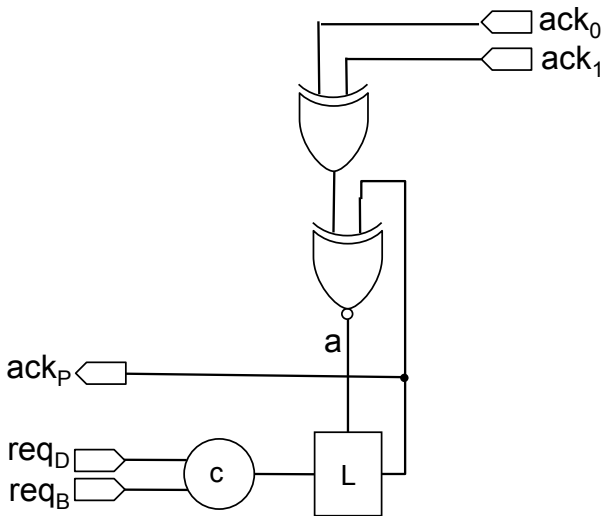


Figure 7.12: The reverse path for all conditional split circuits

sent to the previous stage—remains the same. For reference, Figure 7.12 displays only the reverse path of the conditional split. The wire labeled *a* in the figure serves as an input for the forward path implementations in the following sections.

7.4.3 Options and Optimizations

Basic logic implementation

I generated a circuit with a more optimized forward path by modeling the behavior of the circuit and synthesizing using Petrify [11]. Boolean equations are provided below in Figure 7.13; gate-level circuit implementations can be produced directly from these equations. The signal *req* is the combination of the incoming Boolean and data requests after they have been combined using a C-element, which is similar to the non-optimized design. The signal *a* is the xor of the two incoming *ack* signals, which is also similar to the non-optimized implementation. To produce the signals req_0 and req_1, the circuit uses the signals *req*, *a*, and the Boolean clause to compute the output signals req_0 and req_1.

Using this implementation, the following revised forward latency holds:

Forward latency: $t_{c\text{-}elem} + t_{and} + t_{or} + t_{and} + t_{or}$

```
req = C-element (reqD,  reqB)
req0 = b' (req req1' a' + req' req1 a) + r0 (req' a' + req a + b);
req1 = b (req req0' a' + req' req0 a) + r1 (req' a' + req a + b');
a = ack1 ack0' + ack1' ack0;
```

Figure 7.13: Basic logic minimized expression for conditional split

Generalized C-element implementation

A generalized C-element implementation, generated using petrify [11], for the forward path is shown in Figure 7.14.

```
req = C-element (reqD,  reqB)
[0] = b' (req req1' a' + req' req1 a);
[1] = b' (req req1 a' + req' req1' a);
[req0] = r0 [1]' + [0];        // mappable onto gC
[3] = b (req a' r0' + req' a req0);
[4] = b (req' a r0' + req a' req0);
[req1] = req1 [4]' + [3];        // mappable onto gC
```

Figure 7.14: Generalized C-element implementation of a conditional split

Using this implementation, the following revised forward latency holds:

Forward latency: $t_{c\text{-}elem} + t_{and} + t_{or} + t_{and} + t_{c\text{-}elem}$

7.5 Conditional Select

7.5.1 Behavior

A conditional select stage is similar to an event multiplexer: it has two data input channels, and a third input channel that supplies a Boolean value (select). It has one output channel. The behavior is to first read the select channel; then, based on the value of select, read one of the two data channels, and send the result to the output channel. The circuit begins with the latches opaque. It waits for the Boolean input, and then makes the selected request latch transparent. When the selected request arrives, it triggers the request to be sent to the next stage, and closes the latches once more.

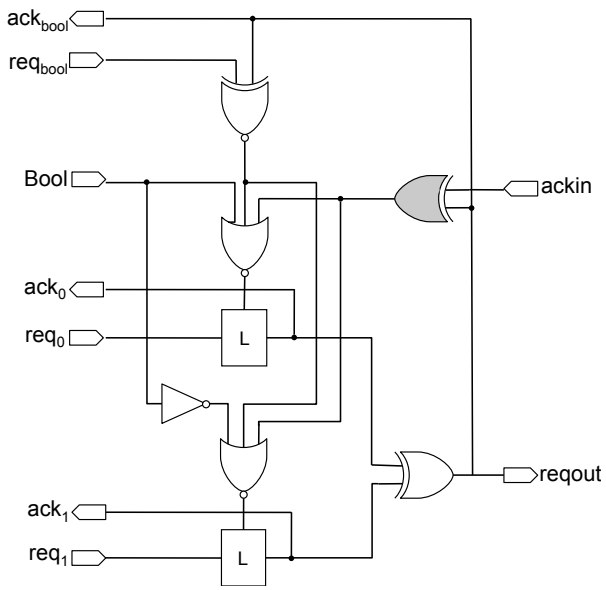


Figure 7.15: Conditional select circuit diagram

7.5.2 Non-optimized Implementation

A basic non-optimized implementation is shown in Figure 7.15. Request latches are held opaque by nor gates until the Boolean arrives. Data request latches become transparent only when all of the following are true: the Boolean has arrived, the Boolean value selects the given data path, and the next stage is ready. The gate that most closely corresponds to the controller xnor in a normal mousetrap stage is shown grayed. The data path can be constructed using multiplexors with the boolean input as the select line. The latches can be controlled with the next ready signal, just as in a normal mousetrap stage.

Performance Metrics

The conditional select has differing forward and reverse latencies for the incoming Boolean and the incoming data paths.

Forward latency (data): $t_{latch} + t_{xor}$

Forward latency (Boolean): $t_{xnor} + t_{nor} + t_{latch} + t_{xor}$

Reverse latency (data): $t_{xor} + t_{nor} + t_{latch}$

Reverse latency (Boolean): $t_{xor} + t_{nor} + t_{latch} + t_{xor}$

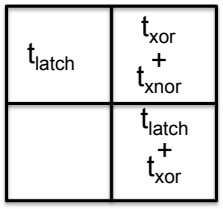


Figure 7.16: Synchronization point model for the data path of the conditional select

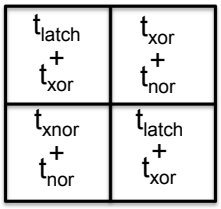


Figure 7.17: Synchronization point model for the Boolean path of the conditional select

Using the synchronization-point notation of Section 3.2 gives the values of Figure 7.16 for the data path and Figure 7.17 gives the values for the Boolean path. In particular, the synchronization point–the point at which an incoming request synchronizes with an incoming acknowledge–is at each of the latches. The Boolean value entering the stage goes through an *xnor* and a *nor* gate before reaching this synchronization point, while the incoming request for req_0 and req_1 arrive immediately at the latches. These values of Figure 7.16 can be used in the analysis and optimization techniques presented in this thesis.

Timing Constraints for Testing

The setup time constraint for the data path is identical to that of a regular MOUSETRAP stage, because no additional stages stand between the incoming request and the latch where the signals synchronize.

The hold time constraint is more restrictive than for a regular MOUSETRAP stage given in Section6.3.2, since the path for closing the latch contains additional gates.

$$t_{\mathrm{hold}_N} + t_{xor} + t_{xor} + t_{nor} \; < t_{xnor} + t_{latch} + t_{logic}$$

7.5.3 Options and optimizations

Logic Minimization

The logic to open and close the latches—shown in Figure 7.15 using an xor, an xnor, and a nor—can also be implemented using a two-level sum-of-products form to reduce delays.

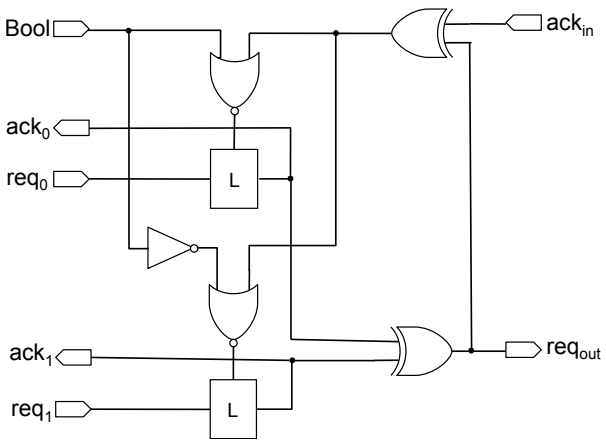


Figure 7.18: Conditional select optimized for constant Boolean value

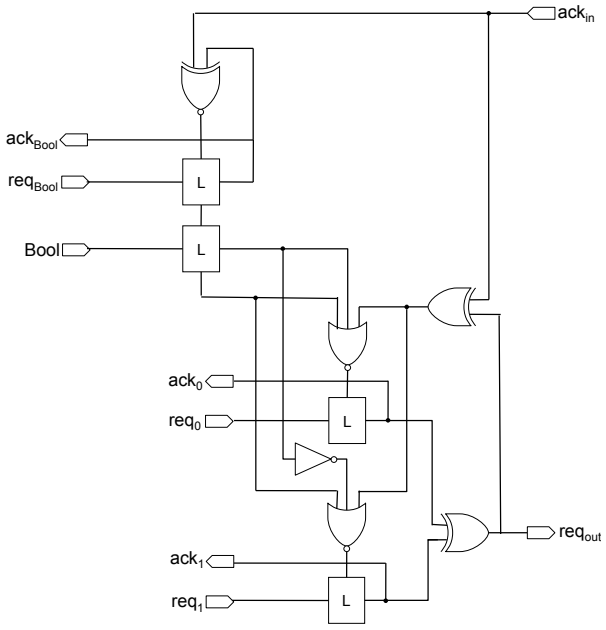


Figure 7.19: Conditional select optimized for forward latency with the assumption that the Boolean value arrives before the data.

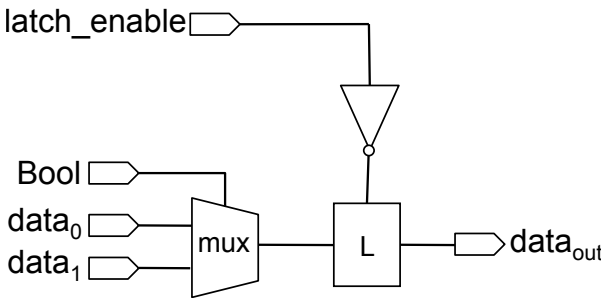


Figure 7.20: Data-path setup for a conditional select

Global Boolean Value

In some systems, the Boolean value may be a global constant that is assumed to be stable, and will therefore not have any associated request. In this case, the circuit is simpler (one xnor gate is removed), as shown in Figure 7.18.

Early Acknowledge for Boolean

The circuit can also be changed to produce the acknowledgment for the Boolean value early, before the data has arrived. The modified circuit, seen in Figure 7.19, latches the Boolean value to use when the data later arrives. This version has a higher forward latency in cases where the Boolean value arrives after the data, and should only be used when the designer knows that the Boolean value will always be available before the data.

7.5.4 Data path

Figure 7.20 below shows one bit of the data path for the conditional select. This data path implementation can be used with either the optimized or non-optimized circuit implementations. The operation of the data path is as follows. First, the Boolean value selects which incoming data to use. When both the data and Boolean requests are present, the latch enable signal makes data latch opaque. This protects the data from being overwritten by new incoming data.

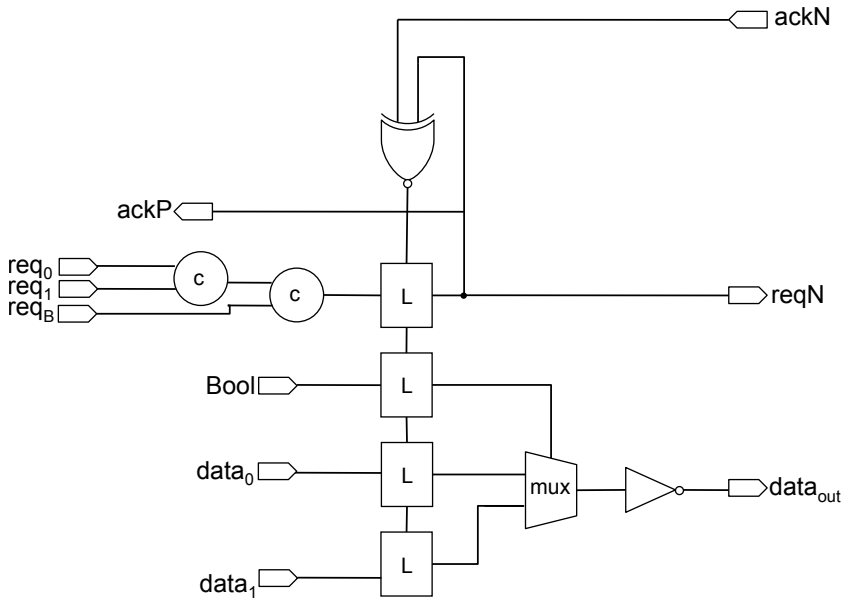


Figure 7.21: Conditional join circuit diagram

7.6 Conditional Join

7.6.1 Behavior

A conditional join is similar to a conditional select, except that all input channels are read even though data from only one of them is forwarded; data from the remaining input channels is discarded. Thus, the handshake behavior is identical to a simple 3-way pipeline join stage. The datapath operation is identical to a combinational multiplexor. It waits for the Boolean input and all data inputs to be ready and acknowledges all inputs once the data is latched. The combinational logic after the latches multiplexes the data based on the Boolean input, and the unused data item is simply discarded.

7.6.2 Gate level implementation

Figure 7.21 shows an implementation of the conditional join. C-elements combine all incoming requests into one request. Combinational logic consists of a multiplexor.

