Title
High-Level Synthesis for Nanoscale Integrated Circuits

Permalink
https://escholarship.org/uc/item/78t0x4m9

Author
Liu, Bin

Publication Date
2012

Peer reviewed|Thesis/dissertation
High-Level Synthesis for Nanoscale Integrated Circuits

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Bin Liu

2012
ABSTRACT OF THE DISSERTATION

High-Level Synthesis for Nanoscale Integrated Circuits

by

Bin Liu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2012

Professor Jason Cong, Chair

Increased design complexity and time-to-market pressure in the integrated circuit (IC) industry call for a raised level of abstraction at which designs are specified. High-level synthesis is the process of generating register-transfer level (RTL) implementations from behavioral specifications, and it is the key enabler for a designing at a higher level beyond RTL. As IC manufacturing technology scales down to nanoscopic scale, the synthesis tools face a number of new challenges, including complexity, power and interconnect. In this dissertation, we propose a spectrum of new techniques in high-level synthesis to address the new challenges and to improve the quality of synthesis results.

1. Efficient and versatile scheduling engine using soft constraints.

We present a scheduler that distinguishes soft constraints from hard constraints when exploring the design space, and identify a class of tractable scheduling problems with soft constraints. By exploiting the total unimodularity of the constraint matrix in an integer-linear programming formulation, we are able to solve the problem optimally in polynomial time. Compared to traditional methods, the proposed approach allows easier expression of various design intentions and optimization directions, and, at the same time, gives the scheduler freedom to make global trade-offs optimally. We show that this scheduling engine is flexible
enough to support a variety of design considerations in high-level synthesis.

2. **Behavior-level observability analysis and power optimization.**

   We introduce the concept of behavior-level observability and its approximations in the context of high-level synthesis, and propose an efficient procedure to compute an approximated behavior-level observability of every operation in a dataflow graph. The algorithm exploits the observability-masking nature of some Boolean operations, as well as the `select` operation, and treats other operations as black boxes to allow efficient word-level analysis. The result is proven to be exact under the black-box abstraction. The behavior-level observability condition obtained by our analysis can be used to optimize *operation gating* in the scheduler. This leads to more opportunities in subsequent RTL synthesis for power reduction. To the best of our knowledge, this is the first time behavior-level observability analysis and optimization are performed in a systematic manner.

3. **Layout-friendly high-level synthesis.**

   We study a number of structural metrics for measuring the layout-friendliness of microarchitectures generated in high-level synthesis. For a piece of connected netlist, we introduce the *spreading score* to measures how far components can be spread from each other with bounded wire length in a graph embedding formulation. The intuition is that components in a layout-friendly netlist (e.g., a mesh) can spread over the layout region without introducing long interconnects. Spreading score can be approximated efficiently using a semidefinite programming relaxation. Another metric based on neighborhood population is also proposed. On a number of benchmarks, spreading score shows stronger bias in favor of interconnect structures that have shorter wire length after layout, compared to previous metrics based on cut size and total multiplexer inputs.
The dissertation of Bin Liu is approved.

Lieven Vandenberghe

Miodrag Potkonjak

Milos Ercegovac

Jason Cong, Committee Chair

University of California, Los Angeles

2012
To my family.

In memory of my grandmother
Zhenying Zhang (1918–2002).
# Table of Contents

1 Introduction ................................................................. 1
   1.1 High-Level Synthesis .................................................. 1
   1.2 A Brief History ......................................................... 2
   1.3 Motivation for Our Research ........................................ 7
   1.4 Overview of Our Research ........................................... 11

2 Preliminaries ................................................................. 15
   2.1 System Design Using High-Level Synthesis ...................... 15
   2.2 The Anatomy of xPilot ................................................ 18

3 Scheduling with Soft Constraints ....................................... 20
   3.1 Introduction ............................................................ 20
   3.2 Problem Formulation .................................................. 27
   3.3 Handling Soft Constraints .......................................... 28
      3.3.1 A Penalty Method for Soft Constraints .................... 28
      3.3.2 Total Unimodularity and Implications ...................... 29
      3.3.3 Form of the Penalty Term .................................... 31
      3.3.4 Overall Flow .................................................... 40
   3.4 Example Applications ................................................ 41
      3.4.1 Register Optimization .......................................... 41
      3.4.2 Delay Budgeting ................................................ 43
      3.4.3 Operation Grouping ............................................ 45
      3.4.4 Incremental Synthesis ........................................ 47
### 4 Behavior-Level Observability Analysis and Power Optimization  

#### 4.1 Introduction  

#### 4.2 Behavior-Level Observability  

- **4.2.1** Observability for a General Function  
- **4.2.2** Dataflow Graph Abstraction  

#### 4.3 Behavior-Level Observability Analysis  

- **4.3.1** Review of Observability Computation in a Boolean Network  
- **4.3.2** Observability Analysis with Black Boxes  

#### 4.4 Scheduling for Operation Gating Optimization  

- **4.4.1** Observability under a Given Schedule  
- **4.4.2** Previous Work  
- **4.4.3** Optimization Using Soft Constraints  

#### 4.5 Experimental Result  

#### 4.6 Remarks  

### 5 Metrics for Layout-Friendly High-Level Synthesis  

#### 5.1 Motivation  

#### 5.2 Spreading Score  

#### 5.3 Weighted Neighborhood Population  

#### 5.4 Previous Metrics  

- **5.4.1** A Metric Based on Cut Size  
- **5.4.2** Total Multiplexer Inputs  

---

vii
LIST OF FIGURES

2.1 Block diagram of a Freescale i.MX SoC (source: Freescale) . 16
2.2 Hardware/software co-design using high-level synthesis . 17
2.3 A typical high-level synthesis flow . 19