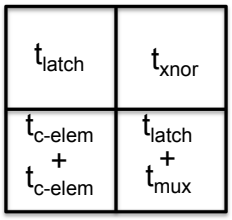


Figure 7.22: Conditional join abstract analysis model

Performance Metrics

Forward latency: $t_{c\text{-}elem} + t_{c\text{-}elem} + t_{latch}$

Reverse latency: $t_{xnor} + t_{latch}$

Using the synchronization-point notation of Section 3.2 gives the values of Figure 7.22. These values can be used in the analysis and optimization techniques presented in this thesis.

Timing Constraints for Testing

The setup time constraint is less restrictive than that of a typical MOUSETRAP stage, as given in Section 6.3.2, because the extra two c-element delays decrease the risk of a timing violation.

$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{\mathrm{c\text{-}elem}_N} + t_{\mathrm{c\text{-}elem}_N} + t_{\mathrm{Latch}_N} + t_{\mathrm{XNOR}_N\downarrow}$

The hold time constraint is identical to that of a regular MOUSETRAP stage, since the path closing the latch to protect the data is identical.

7.6.3 Options and Optimizations

The selection of data values can also take place before the latches. This reduces the total number of latches, but also creates the timing assumption that the data will arrive in time to be selected before the latches become opaque.

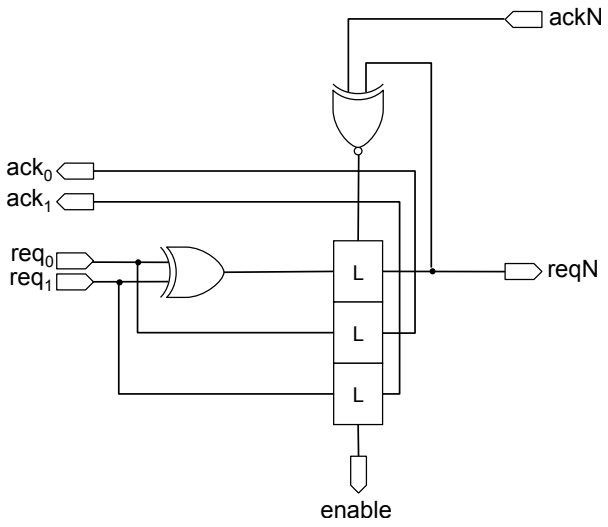


Figure 7.23: Circuit diagram for merge without arbitration

7.7 Merge without Arbitration

7.7.1 Behavior

This pipeline stage has two input channels and one output channel. Data is read from whichever input channel has a new incoming request and then sent to the output. No arbitration is provided; it is assumed that the input channels are mutually exclusive. An incoming request on either $req0$ or $req1$ will trigger a toggle on the outgoing request line. This toggle will quickly make the latches opaque, so any new incoming request will not be processed until the acknowledge from the succeeding stage has been received.

7.7.2 Gate level implementation

Figure 7.23 shows an implementation of a merge without arbitration. An xor is used to combine the two incoming requests, such that a toggle on exactly one incoming request line will lead to a toggle on the output request. The latch control works similarly to a normal mousetrap stage.

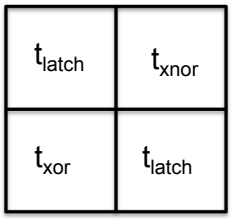


Figure 7.24: Merge without arbitration abstract analysis model

Performance Metrics

Forward latency: $t_{xor} + t_{latch}$

Reverse latency: $t_{xnor} + t_{latch}$

Using the synchronization-point notation of Section 3.2 gives the values of Figure 7.11. These values can be used in the analysis and optimization techniques presented in this thesis.

Timing Constraints for Testing

The setup time constraint is less restrictive than that of a typical MOUSETRAP stage because the extra xor delay decreases the risk of a signal arriving to the latch before the circuit is ready.

$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{xor_N} + t_{\mathrm{Latch}_N} + t_{\mathrm{XNOR}_N\downarrow}$

The hold time constraint remains identical to the one for a typical MOUSETRAP stage, since the path for closing the latch is identical to that of the original stage design.

7.7.3 Datapath

The datapath used with the merge without arbitration depends on the previous stages used in the system. Often, an arbitration stage just before the merge stage will perform the merging of the datapath. In this case, the data can simply be latched with the signal *enable* shown in Figure 7.23.

If the datapath has not already been merged, however, the two incoming datapaths are multiplexed to give one output data value. Two possible datapaths that are capable of merging incoming data are shown in Figures 7.25 and 7.26 respectively. The circuit of Figure 7.25

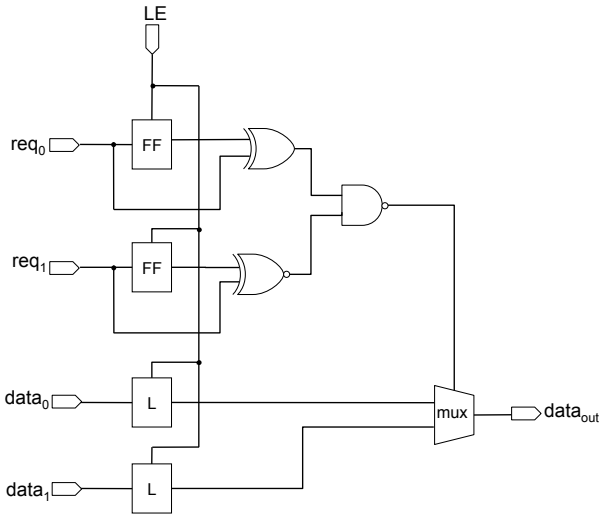


Figure 7.25: Implementation of the data path for a merge

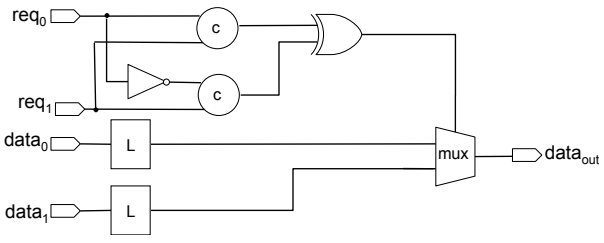


Figure 7.26: Implementation of the data path for a merge using C-elements

uses flip-flops to hold the state of the outgoing request while the circuit of Figure 7.26 uses c-elements.

7.8 Arbitration Stage

7.8.1 Behavior

This pipeline stage performs arbitration between two input channels, and produces results on two output channels. Only one input channel is read at any time, and its value is sent to its corresponding output channel. This circuit is essentially a 2-phase wrapper around a mutex element, which is a circuit that can reliably break ties between two incoming signals. This implementation of a two-phase arbiter is adapted from Erik Brunvand's thesis [7]. It allows the earlier request that arrives on either channel to pass through, and ignores subsequent requests until the current handshake cycle is complete. Based on which incoming request is received first, it sends a request out on one of the two request lines.

7.8.2 Gate level implementation

The first set of latches begins transparent, while the second set begins opaque. When a request arrives, the first latch becomes opaque, the second becomes transparent, and the mutex to be unresponsive to new incoming requests. The ack returning from the next stage re-enables the mutex so more incoming requests can be processed.

Performance Metrics

Forward latency: $t_{latch} + t_{xor} + t_{mutex} + t_{latch}$

Reverse latency: $t_{xnor} + t_{mutex} + t_{latch}$

Using the synchronization-point notation of Section 3.2 gives the values of Figure 7.28. The synchronization point of this arbitration stage is the second latch that the request passes

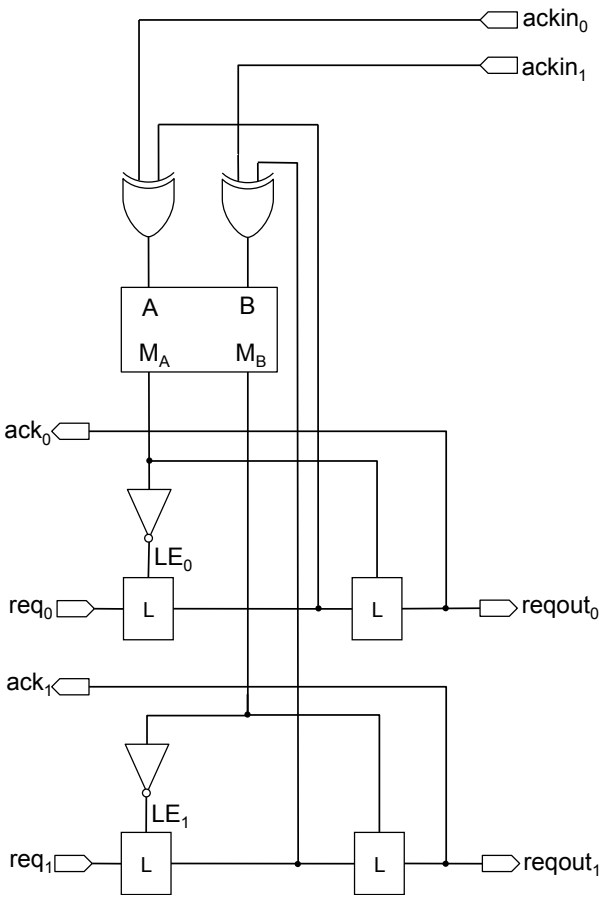


Figure 7.27: Arbitration stage built around a mutex

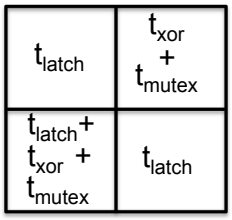


Figure 7.28: Arbiter abstract analysis model

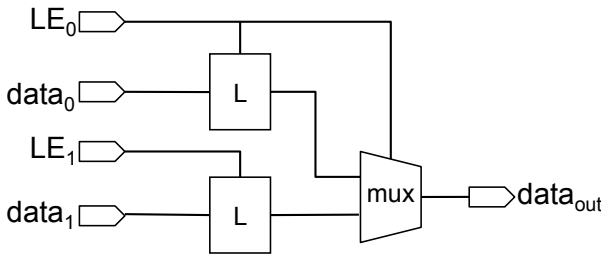


Figure 7.29: Datapath for arbitration stage

through, so all delays in Figure 7.28 are relative to that point.

Timing Constraints for Testing

The setup time constraint is less restrictive than that of a typical MOUSETRAP stage, and also more complicated because the path to shut the latch includes an additional mutex element.

$$t_{\mathrm{data}_{N-1}} + t_{\mathrm{setup}_N} < t_{\mathrm{req}_{N-1}} + t_{xnor} + t_{mutex}$$

The hold time constraint is more restrictive for the arbitration stage than it is for other MOUSETRAP stages. Specifically, because the reset path includes a latch in addition to an xor, the time between data arriving and the latches becoming opaque to protect the data is higher than in a regular MOUSETRAP stage.

$$t_{\mathrm{XNOR}_N\downarrow} + t_{mux} + t_{\mathrm{hold}_N} < t_{\mathrm{XNOR}_{N-1}\uparrow} + t_{\mathrm{Latch}_{N-1}} + t_{\mathrm{logic}_{N-1}}$$

7.8.3 Datapath

Depending on the next stages used in the system, the arbitration stage may either maintain two separate data paths and requests, or it may merge the data paths. If two separate data paths are required, then the data latches on paths 0 and 1 will use the latch enable signals LE0

and LE1 respectively. Merging of the two data paths can be accomplished using the merge without arbitration stage described in Section 7.7.

7.9 Conclusion

This section presented a set of five new pipeline circuits that, together with existing MOUSETRAP circuits [55], can form any of the hierarchical components handled by the analysis and optimization methods of Chapters 4 and 5. For each component, the section gave performance metrics, which can be used as inputs to the analysis method. If new types of connections or constructs are eventually added to the analysis approach, then new stages in the same style may need to be added as well.

Additionally, each stage is has information about its timing constraints listed, to aide in testing for timing constraint violations. Although the testing methods of Chapter 6 have not been extended specifically to these new stages, their timing constraints are in line with the kinds of constrains that it can test.

Chapter 8

Case Study

8.1 GCD Circuit Design

8.1.1 Introduction

This section presents the design of a greatest common divisor (GCD) chip as a case study to investigate some of the potential benefits of pipelined asynchronous design. We are currently witnessing a trend where mobile applications increasingly require chips that can deliver high performance when needed, but that are ideally also capable of operating robustly on scaled-down supply voltages when battery life is at a premium [48]. While researchers believe that asynchronous technology can deliver chips to satisfy this need, very few concrete demonstrations of high-performance yet robust asynchronous designs exist [12]. The goal of the GCD case study was to design, fabricate, test and evaluate a medium-complexity chip, and to report whether the goals of high performance, voltage and temperature robustness, and testability can be achieved while expending reasonably low designer effort.

The GCD chip implements a modified version of Euclid's iterative algorithm. A total of 98 self-timed stages, connected into the shape of a ring, implement the iterative algorithm. Each trip through the 98 stages represents eight algorithmic iterations (*i.e.,* an eight-way unrolled loop). This level of fine-grain pipelining results in a fairly high performance, equivalent to a 1 GHz clocked ASIC. However, unlike a synchronous implementation, the asynchronous

nature of this design results in good robustness to temperature and voltage variation.

The design was fabricated in a 0.13μm CMOS process, using standard cells and with full testability support. Since the design is clockless, its performance is measured and reported in terms of the number of GCD algorithmic iterations completed per second. A total of 24 parts were received and all were tested. At nominal temperature and voltage (1.5V and 27°C), the fabricated chips complete about 8 giga algorithm iterations per second (equivalent to 1 GHz clock speed). Testing was performed over a wide range of supply voltages (0.5–4.0V) and temperatures (-45–150°C), and all of the parts were found to be functionally correct. The asynchronous nature of the implementation allows the performance to gracefully vary with changing operating conditions, without the need for clock re-calibration. For example, at nominal temperature and 0.7V, which is less than half of the nominal voltage, the performance slows down to 1.4 giga iterations per second, but the chip still remains functionally correct. At a temperature of 150°C but nominal voltage, the performance slows to 6.7 giga iterations per second, though functional correctness is still maintained. Finally, the implementation required a fairly modest designer effort—the entire physical design entailed 5 person-weeks of effort—which likely would not have been possible but for the inherent modularity of asynchronous components and the timing robustness of their interaction.

8.1.2 Overview of Design

This section presents an overview of the GCD chip's implementation, including the high-level algorithmic specification, the overall architecture, and brief details on its pipelined implementation.

Figure 8.1 shows the overall architecture of the core of the GCD chip, along with a picture of its layout. The design is organized as a self-timed ring, which iteratively computes the GCD function, and a special interface which allows the design to communicate with, and to be controlled by, its external environment.

Figure 8.2(a) shows the basic GCD algorithm that was used as the starting point for our

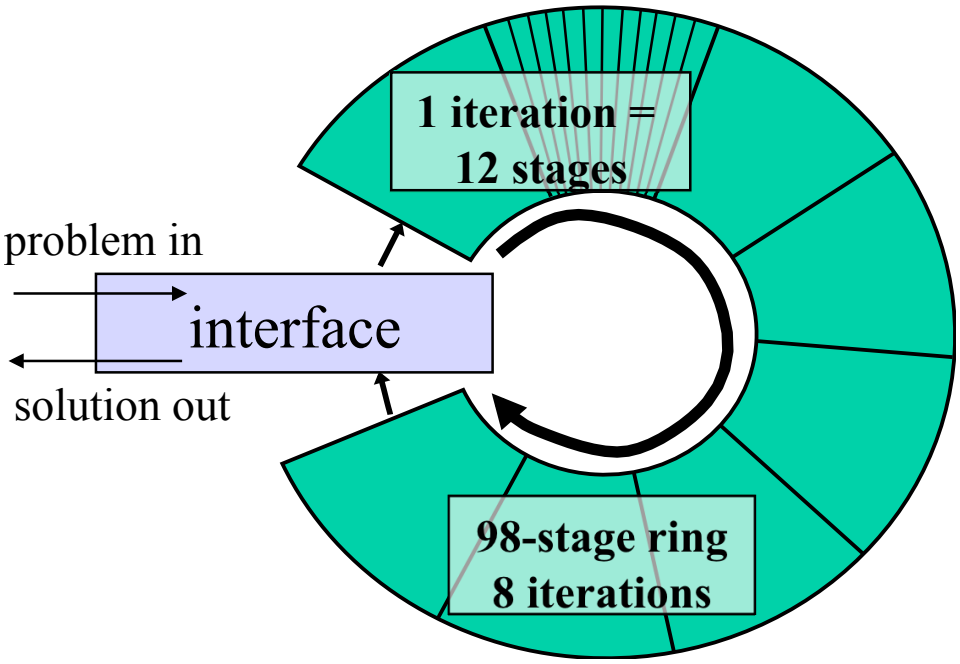


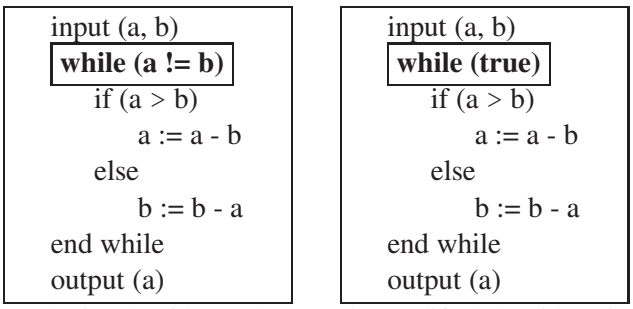
Figure 8.1: Overall architecture of GCD chip

implementation. The result is computed by repeatedly subtracting the smaller number from the bigger one until both numbers are equal. The end value is then the result of the computation. Several modifications were made to this basic algorithm to reduce its implementation complexity. These modifications will be introduced in the following sections.

The operand width was chosen to be 8 bits for this implementation. However, the GCD algorithm, the overall ring structure, and the interface design presented here actually apply to operands of any length. The motivation for choosing 8-bit operands was to keep design effort reasonable, since all placement and routing was performed manually. The choice of 8-bit operands, however, results in a non-trivial datapath—24 bits wide—since two operands and one intermediate result are transfered between successive stages.

Ring Architecture: Loop Unrolling

The algorithmic loop was unrolled eight times before implementation. That is, the datapath fragment implementing one algorithmic iteration was replicated eight times within the ring. Therefore, when a data item completes one revolution around the ring, it effectively executes



(a) basic algorithm: data-dependent iteration count

(b) modified algorithm: infinite looping

Figure 8.2: Basic GCD algorithm before and after simplification of the ring interface

eight iterations of the GCD algorithm's `while` loop.

There were two objectives of unrolling the algorithmic loop. First, unrolling allows a greater number of distinct problems to be concurrently processed inside the ring, thereby achieving an approximately 8-fold increase in overall work throughput. Second, replicating the datapath of the ring allows the latency overhead of the interface to be amortized over a large number of stages. That is, the slopes of the forward and reverse lines of the canopy graph will not be much affected by the presence of the extra logic in the interface stage.

While loop unrolling techniques generally require careful management of the count of iterations executed, the GCD algorithm is easily unrolled because it is unaffected by extra iterations being performed. Once integer b becomes zero (see Figure 8.2(a)), the integer a will maintain its value, regardless of how many extra iterations are executed.

Finally, the datapath of the entire ring was pipelined into a total of 98 stages, each implemented using the MOUSETRAP style. Specifically, the datapath corresponding to each of the eight iterations was divided into 12 pipeline stages. Two additional "corner turner" pipeline stages perform routing of long datapath wires around the corners, and contain no additional processing logic. Given that at least one hole is required for data to be able to move inside the ring, a total of as many as 97 problem instances could be loaded into the ring for concurrent execution. The high stage count also allows the interface's latency to be amortized, allowing an estimation of the internal stages' latency to within 1% accuracy.

Interface: Infinite Looping

The interface serves three functions: (i) ring initialization, (ii) testing for faults, and (iii) loading of problems (*i.e.,* operand pairs) into the ring, and draining of results from the ring after computation.

An interface that exactly implements the behavior of the GCD algorithm shown in Figure 8.2(a) should be capable of dynamically managing the entry and exit of items into and out of the ring. In addition, to achieve optimal throughput, the interface should maintain ideal occupancy in the ring, *i.e.* neither under-utilized nor congested. The recent loop pipelining approach [16] presents an interface design that implements this behavior precisely.

This work, however, opted for an interface that allows data items to continue to loop around the ring until the operation is interrupted by the external environment. This interface effectively modifies the original algorithm to contain an infinite loop, as shown in Figure 8.2(b). As discussed above, since the GCD algorithm is unaffected by extra iterations, infinite looping does not affect the value of the results.

There were two motivations for using the simpler interface. First, it allows greater controllability of the chip from the external environment, thereby simplifying the task of testing the design for faults. Second, speed measurements can be made more easily using low-speed external test equipment because the ring operates continuously because of the infinite loop. Otherwise, each computation would be short-lived—on the order of tens of nanoseconds—which would require complex on-chip circuitry for evaluation of the chip's performance. By turning the computation into an infinite loop, we enable steady-state observations of the chip's performance, and leverage existing work on analysis of self-timed rings to estimate internal latencies and cycle times [68].

External Environment

The test environment must meet two requirements in order for it to interact correctly with our chip's interface. First, it must allow sufficient time for computations to be completed before

draining the results. This requirement was easy to meet because our test environment was several orders of magnitude slower than the GCD chip.

Second, the environment must be capable of handling results which emerge out-of-order. In particular, since the ring is cut open for draining results at an arbitrary time by the environment, the results drained are a cyclic permutation of the order in which problems were loaded. The correctness of results is easily checked by our testing software, which checks every possible cyclic permutation of the output data against the precomputed expected values.

8.1.3 Implementation of a Single Iteration

Datapath Design and Optimization

Figure 8.3 shows again the basic GCD algorithm of Figure 8.2(b) with infinite looping. The body of the iterative loop is shown highlighted. A corresponding straightforward dataflow implementation of the loop body is shown in Figure 8.3. Both $(a-b)$ and $(b-a)$ are computed using two separate subtract units, and then the values of a and b are updated appropriately depending upon the test $(a > b)$.

The basic algorithm and dataflow implementation of Figure 8.3 have significant area overheads. In particular, assuming no resource sharing, this algorithm requires the use of two distinct subtract blocks because subtraction is performed in two different steps of the algorithm. Further, the comparison test $(a > b)$ is an expensive operation, whether it is implemented directly as a comparator block, or as a subtract operation followed by a sign check.

The basic algorithm was modified to improve its area, as shown in Figure 8.4. The new version of the algorithm requires only a single subtract operation. The second subtract operation was eliminated through use of a relatively inexpensive swap operation: if $a < b$, the current iteration swaps a and b, letting the next iteration performs the correct subtract operation. While this modification may increase the number of total iterations needed per computation, and hence lengthen the latency, the key goal in this work was high area efficiency and high throughput. Furthermore, the expensive comparison operation $(a > b)$, which required

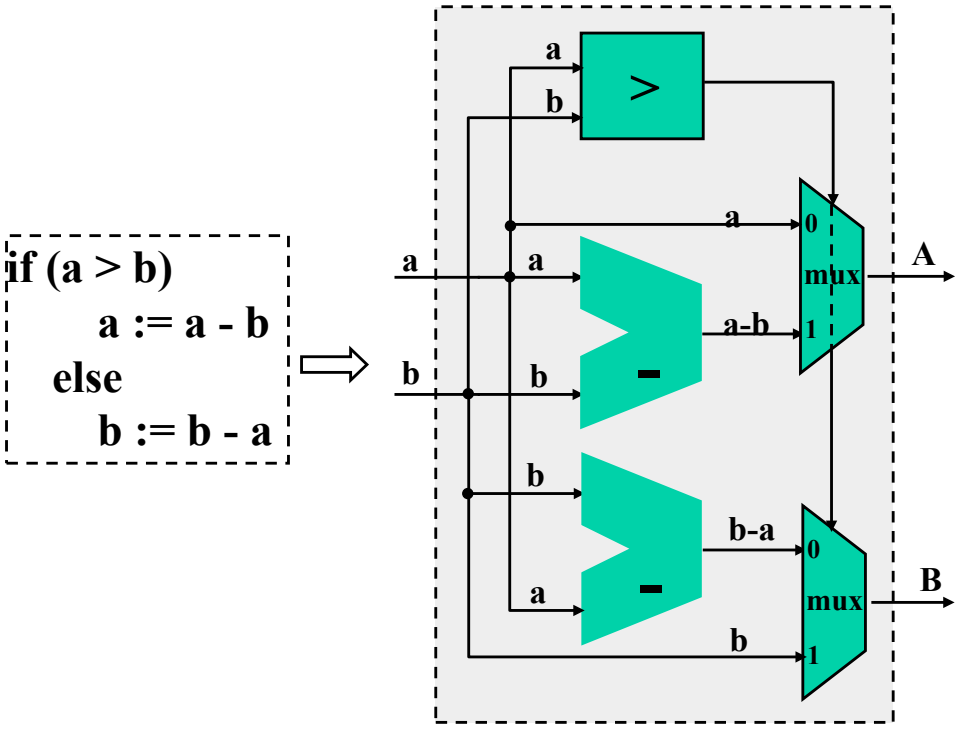


Figure 8.3: Basic GCD algorithm, and the corresponding dataflow implementation of one iteration

the examination of all of the operand bits, is now replaced by the significantly simpler test $(s < 0)$, which requires only the examination of the sign bit. The net result is the much more area-efficient implementation of Figure 8.4.

Pipelining an Iteration

The datapath corresponding to one algorithmic iteration, shown in Figure 8.4, was pipelined into 12 stages. In particular, the subtract block was pipelined into 8 stages, while the conditional swap operation, implemented by the multiplexers, was pipelined into 4 stages. Figure 8.5 shows the layout of one iteration.

Subtract. A simple pipelined ripple-borrow implementation was used for the subtract block. This implementation is identical in architecture to a ripple-carry adder [23], but uses slightly different logic in each stage: instead of a full adder, a "full subtractor" block is used. If X and

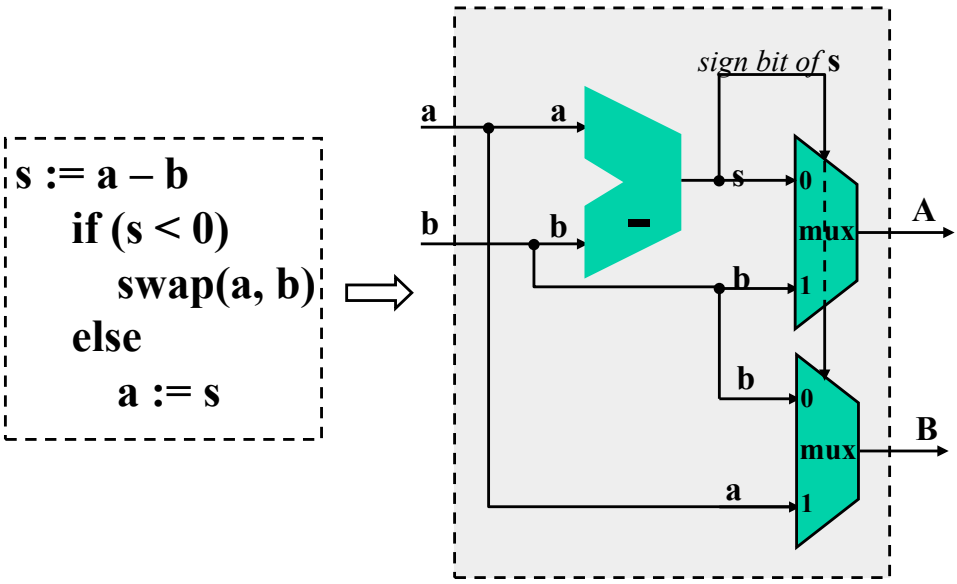


Figure 8.4: Area-optimized GCD algorithm, and the dataflow implementation of one iteration

Y are the operand bits at a given stage and B_{in} is the input borrow, then the result D and the borrow output B_{out} generated at this stage are given by:

$$D = X \oplus Y \oplus B_{in} \tag{8.1}$$

$$B_{out} = \overline{X} \cdot (Y + B_{in}) + Y \cdot B_{in} \tag{8.2}$$

Since the operands in this implementation are eight bits in length, the subtract block was implemented using eight pipeline stages. The left portion of Figure 8.5 shows the layout of the subtract block. By using a simple rotation of the datapath wires as they go from one stage to the next, it was possible to simply tile the layout of one subtract stage eight times to obtain a regular layout pattern.

Conditional Swap. The swap operation, implemented using multiplexers, was pipelined into four stages. While multiplexing of single bits is easily accomplished within a single pipeline stage, this design requires two 16x8 multiplexers, which represent too large a load to be efficiently implemented as a single pipeline stage. In addition, the swap operation required

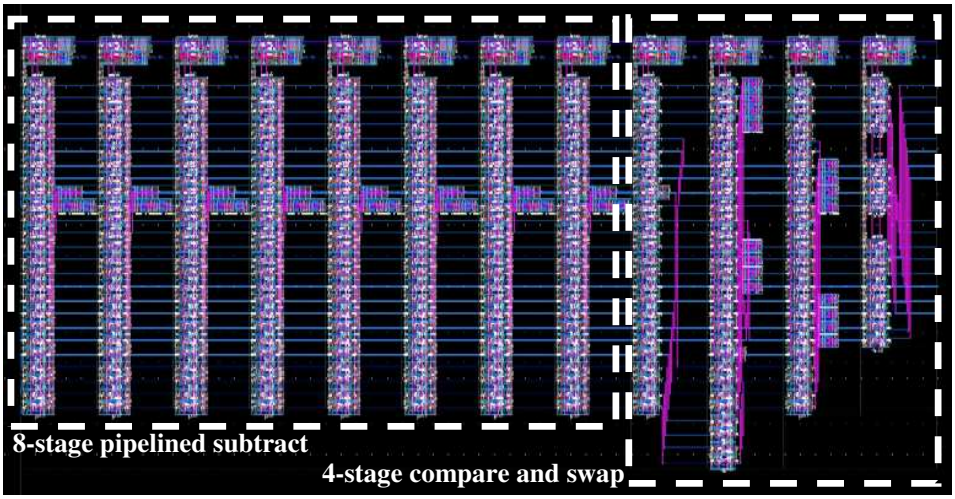


Figure 8.5: Layout of one iteration

a significant amount of routing and rearrangment of the data bits so that the ordering of bits at the output of an iteration matched the input order for the next, thereby further increasing loading. Therefore, in order to keep the loading on the datapath manageable, the swap operation was spread across multiple pipeline stages. In particular, the first and the last stages were functionally FIFO stages, but performed a significant portion of the routing and rearrangement of data bits required in the design. The middle two stages performed the actual multiplexing operations.

Stage Implementation: Details

This section presents details of the implementation of each pipeline stage. In addition, the discussion highlights certain design decisions that were made to ensure robustness to timing variations.

Figure 8.6 shows a gate-level implementation of each pipeline stage, including buffers that were added for robustness. The logic will vary between stages, but the other gates do not.

Datapath. The datapath consists of standard level-sensitive D-type transparent latches and static logic function blocks (either subtract logic or swap logic). These were implemented using gates from a standard-cell library.

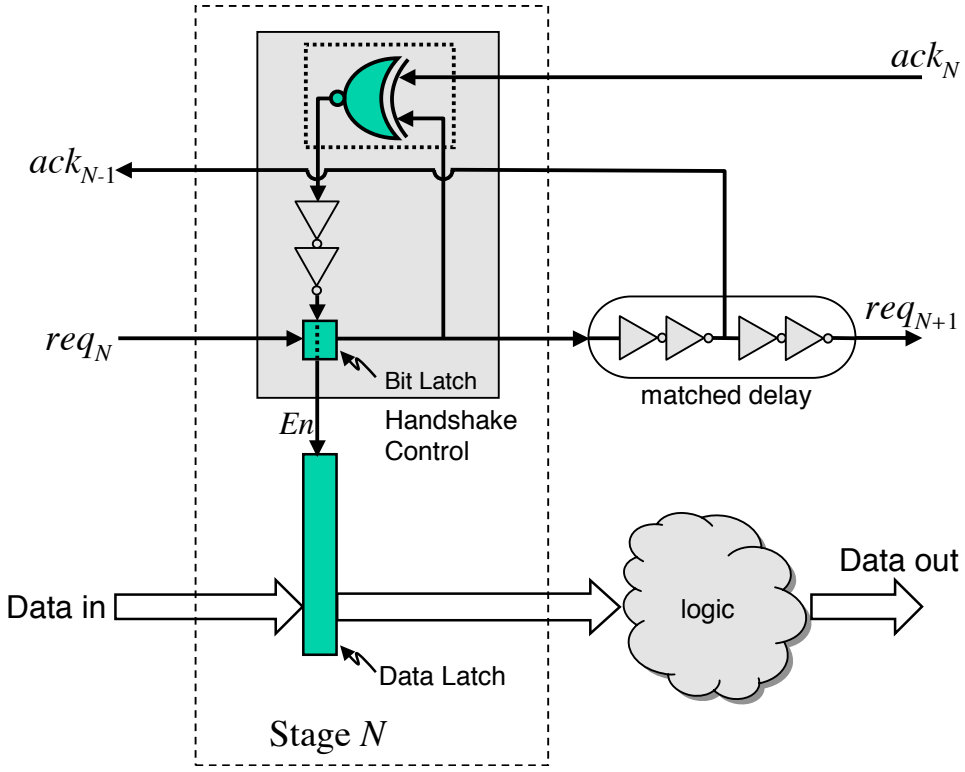


Figure 8.6: Implementation of a stage: delay matching and control buffering

Control. The control logic consists of the XNOR gate, its output buffers, and a matched delay, all of which were implemented using standard gates. The controller's output was amplified using two series inverters in order to provide adequate strength to drive the entire datapath, whose width was 24 bits (8 bits each for a, b and s). A matched delay consisting of four series inverters was used which, for all stages, adequately matched the worst-case delay through the function logic.

Robustness Considerations. Two design decisions were made in order to ensure high robustness to timing variations.

First, buffer insertion did not employ the technique of *control kiting* [39, 72]. In kiting, unbuffered control signals are used for producing the handshake signals, and the control is buffered only to drive the datapath. Thus, when kiting is used, the datapath operates with a skew with respect to the control, and the overhead of buffering is eliminated from the cycle time. However, unless the datapath skew is exactly the same in neighboring stages, kiting has

the disadvantage of reducing timing margins, thereby sacrificing robustness. Therefore, this work did not use kiting, opting instead for higher timing robustness.

Second, as shown in Figure 8.6, the acknowledgements were delayed by two inverter delays before being sent to the previous stage. This delay was added in order to provide a healthier timing margin to the hold time constraint. In particular, this approach gives the current stage additional time to make its latch opaque before the acknowledgement stimulates the previous stage to generate the next data item.

Performance Impact. The cost of higher timing robustness was an increase in the cycle time by four inverter delays: two inverters due to control buffering, and another two due to the delayed acknowledgments.

8.1.4 Interface

The interface of the GCD chip performs several key tasks, including data insertion and removal. For test purposes, it allows full controllability and observability of the request, acknowledge, and data wires between the first and last ring stages. Finally, during steady-state operation, the interface provides at-speed measurement of the ring's throughput by allowing handshake activity between the first and last stages to be externally observed. To facilitate measurements using slower external equipment, a decimated version of the ring's frequency (divided by 64) is also provided.

Structure

The structure of the interface is shown in Figure 8.7. The interface connects the first ring stage, Stage 1, with the last ring stage, Stage 98, and also provides the external off-chip environment with full controllability and observability of this connection.

Specifically, the interface allows three pieces of information to be exchanged between the first and last ring stages: (i) request, *req,* from the last stage to the first; (ii) acknowledgment,

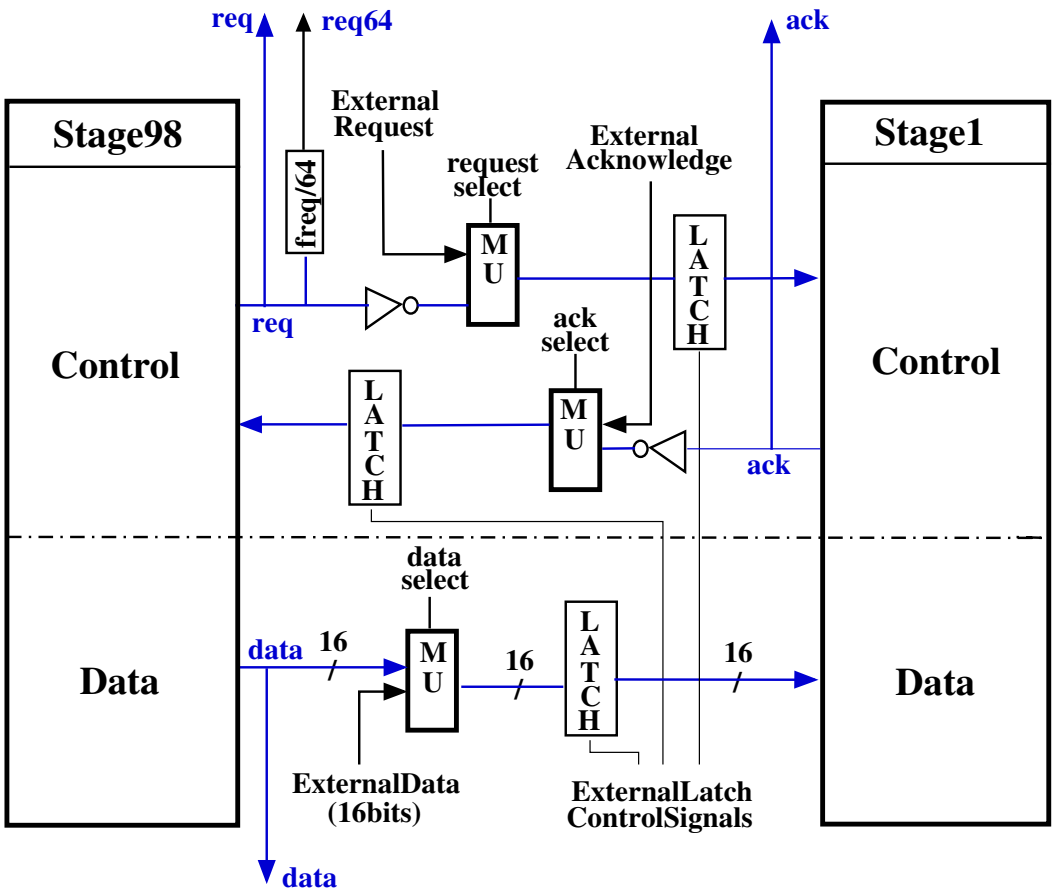


Figure 8.7: Detail of the chip interface

ack, from the first stage to the last; and (iii) data (16 bits) from the last stage to the first.

The interface allows the external environment to passively observe these three signals, as well as to actively control them. In particular, copies of these signals (shown highlighted in blue in Figure 8.7) are simply forked off to the environment for observability. In addition, the interface contains a frequency divider—consisting of a series of six edge-triggered flip-flops—that produces a frequency-decimated version of the request signal (frequency divided by 64), which facilitates observation of the high-speed operation on slower test equipment.

To provide the environment with full control, multiplexers and latches are added to the path of each signal. The latches are controlled by the environment, and allow the ring to be stopped or restarted under external control. The multiplexers, also controlled by the environment, allow externally generated request, acknowledgment and data values to be substituted

for the internal signals. Before the control input to any of the multiplexers is changed, the associated latch is temporarily disabled to ensure that no glitches generated by the multiplexers are transmitted.

Finally, the request and acknowledge paths contain one inverter each, to ensure correct polarity. These inversions are necessitated by transition signaling, when the ring is filled with an odd number of items. For instance, if the ring is filled with only a single item, it should toggle the request going into the first stage every time that item crosses the interface. On the other hand, inversions should not be performed if the ring is to be run with an even number of items. In this work, we chose to design the ring to be run with an odd number of items, so inverters were required.

Operation

The operation of the chip consists of the following four phases in sequence: (i) initialization, (ii) loading of problem instances, (iii) computation by the ring, and (iv) draining of results. Below, the role of the interface is described during each of these phases.

Initialization. An externally-supplied *reset* signal initializes all of the pipeline stages in the ring to be empty. In particular, all of the requests and acknowledgments are forced high, which in turn asserts all the latch enables, thereby making all latches transparent and the pipeline empty.

Loading of Problems. After initialization, the interface allows the external environment to insert problems into it. To do so, first all of the three latches in the interface are made opaque to cut the ring open and to prevent any glitches from propagating through the circuit. Next, the three multiplexers are set to select inputs from the environment. Finally, the latches are made transparent again to allow the inputs from the environment to enter the circuit. Each problem instance is loaded into the ring by first providing the appropriate data bits, and then toggling the value of the request to perform a handshake with the first stage of the ring.

Computation. Once all problems are loaded into the ring, the interface allows the environ-

ment to seal the ring close by connecting the first and last stages together, thereby allowing data to flow around the ring and computation to occur. Once again, to avoid glitches, the interface latches are first made opaque. Then the multiplexers are set to transmit signals between the first and last ring stages. Finally, the interface latches are made transparent again, effectively reconnecting the ring and allowing computation to occur.

Draining of Results. Once computation is completed, the results are drained under the control of the external environment. For reasons discussed in Section 8.1.2, the environment simply waits long enough to allow the results to be computed; there is no explicit checking of the computation's completion.

The draining of results takes place using the following sequence of steps. First, the latch on the acknowledge path is made opaque. This action stops any further acknowledgment from the first stage to reach the last stage, thereby freezing the ring's operation. When the ring stops, the data item stored in the last stage has been received by the first stage, but not acknowledged. Therefore, two copies of that data item exist, at the beginning and end of the ring, which is a situation that is checked by our test software.[1] After sufficient time has been allowed for the interface to stabilize, the environment disables the request and data latches as well. Next, the multiplexer on the acknowledge path is set to allow the environment to generate the acknowledgment for the last stage, and the acknowledge latch is made transparent. Finally, the environment drains data items from the ring by toggling the acknowledgment to perform successive handshakes with the last stage of the ring.

Metastability Issues

Our method of stopping the chip when it is running by cutting open the ring has some non-zero probability of causing a metastable condition. In particular, metastability may arise if the acknowledge from the first stage to the last one was in the process of transitioning just as the

[1] Alternatively, the ring can be stopped by freezing the request from the last stage to the first stage. This scenario does not duplicate the last data item.

latch on the acknowledge path became opaque under the environment's control.

Two metastable scenarios are possible. In most cases, metastability will lead to the acknowledge simply taking extra time to change to the correct value, but the change itself will be monotonic. In such a situation no circuit malfunction will occur, provided the interface allows sufficient time for the metastability to resolve before the environment drains data from the ring. This indeed is the case since the external environment is orders of magnitude slower than the chip. In rare cases, however, the metastable acknowledgment could theoretically toggle several times before stabilizing. In such a situation, one or more data items may be lost from the last stage of the ring. In order to estimate the likelihood of the latter harmful metastable scenario, we performed over 60,000 tests on our fabricated chip; none resulted in any loss of data.

Support for Testability

Our approach to testing the GCD chip is based on recent work on non-intrusive testing of both stuck-at faults [53] and timing violations, using the method described in Chapter 6. This section presents the role performed by the interface to support testing.

For stuck-at fault testing, the interface allows the environment to cut the ring open, initialize it empty, and then test the transparent datapath using standard combinational test methods.

For testing timing constraint violations, the interface again allows the environment adequate controllability to expose such violations and to functionally observe erroneous behavior. In particular, the key idea behind testing for timing violations is to create the situation that stresses the timing constraint using only low-speed test equipment. As described in Section 6.3, setup time violations are exposed when a data item is introduced into an empty pipeline; as it moves unhindered through the pipeline, it exposes setup violations through successive stages. Creating this situation requires the ability to send all of the data bits of the test pattern into the ring, in parallel with the associated request signal; the interface clearly allows this test scenario to be realized. Similarly, hold time violations are exposed when a data item

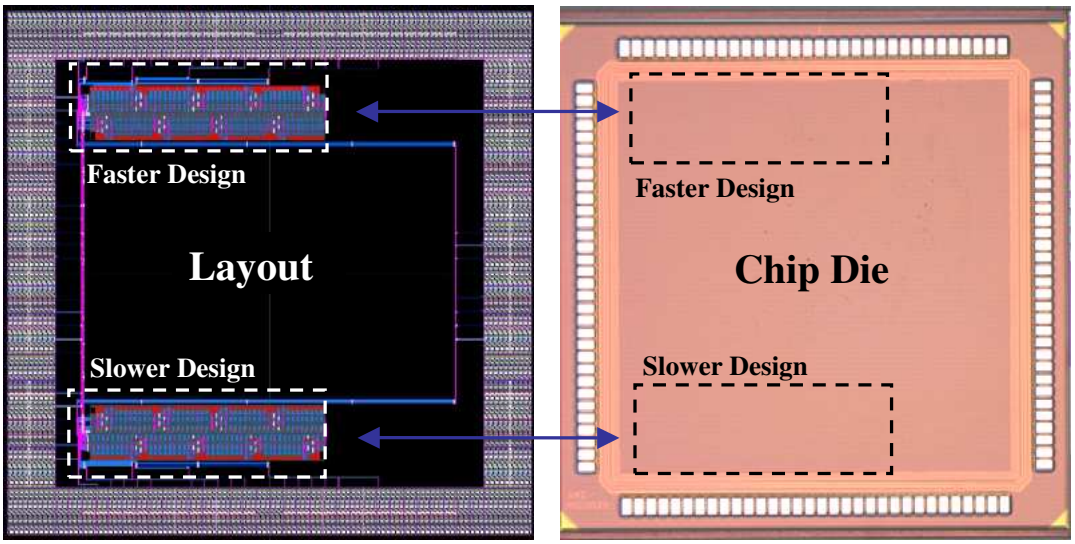


Figure 8.8: Fabricated chip layout and photomicrograph

is removed from an initially full pipeline. Once again, the interface allows the environment to create this scenario as well.

8.1.5 Testing Setup

Layout and Fabrication

Our chip was implemented in a 0.13μm process (IBM 8RF CMOS). It was designed using the Cadence tool suite, using a standard-cell library from the University of Washington.

Two versions of the GCD ring were fabricated on the same die: a slower version with conservative timing margins and a faster version with slightly tighter timing margins. Each version contains 50K transistors and occupies 1.5X.5mm^2. Figure 8.8 shows the layout of the completed chip and a photomicrograph of the fabricated die. Both versions were found fully functional during testing. Results for the faster version are presented here; the throughput of the slower version was about 15% lower.

Since the design was standard-cell, only coarse-grain gate sizing and delay matching could be performed. In particular, ideal gate sizes were computed using Logical Effort [60], and then standard gates were chosen from the library that most closely fit these calculations.

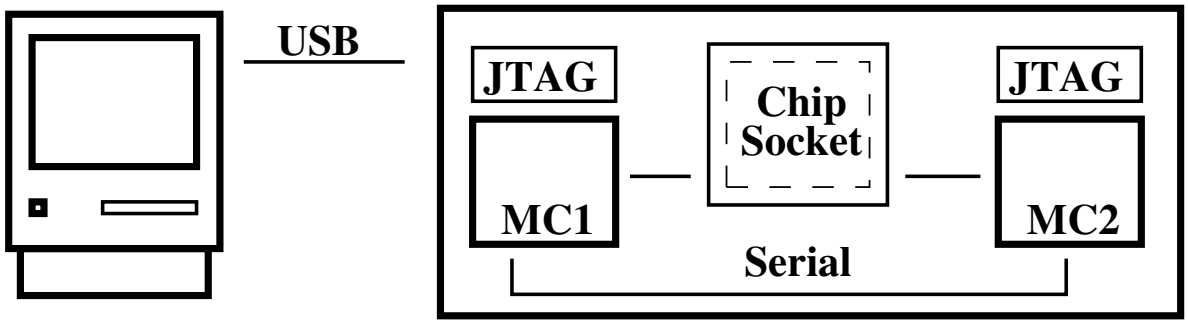


Figure 8.9: Testing and evaluation setup

SPICE simulations were performed to estimate timing; all timing constraints were satisfied with healthy margins.

Board Design and Operation

A custom test board, shown in Figure 8.9, has a socket where the GCD chip is inserted for testing. The chip interacts with two TI MSP430 microcontrollers: the left microcontroller manages data and control to the chip, and the right microcontroller reads results from the chip, including the operating frequency at the interface. The microcontrollers communicate with each other using a two-wire serial connection. The results are relayed to a PC via on-board JTAG headers.

The operation of the board is in turn managed on the PC using a custom application written in Java. The user inputs data on the PC and sends it to the left microcontroller through a USB connection. The microcontroller then initializes the GCD chip, loads data into the ring, and begins the ring operation. The operating throughput is measured by the right microcontroller and reported to the PC. Finally, the left microcontroller opens the ring and drain the results, which are sent to the PC for display and verification.

Test Operation

The operation of the chip consists of the following four phases in sequence: (i) initialization, (ii) loading of problem instances, (iii) computation by the ring, and (iv) draining of results.

During initialization, an externally-supplied *reset* signal initializes all of the pipeline stages in the ring to be empty. Next the environment loads a set number of problem instances by toggling the ingoing request signal while supplying the desired test patterns.

The environment begins the computation phase by setting the multiplexers in the interface to transmit signals between the first and last ring stages. During the computation phase, the external environment can observe the request and the toggling at the interface of the chip. Since the ring was designed to operate in the gigahertz range, it also includes divide-by-64 circuitry that produces a signal slow enough for our testing equipment to read.

Finally, the environment drains the results. First it stops the computation by making the interface latches opaque so no requests can cycle through the ring. Then it toggles the acknowledgement signal to the last pipeline stage while reading out each resulting dataset one by one.

Fault Testing

The test patterns generated for this circuit according to the approach of [53] gave over 98% fault coverage. For testing violations of the two timing constraints of MOUSETRAP, a custom tool implementing the the non-intrusive functional test presented in Chapter 6 was used. Test patterns generated achieved 100% coverage for hold time violations and about 33% coverage for setup time violations, yielding an overall 66% fault coverage.

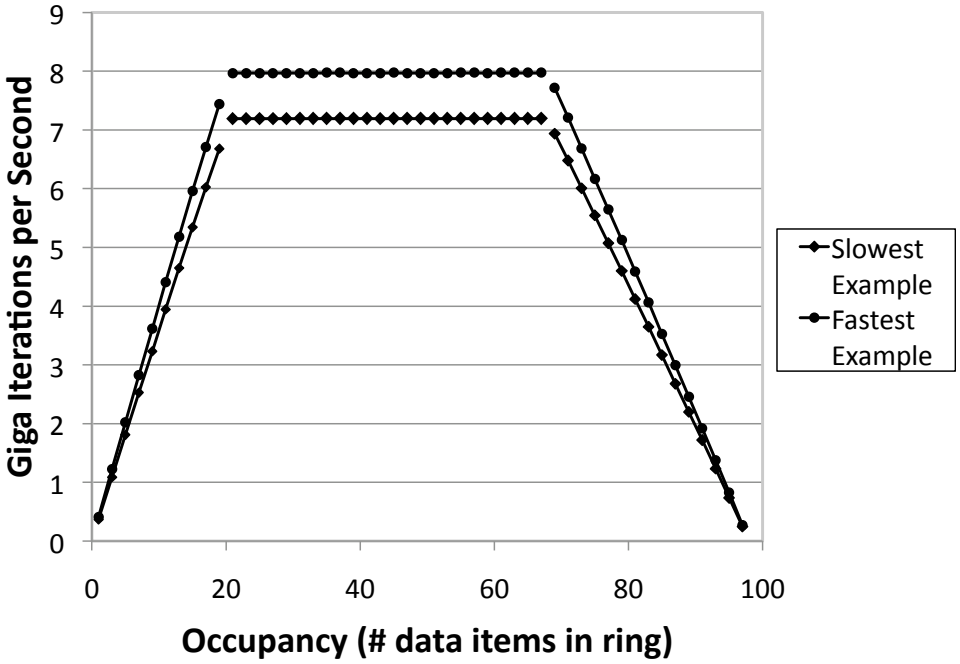


Figure 8.10: Throughput vs. occupancy for the fastest and slowest chips tested

8.1.6 Results and Discussion

Performance and Correctness

A total of 24 chip samples were tested, and all were found fully functional, with no faults detected. Figure 8.10 presents the throughput measured at various ring occupancies. The results for the fastest and slowest chips tested are shown. The fastest chip reported 1.01 billion toggles per second at the observable interface to the chip, which indicates that the chip was completing about 8.08 billion GCD iterations each second, while the slowest chip was about 10% slower.

Note that the shape of the graph of Figure 8.10 agrees with the expected performance of a pipelined ring, as discussed in Section 2.2.2. Although individual stages within the chip are not directly observable, their forward and reverse latencies can still be estimated. In particular, the total forward latency is inversely proportional to the slope of the line in the data-limited region, and the total reverse latency is inversely proportional to the slope of the line in the hole-limited region. The forward latency of an individual stage is therefore estimated to be

170ps and the reverse latency 300ps. Together, this gives a cycle time of about 470ps.

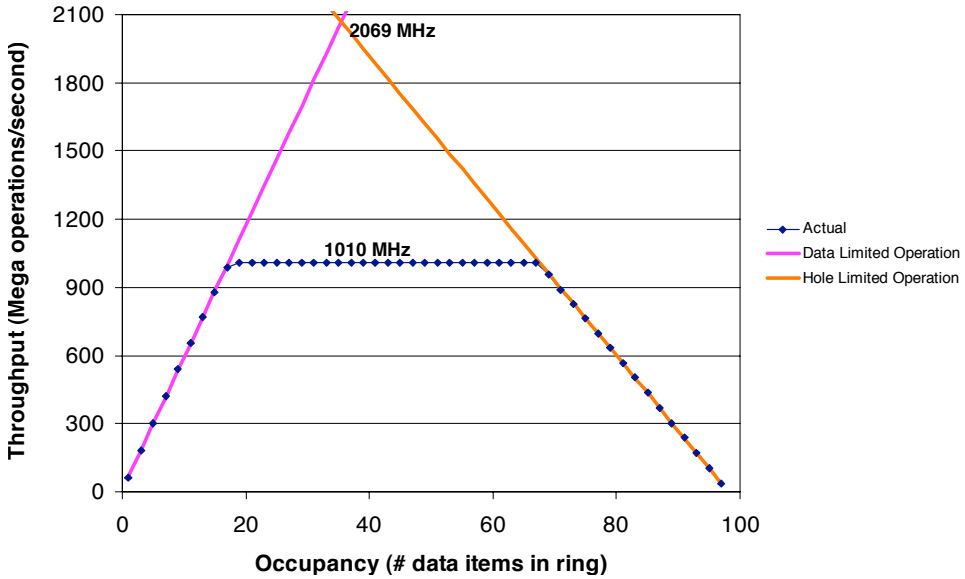


Figure 8.11: Estimating the ideal peak internal throughput

This indicates that each individual stages is capable of switching over 2 billion times per second. If the chip could realize this in practice, the switching of each stage would be 2GOPS and the chip as a whole would be capable of completing over 16 billion GCD iterations each second. This throughput is not realized in practice, however, because the ring's testing interface, which is substantially slower than the ring's internal stages, acts as a bottleneck If the interface were not a bottleneck, a careful analysis of the measured data, supported by thorough SPICE simulations, shows that a throughput quite close to 2.1 GOps/sec will indeed have been achieved. This is illustrated in Figure 8.11; for the fastest chip tested, the forward and reverse lines intersect where the switching activity of a single stage is at a rate of 2069MHz.

Though it is perhaps counterintuitive that the forward path is quicker than the reverse path—because the forward path contains computational logic while the reverse path does not—these delays are in accordance with our estimates of the forward and reverse latencies of a MOUSETRAP pipeline. In particular, the forward path contains one latch and the computation logic. For this system, the complexity of the logic is about the same as the latch itself. Since

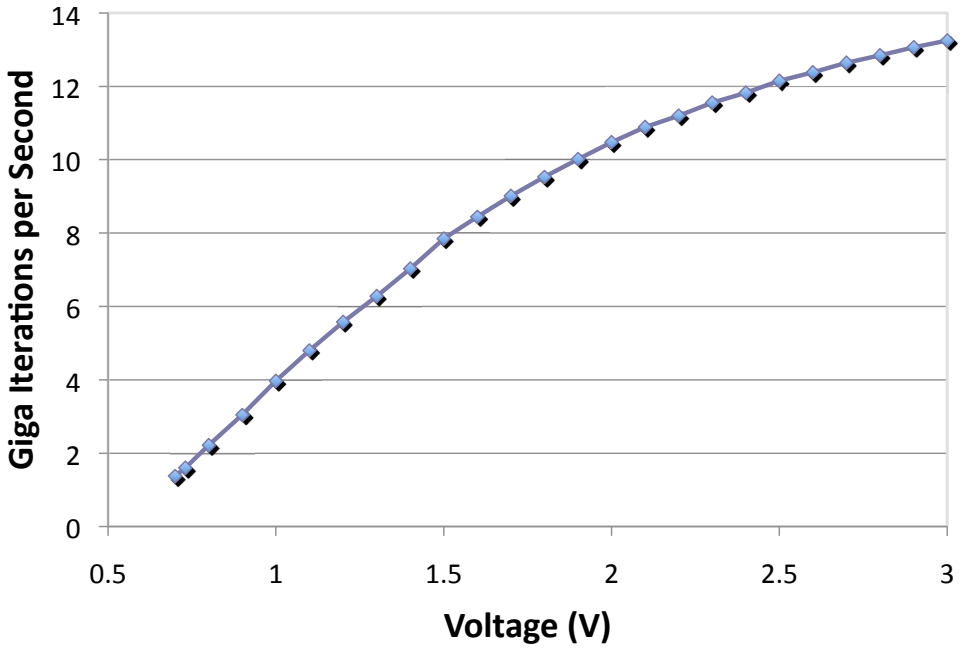


Figure 8.12: Speed increases with increased voltage

the latches and logic each drive only *one* other component that is of about the same complexity, their delays are very low, they can be sized to be very small, and they consume little power. On the other hand, in the reverse path, the controller must drive all of the latches in the datapath. To improve the drive strength and reduce step-up, buffers are added between the *xnor* controller and the latches. In addition, the reverse path has additional buffers on the acknowledge path to improve robustness. The final result is a reverse path that has a greater delay than the forward path.

Robustness Tests

In addition to basic test and evaluation, the chips were subjected to extensive robustness tests by varying temperature and voltage quite beyond nominal operating conditions. The results below show performance when the occupancy is 45, which is within the peak performance range of the canopy graph; the effect on performance at this occupancy is representative of the effects at other occupancies.

Figure 8.12 demonstrates the effect of scaling the voltage on peak performance. Extensive

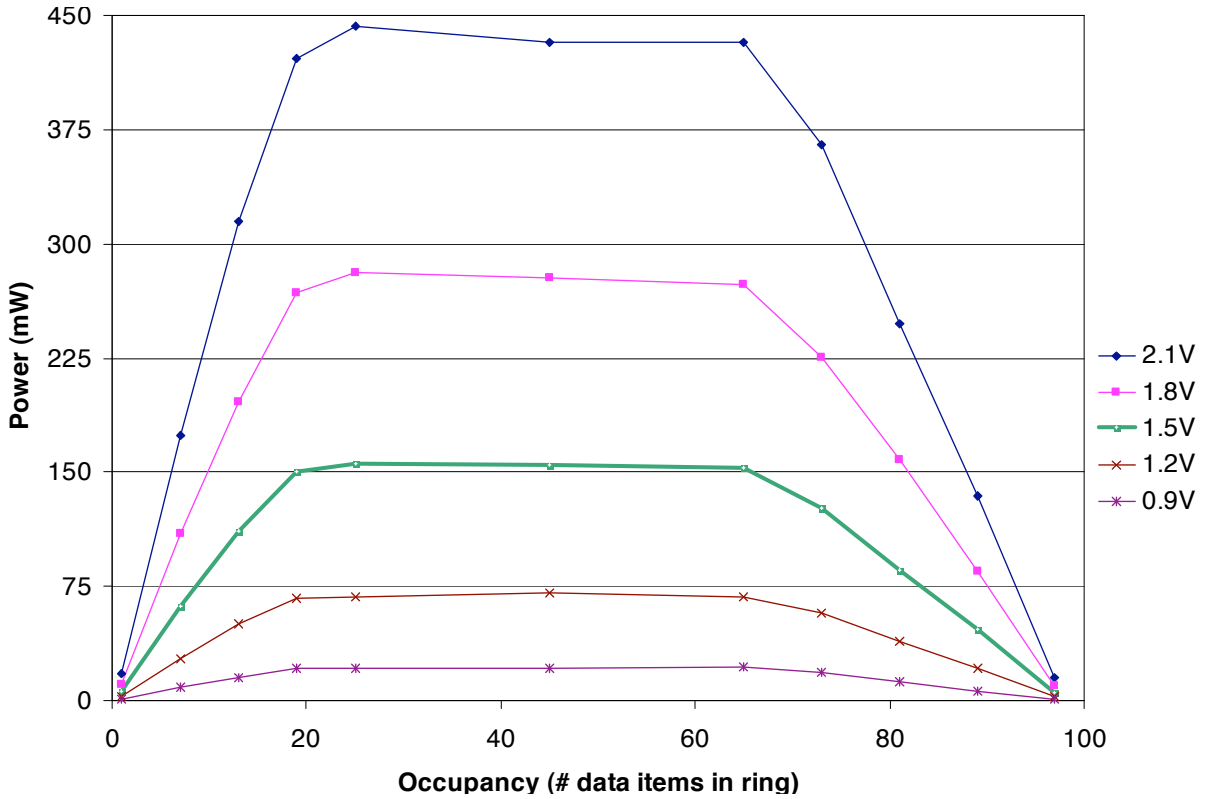


Figure 8.13: Impact of voltage variation on power consumption

testing was performed over the range of 0.5V–4V. The throughput changed automatically with the scaled voltage, and correct operation was observed throughout this range. Figure 8.14 shows power consumption (in milliWatts) for the chip. At nominal voltage, the peak power consumption (measured at peak throughput) was 75 mW, and the power consumption increases with increasing voltage.

Figure 8.15 plots $E\tau^2$ (*i.e.,* energy consumed per data item processed, times the cycle time squared), an energy-efficiency metric that is fairly invariant to voltage scaling [32]. In this experiment, the chip is operating at peak throughput as the voltage is varied. This value remains fairly constant, as expected, over the range 1V to 3V.

Finally, Figure 8.16 demonstrates the effect of temperature on the chip's throughput. This test was performed at nominal voltage. As expected, throughput decreases with an increase in temperature. The chip was fully functional over the entire range of temperatures tested

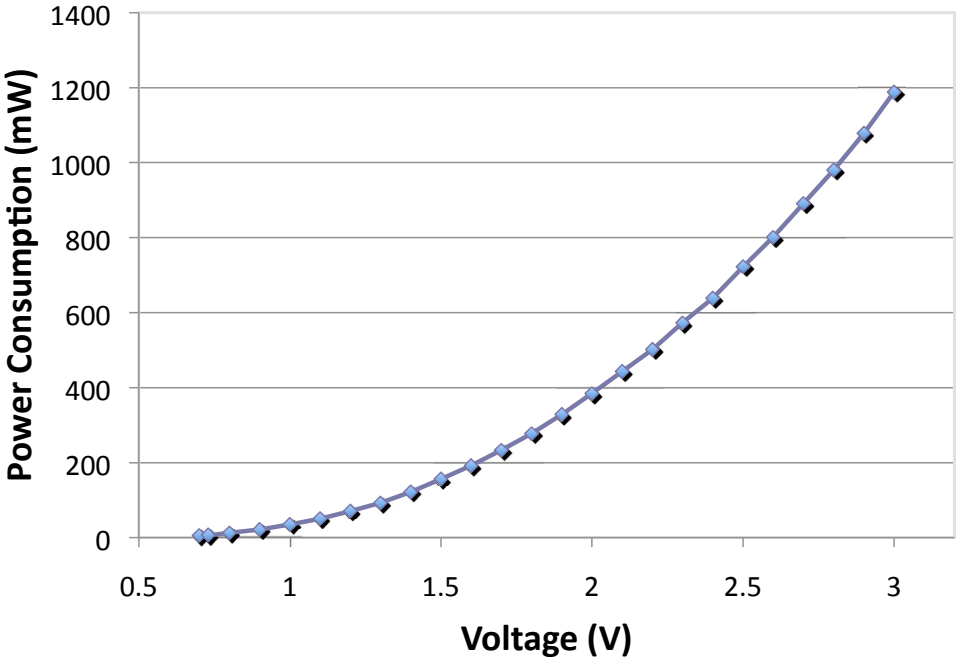


Figure 8.14: Power consumption varies with voltage

(-45°C to 150°).

8.2 JPEG Example

The goal of this section is to give a better understanding and intuition about the results of applying the analysis and optimization methods described in Chapters 4 and 5. It goes into greater detail on one of the examples already presented in Chapter 5: the JPEG encoder. This benchmark a real-world example of non-trivial size, and is therefore an interesting example to examine more carefully.

8.2.1 JPEG Benchmark Structure

The JPEG example used here is from a set of asynchronous benchmarks introduced by Hansen [19]. This benchmark uses 8-bit arithmetic and is set to convert an image of 8 pixels by 8 pixels. Figure 8.17 shows a diagram of the hierarchy of the JPEG encoder. It includes nested

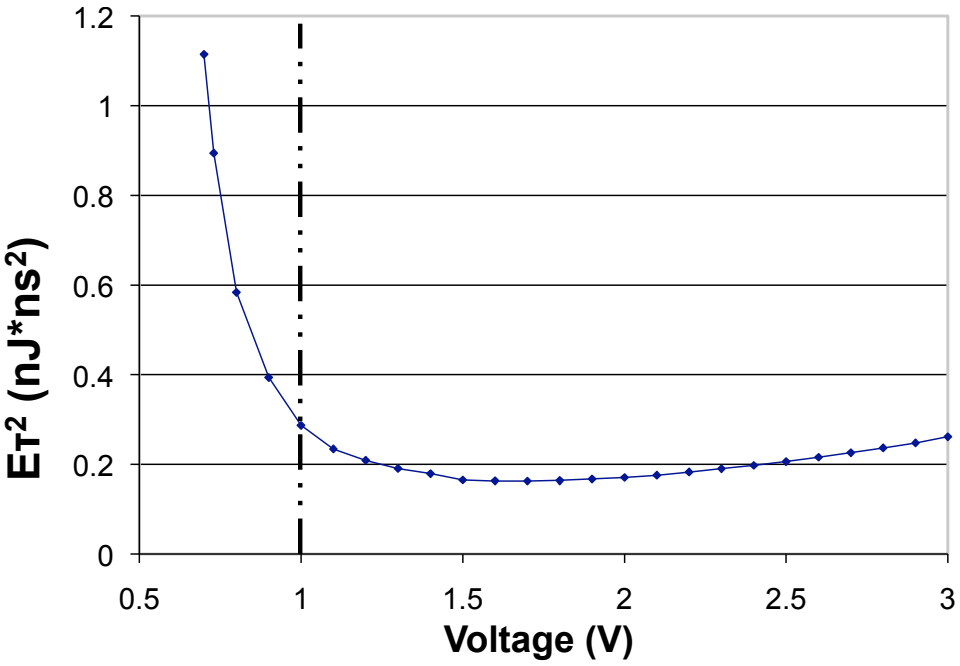


Figure 8.15: $E\tau^2$ vs. voltage at maximum throughput

loops, conditionals, and serial composition. For conciseness, the leaf nodes (*i.e.* individual stages) are not included in this figure. The original, non-optimized implementation includes a total of 46 pipeline stages.

Each of the nested loops in Figure 8.17 iterates a fixed number of times through the 8x8 image data. Following the loops are two conditionals in series, along with the logic to calculate their Boolean values. The final part of the JPEG encoder is a series of stages that performs the quantization step of the JPEG encoding algorithm.

The JPEG benchmark was also modeled at the gate level in Veriog; the tool for performance analysis can also output a Verilog model of the hierarchical system. Delays were assigned to the gates in the model based on the complexity of the operations (*e.g.* an 8-bit multiply has higher delay than an 8-bit add). The types of stages used to model this example include MOUSETRAP stages, forks, and joins [55], which were described in Section 7.2. In addition, several of the circuits introduced in Chapter 7 were also used to implement the JPEG example: conditional split, conditional select, merge without arbitration, and the arbitration

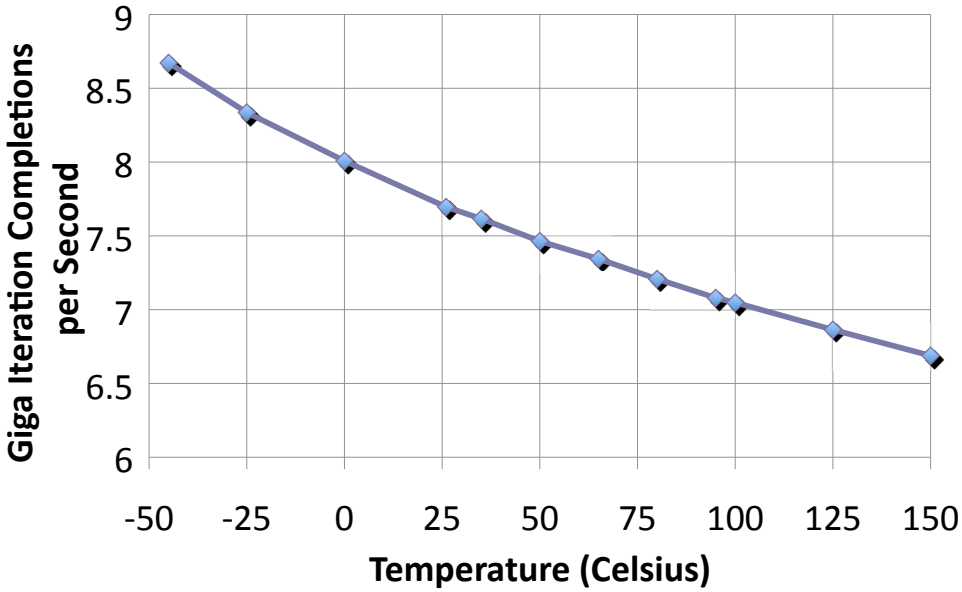


Figure 8.16: Impact of temperature on performance

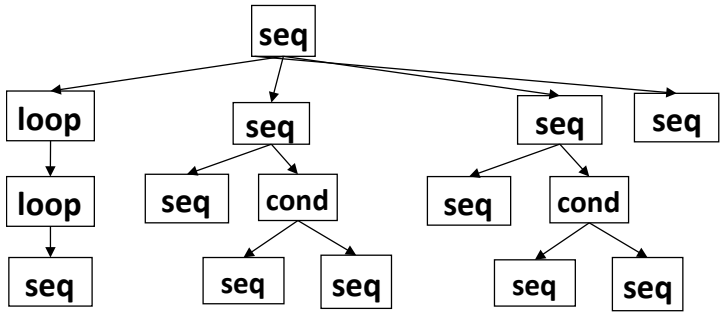


Figure 8.17: The hierarchical structure of the JPEG example.

stage. This example serves as a demonstration of the performance and correctness of these circuit components.

During Verilog simulation, the system is allowed to reach steady state operation by keeping the environment at a steady tick for a total of 1000 data item entrances and exits, which is quite sufficient for a system of this size. The system is then observed for a total of 10,000 item entrances and exits and the average throughput seen during this time is reported.

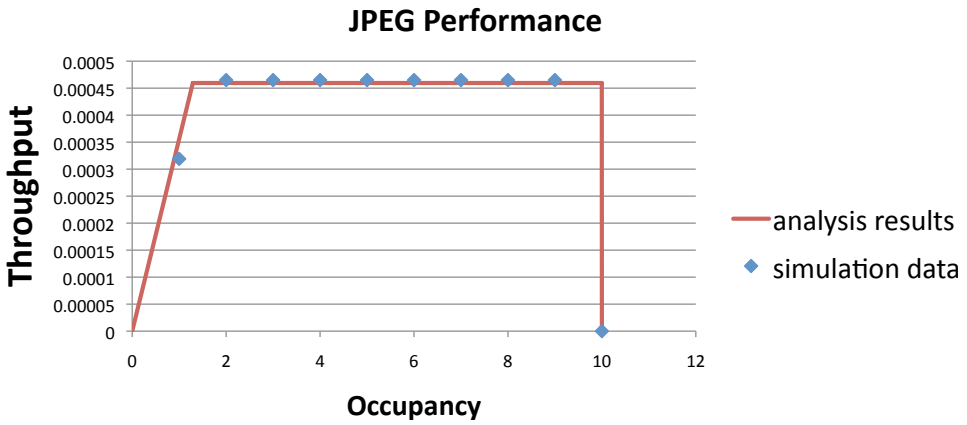


Figure 8.18: Performance of the JPEG benchmark before optimizing.

8.2.2 Performance Results

Figure 8.18 shows the performance of this system before any optimization takes place. In the figure, the line is the throughput as predicted by my analysis method. The predicted performance is very low because the nested loop (the left-most nodes in 8.17) is expected to act as a bottleneck during operation. In particular, each data item must loop 8 times through each loop, leading to a total of 64 iterations through the combined nested loop.

The points in Figure 8.18 are data obtained through Verilog simulation. Notice that they track very closely with the predicted performance of the system. Indeed, the maximum performance predicted by the analysis method is 4.60E-4 and the performance achieved during simulation is 4.65E-4. Assuming that the performance seen during Verilog simulation is the correct performance, the analysis method here shows a percent error of only about 1%. This shows that the performance analysis, even in non-trivial systems, is reasonably accurate.

8.2.3 Optimization Results

Because the main bottleneck is the loop, one of the first steps suggested by the bottleneck identification method is loop unrolling. If the inner loop is unrolled one time (*i.e.* the body of the inner loop is repeated twice) the performance is expected to double. Figure 8.19 shows the analyzed performance of the system after applying the loop unrolling TRIC one time. For

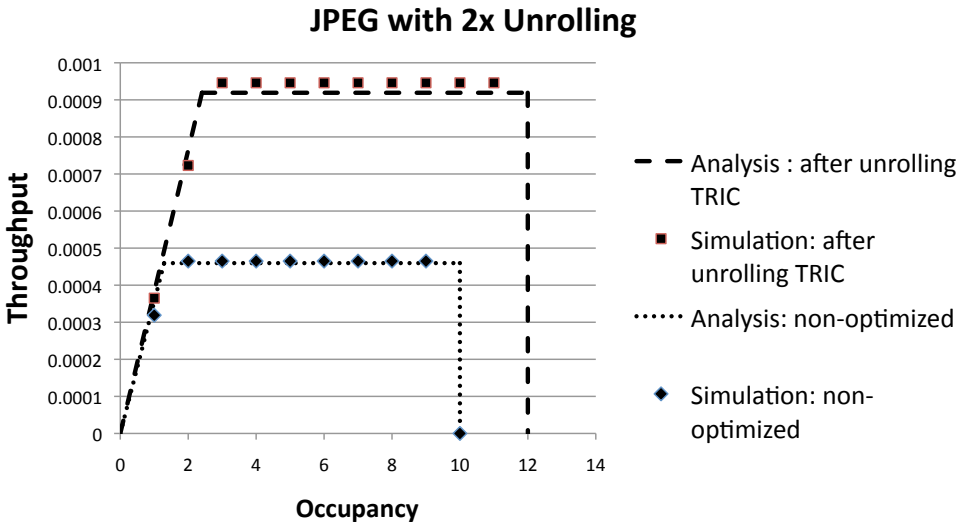


Figure 8.19: The performance of the JPEG benchmark after loop unrolling

comparison, the non-optimized performance is displayed again in the same chart. Verilog simulation again confirms that the throughput after unrolling once indeed reaches to twice the initial throughput. The throughput predicted by the analysis method is 9.19E-4 and the throughput found through simulation is 9.46E-4. This is a percent error of about 2.9%. This may be attributed to errors in modeling the circuit or to a possible over-estimate of the overheads associated with a pipeline loop.

To provide a challenge for the optimization framework, I next give it a goal throughput of 40x the initial throughput, which would require it to attain the throughput of 1.84E-2. The optimization method also needs a cost function to minimize during its search; in this case I use $E\tau^2$ as the cost function. The optimization method performs a total of 13 steps in reaching this goal, and in alleviates bottlenecks throughout the entire system. Figure 8.20 indicates the areas of the circuit in which the optimization method applies TRICs. The areas are numbered to show the order in which the TRICs were applied. In particular, the loop receives much attention, with the first three TRICs going towards loop unrolling. Within the loop, splitting of a slow stage also takes place. After the fifth TRIC, the loop ceases to be the bottleneck and the attention is turned to the first conditional in the system, which needs to be slack matched. Finally, a very slow, large multiplication at the end of the sequence of stages becomes the

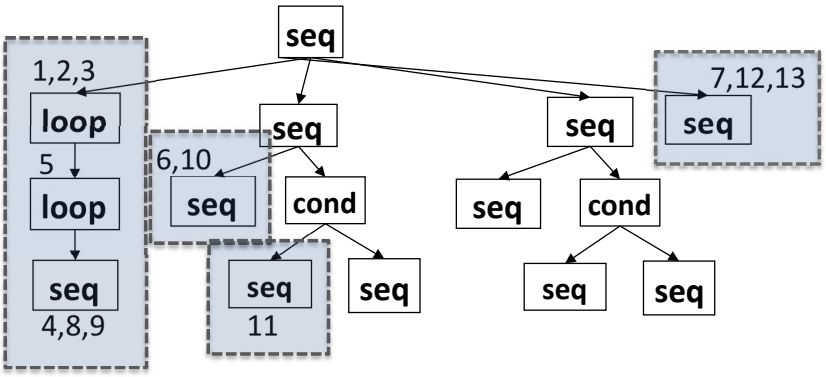


Figure 8.20: Nodes with detected bottlenecks

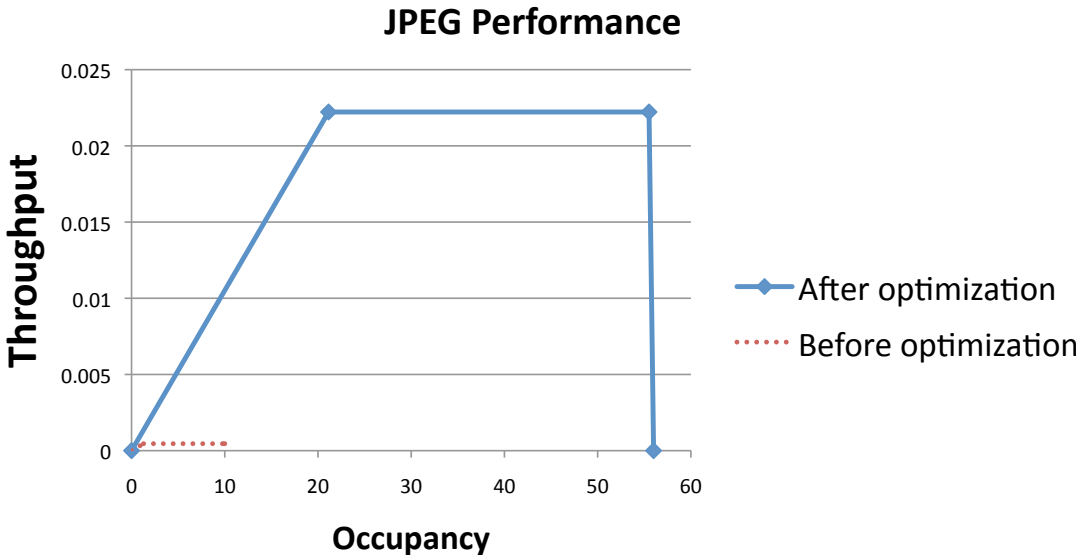


Figure 8.21: JPEG performance after 40x throughput improvement

bottleneck, so it is split and pipelined into a series of faster stages in using a sequence of TRICs.

The final throughput of the system after all this optimization takes place is shown in Figure 8.21. The final performance achieved is 2.2E-2. Though this slightly overshoots the performance goal, the optimization system found that this circuit is the best one for meeting the goal throughput of 1.84E-2 while maintaining a low energy cost. For comparison, the results before optimization are also shown on the same chart; the reader must look closely as the initial throughput was much lower.

8.3 Conclusion

This chapter first presented the design and analysis of a GCD chip as a case study in asynchronous design. It was implemented as a self-timed ring using MOUSETRAP pipeline style [56], and fabricated in a 0.13μm CMOS process. The chip was tested for stuck-at faults and timing constraint violations, and evaluated for performance over a wide range of supply voltages (0.5V to 4V) and temperatures (from -45°C to 150°C). A total of 24 parts were received, and all were tested to be functionally correct. The performance obtained was about 8 giga GCD algorithmic iterations completed per second, which is equivalent to a 1 GHz clocked ASIC. This case study bolsters our confidence that asynchronous design techniques can be useful for the design of reliable ASICS that are fast, testable, and that can operate under a wide range of conditions.

This chapter then went on to present a detailed example of the application of my analysis and optimization methods to a JPEG benchmark. The benchmark was designed at the gate level and simulated in Verilog, to provide a performance comparison that predicted by the analysis method. The optimization method is demonstrated first by giving a detailed description of the application of just one TRIC and its affects on the canopy graph. I then set the ambitious goal of 40x throughput, and illustrate which parts of the system are targeted for bottleneck removal. This case study serves as an example of how the analysis and optimization methods can be used in a real-world system to replace the tedium of repeated simulation and hand optimization.

Chapter 9

Conclusion

The goal of the work in this dissertation has been to support the design of fast and efficient dataflow systems. I have identified some key missing pieces in this area, and this work has taken several steps in the direction of creating fully-automated toolflow for the system-level design of pipelined, asynchronous application specific circuits. In particular, it focused on creating new algorithms that improve on the running time of performance analysis and optimization methods, without sacrificing accuracy. It also delves into low-level support for these designs by exploring testing procedures specific to pipelined asynchronous systems and by presenting five circuit-level designs that can be used to implement pipelined asynchronous systems.

Section 9.1 gives a more detailed summary of these contributions and the results obtained when applying them to examples and Section 9.2 suggests some future directions for extending this work.

9.1 Contributions of This Work

9.1.1 Performance Analysis

In this dissertation, I presented a fast and accurate performance analysis method for pipelined asynchronous systems. The results indicate that it is versatile enough to handle a variety of

system inputs, and that the runtime of the algorithm remained very fast (under 10 ms) for even large examples. When compared to Verilog simulations of the same examples, the match between the simulated results was within 2%.

The analysis method derives its speed from the observation that many circuits have a naturally hierarchical structure. The class of hierarchical systems is expressive enough to implement a wide range of pipelined implementations of several non-trivial examples, yet predictable enough in structure and operation that an analytic (rather than simulation-based) analysis method is possible.

9.1.2 System-level Optimization

This dissertation identified and classified a set of transformations that can improve throughput, developed a method for identifying bottlenecks that leverages the speed of the analysis method, and presented a system-level optimization method that iteratively applies transformations to remove bottlenecks. The system-level optimization problem was cast as a constrained optimization problem: minimize some cost function while reaching some throughput goal. The chosen solution method was a tree search, and includes two tree pruning methods that significantly improve the runtime.

The framework was able to achieve between 2x and 40x throughput improvement on all example circuits. It can handle a variety of different cost metrics, and the results show successful optimization for five different cost functions: $E\tau^2$, energy·area·, energy alone, area alone, and the energy-area product. When pruning techniques were used, all running times were less than two minutes for every example, even for very ambitious throughput goals.

9.1.3 Testing

Because many asynchronous pipeline styles achieve high speed by making assumptions about the order of local events, new kinds testing needed to be developed in order to expose errors that can occur when manufacturing defects invalidate these assumptions. This work further

classified these timing constraints into commonly occurring two categories: forward constraints and reverse constraints. It presented testing methods that will work on any pipeline style that contains these kind of constraints, and provided an example of how to apply testing to MOUSETRAP pipelines in particular. The approach is minimally intrusive, in that very little testing hardware needs to be added in order to achieve good fault coverage. The results show that 70% to 100% fault coverage is possible using this method.

9.1.4 Case Studies

This work presented two case studies in the design of pipelined asynchronous systems. The GCD chip case study was designed and fabricated in silicon, and tested for correctness. Its measured performance matches well with the predicted canopy graph. When tested for correctness using the testing methods presented in this dissertation, all 24 parts were found to operate correctly. The results also show that it is very robust to variations in voltage and temperature.

The JPEG case study was simulated at the gate level in Verilog. It is used as a detailed example of the process of applying analysis and optimization to a non-trivial circuit. The optimization framework was able to achieve a 40x speedup for the JPEG example, and this speedup is confirmed by detailed Verilog simulations.

9.2 Future Research Directions

This work has focused on hierarchical systems and fully leveraged this structure to improve running times of the analysis and optimization methods. However, it is possible that many of the methods presented here can be extended to systems that have topologies that are not strictly hierarchical. In particular, if sub-systems have non-hierarchical aspects but the system as a whole remains largely hierarchical, a hybrid approach could be applied: the canopy graph of a sub-system is obtained through simulation and then hierarchically combined with the rest

of the system-level performance information. Additionally, some non-hierarchical structures may still have operation that is regular enough to be analyzed directly using similar methods.

The work presented here could also benefit from further generalizations. For instance, the current system models the stages to have a fixed delay. A natural extension, then, is to include other delay distributions. Also, more optimizing transforms can be collected and categorized, to further improve the outcomes of the optimization framework. Finally, more work is needed in the area of testing for asynchronous testing to further improve fault coverage and extend testing to a more versatile set of systems.

Looking at the big picture, this entire work will be more useful if it is integrated into a fully automated and easy-to-use design flow. Full design flows, both for asynchronous and synchronous design, currently exist. [19, 2, 6] The goal is to integrate this work more fully into an existing flow rather than create an entirely new automated tool.

Finally, this work should be applied to designs that are even larger and more extensive than the ones presented in this dissertation. This will serve two goals. First, it will show the applicability of these methods under more diverse situations and in real-world applications. In particular, it would be useful to study the behavior of these systems during times when the assumptions (*e.g.* the steady state assumption) are invalidated due to internal or external pressures. This could lead to further extensions and revisions to the analysis and optimization methods. Second, a large design project will serve as a proof of concept for these design methods. Such a proof of concept will allow more designers to feel comfortable using these techniques to create efficient, high performance circuits. I believe that this will have an impact on the area of application specific circuit design, in that it will lead to faster design times, lower costs, and better performance.

Bibliography

[1] Alexander Taubin Alexander Smirnov. Heuristic based throughput analysis and optimization of asynchronous pipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, May 2009. 6

[2] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000. 224

[3] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2006. 6, 84, 90, 95

[4] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 184, Washington, DC, USA, 2006. IEEE Computer Society. 61

[5] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press. 89

[6] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conf. on Design Automation (EDAC)*, pages 384–389, 1991. 224

[7] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991. 188

[8] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217. 6, 85, 102

[9] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991. 59, 86

[10] S. Chakraborty, K. Yun, and D. Dill. Timing analysis of asynchronous systems using time separation of events. *IEEE Trans. on Computer-Aided Design*, 18(8):1061–1076, August 1999. 60, 86

[11] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996. 177, 178

[12] Uri Cummings. Terabit crossbar switch core for multi-clock-domain SoCs. In *Symposium Record of Hot Chips 15*, August 2003. 192

[13] J. C. Ebergen, S. Fairbanks, and I. E. Sutherland. Predicting performance of micropipelines using Charlie diagrams. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 238–246, 1998. 56

[14] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002. 2

[15] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Trans. on VLSI Systems*, 4(2):247–253, June 1996. 7, 14

[16] Gennette Gill, John Hansen, and Montek Singh. Loop pipelining for high-throughput stream computation using self-timed rings. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, November 2006. 58, 90, 196

[17] Mark Greenstreet and Brian De Alwis. How to achieve worst-case performance. In *In Proc. Seventh International Symposium on Asynchronous Circuits and Systems (ASYNC*, 2001. 27

[18] Mark R. Greenstreet and Kenneth Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2(3):139–148, November 1990. 7, 58, 59, 86

[19] John Hansen. Concurrency-enhancing transformations for asynchronous behavioral specifications. Master's thesis, University of North Carolina at Chapel Hill, 2007. 5, 214, 224

[20] John Hansen and Montek Singh. Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 2008. 88

[21] John Hansen and Montek Singh. Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach. In *ASYNC '08: Proceedings of the 2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 15–25, Washington, DC, USA, 2008. IEEE Computer Society. 123

[22] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Trans. on Computers*, 44(11):1306–1317, November 1995. 60, 86

[23] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. Wiley, 1979. 198

[24] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 159–170, December 2000. 6, 85

[25] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 40

[26] Ajay Khoche and Erik Brunvand. Testing micropipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 239–246, November 1994. 138, 140

[27] Matthew King and Kewal Saluja. Testing micropipelined asynchronous circuits. In *Proc. Int. Test Conf.*, 2004. 8, 140

[28] Alex Kondratyev, Amy Streich, and Lief Sorensen. Testing of asynchronous designs by inappropriate means: Synchronous approach. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 2002. 141

[29] P. Kudva, G. Gopalakrishnan, and E. Brunvand. Performance analysis and optimization for asynchronous circuits. In *Proc. Int. Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994. 7, 51, 86

[30] Prabhakar Kudva and Venkatesh Akella. Testing two-phase transition signalling based self-timed circuits in a synthesis environment. In *Proceedings of the 7th Int. Symp. on High-Level Synthesis*, pages 104–111. IEEE Computer Society Press, May 1994. 26, 57, 59

[31] Andrew M Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1998. 7, 21, 22, 27, 46, 53, 58, 86, 93

[32] Alain Martin, Mika Nystroem, and Paul Penzes. *Power-Aware Computing*, chapter ET^2: A Metric For Time and Energy Efficiency of Computation. Kluwer Academic Publishers, 2001. `http://caltechcstr.library.caltech.edu/archive/00000308`. 112, 131, 213

[33] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997. 153, 154

[34] Peggy B. McGee, Melinda Y. Agyekum, Moustafa A. Mohamed, and Steven M. Nowick. A level-encoded transition signaling protocol for high-throughput global communication. In *14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 116 – 127, 2008. 13

[35] Peggy B. McGee and Steven M. Nowick. An efficient algorithm for time separation of events in concurrent systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 2007. 60, 86

[36] Peggy B. McGee, Steven M. Nowick, and E. G. Coffman. Efficient performance analysis of asynchronous systems based on periodicity. In *Proc. of the 3rd IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 225–230, 2005. 7, 26, 51, 57, 59, 86

[37] E. G. Mercer and C. J. Myers. Stochastic cycle period analysis in timed circuits. In *Proc. Int. Symp. on Circuits and Systems*, pages 172–175, 2000. 59, 86

[38] Amitava Mitra, William F. McLaughlin, and Steven M. Nowick. Efficient asynchronous protocol converters for two-phase delay-insensitive global communication. In *ASYNC '07: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 186–195, Washington, DC, USA, 2007. IEEE Computer Society. 13

[39] Charles E. Molnar, Ian W. Jones, William S. Coates, Jon K. Lexau, Scott M. Fairbanks, and Ivan E. Sutherland. Two FIFO ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, February 1999. 201

[40] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an Int. Symp. on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959. 15

[41] James R. Munkres. *Topology; A First Course*. Prentice Hall College Div, 1974. 51, 52, 64

[42] Recep O. Ozdag and Peter A. Beerel. High-speed QDI asynchronous pipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 13–22, April 2002. 7, 14

[43] S. Pagey, G. Venkatesh, and S. Sherlekar. Issues in fault modeling and testing of micropipelines. In *Proc. of the Asian Test Symp.*, Hiroshima, Japan, November 1992. 8, 140

[44] Peggy Pang and Mark Greenstreet. Self-timed meshes are faster than synchronous. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 30–39. IEEE Computer Society Press, April 1997. 59, 86

[45] Ad Peeters. Asynchronous circuit technology is on the market. In *ASYNC '08: Proceedings of the 2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, page xiv, Washington, DC, USA, 2008. IEEE Computer Society. 2, 5, 124

[46] O. A. Petlin and S. B. Furber. Scan testing of micropipelines. In *Proc. IEEE VLSI Test Symp.*, pages 296–301, May 1995. 138, 140

[47] Piyush Prakash and Alain J. Martin. Slack matching quasi delay-insensitive circuits. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2006. 6, 84, 90, 95

[48] J. M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 1996. 192

[49] Marly Roncken, Emile Aarts, and Wim Verhaegh. Optimal scan for pipelined testing: An asynchronous foundation. In *Proc. Int. Test Conf.*, pages 215–224, October 1996. 140

[50] Marly Roncken and Erik Bruls. Test quality of asynchronous circuits: A defect-oriented evaluation. In *Proc. Int. Test Conf.*, pages 205–214, October 1996. 140

[51] Marly Roncken, Ken Stevens, Rajesh Pendurkar, Shai Rotem, and Parimal Pal Chaudhuri. CA-BIST for asynchronous circuits: A case study on the RAPPID asynchronous instruction length decoder. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 62–72. IEEE Computer Society Press, April 2000. 141

[52] Shai Rotem, Ken Stevens, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun, Rakefet Kol, Charles Dike, Marly Roncken, and Boris Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, April 1999. 141

[53] Feng Shi, Yiorgos Makris, Steven M. Nowick, and Montek Singh. Test generation for ultra-high-speed asynchronous pipelines. In *Proc. Int. Test Conf.*, November 2005. 8, 138, 139, 141, 143, 147, 150, 151, 156, 157, 158, 206, 209

[54] Montek Singh and Steven M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111–118. IEEE Computer Society Press, April 2000. 17, 139, 142

[55] Montek Singh and Steven M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 9–17, November 2001. xviii, 7, 14, 15, 139, 142, 166, 167, 168, 169, 171, 172, 191, 215

[56] Montek Singh and Steven M. Nowick. Mousetrap: High-speed transition-signaling asynchronous pipelines. *IEEE Trans. on VLSI Systems*, 15(6):684–698, June 2007. 220

[57] Montek Singh, Jose A. Tierno, Alexander Rylyakov, Sergey Rylov, and Steven M. Nowick. An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 gigahertz. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 84–95, April 2002. 21, 27, 46

[58] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001. 24, 30

[59] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001. 7, 14, 16, 139, 142

[60] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, Inc., 1999. 207

[61] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. 7, 14, 15

[62] Frank te Beest and Ad Peeters. A multiplexor based test method for self-timed circuits. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 2005. 141

[63] M. Theobald and S. M. Nowick. Transformations for the synthesis and optimization of asynchronous distributed control. In *Proc. ACM/IEEE Design Automation Conf.*, June 2001. 6, 85, 102, 123

[64] Kees van Berkel, Ad Peeters, and Frank de Beest. Adding synchronous and LSSD modes to asynchronous circuits. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 2002. 141

[65] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. Global critical path: A tool for system-level timing analysis. In *Proc. ACM/IEEE Design Automation Conf.*, pages 783–786, June 2007. 26, 51, 86

[66] Girish Venkataramani and Seth Goldstein. Leveraging protocol knowledge in slack matching. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 2006. 85

[67] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang. A self-timed chip for division. In Paul Losleben, editor, *Advanced Research in VLSI*, pages 75–95. MIT Press, 1987. 7, 14, 19, 21, 27, 30, 36, 46, 58, 59, 74, 86

[68] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991. 196

[69] Anthony J. Winstanley, Aurelien Garivier, and Mark R. Greenstreet. An event spacing experiment. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 47–56, April 2002. 56

[70] A. Xie and P. A. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 239–268. Kluwer Academic Publishers, March 2000. 26, 57, 59, 86

[71] Aiguo Xie and Peter A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75. IEEE Computer Society Press, April 1997. 7, 26, 51, 59, 86

[72] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996. 7, 14, 201

[73] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 140–153. IEEE Computer Society Press, April 1997. 123

[74] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Trans. on VLSI Systems*, 6(4):643–655, December 1998. 123