3.1 Structure of a scheduler with soft constraints . 25
3.2 Some typical penalty functions. An offset is added to the exponential penalty to satisfy Equation 3.6. The binary penalty is nonconvex. . . 33
3.3 The area-latency tradeoff curve of an operation . 44
3.4 The penalty function for the delay-budgeting soft constraint . 45
3.5 A simplified illustration of a DSP48A slice . 46

4.1 An example of clock gating based on ODC conditions in RTL synthesis . 53
4.2 An example of the limitation of ODC-based clock gating in RTL synthesis . 54
4.3 A dataflow graph described as a C++ function . 55
4.4 A schedule for the dataflow graph in Figure 4.3 and the implied control structure after operation gating . 56
4.5 An alternative schedule of the dataflow graph in Figure 4.3 and the implied control structure after operation gating . 57
4.6 The black-box abstraction of the dataflow graph in Figure 4.3, where v9 is a generalized select operation, and black boxes are filled with shade . 63
4.7 Relations between observabilities. The red arc shows the way we obtain BFSMDO . 76
4.8 Transformation to get an even better implementation . 84
5.1 Linear regression for test1. ........................................ 97
5.2 Linear regression for test2. ........................................ 98
5.3 Linear regression for test3. ........................................ 99
5.4 Linear regression for test4. ....................................... 100
5.5 Linear regression for test5. ....................................... 101
5.6 Linear regression for test6. ....................................... 102
5.7 Total area and average wire length of solutions for test 1. .... 104
5.8 Total area and average wire length of solutions for test 2. .... 105
5.9 Total area and average wire length of solutions for test 3. .... 105
5.10 Total area and average wire length of solutions for test 4. ... 106
5.11 Total area and average wire length of solutions for test 5. ... 106
5.12 Total area and average wire length of solutions for test 6. ... 107
List of Tables

3.1 Benchmark characteristics. .......................................................... 50
3.2 Experimental results for register reduction. ................................. 51
4.1 Smoothed behavior-level observability computed by Algorithm 1. ... 74
4.2 Benchmark characteristics. .......................................................... 80
4.3 Experimental results for operation gating. .................................... 82
ACKNOWLEDGMENTS

First and foremost, I thank my advisor, Prof. Jason Cong, for his guidance and constant support over the past six years, and for being an inspiring role model. What I learned from him has greatly benefited my research, career, and life. Although I have completed my Ph.D. program, I feel reluctant to leave his research group because I find that there is still a lot to learn.

I also wish to express my gratitude to Prof. Milos D. Ercegovac, Prof. Miodrag Potkonjak, and Prof. Lieven Vandenberghe for their help and constructive comments on my research. In particular, what I learned in Prof. Vandenberghe’s classes enabled me to develop the formulations presented in this dissertation.

It has been a great pleasure to work with many talented colleagues during my years of study. Dr. Yiping Fan, Dr. Guoling Han, Dr. Wei Jiang, and Dr. Zhiru Zhang were the original builders of the xPilot/AutoPilot system, which serves as the infrastructure of my work. I also benefited from discussions and collaborations with members of the UCLA VLSI CAD group: Yuting Chen, Karthik Gururaj, Hui Huang, Muhuan Huang, Cheng-Tao Hsieh, Jeonghun Kim, Taemin Kim, John Lee, Albert Liu, Chunyue Liu, Guojie Luo, Kirill Minkovich, Eric Radke, Junjuan Xu, Bo Yuan, Yuxin Wang, Peng Zhang, and Yi Zou. I am particularly grateful to Dr. Zhiru Zhang, who is my primary research collaborator. I enjoyed my time working with him both at UCLA and at AutoESL/Xilinx. Zhiru generously shared many interesting and enlightening ideas with me; some of those ideas led to the fruitful results as reported in this dissertation. I also sincerely thank Ms. Janice Wheeler, who helped me polish this dissertation, as well as my other publications.

I appreciate the understanding and support from my managers in industry, including Dr. Zhiru Zhang and Dr. Peichen Pan at AutoESL/Xilinx, and Mr. Paul Glendenning at Micron. They graciously approved my excessive time-off requests, so that I could
have adequate time to work on this dissertation while still continuing my full-time employment. I would also like to acknowledge the funding support provided by the Semiconductor Research Corporation, the Gigascale Systems Research Center, and a dissertation-year fellowship from the UCLA graduate division.

This dissertation would not have been possible without my family. I feel deeply indebted to my parents, Haoan Liu and Fuge Zhang, for all the sacrifices they made. This achievement is so humble compared to their unconditional love and generous help. And I am very grateful to my wife Wen, who decided to move from China to the United States to support me, even though it meant significant challenges in her own life and career. The birth of my son David in 2009 has made life even more enjoyable. I want to say thank you to my family: I feel very blessed to have a family like you, and you have always been the biggest motivation in my quest for excellence.
VITA

2004 B.E. (Computer Science and Engineering), Tsinghua University, Beijing, China

2006 M.E. (Computer Science and Engineering), Tsinghua University, Beijing, China


2011–2012 Senior software engineer, Xilinx, Inc.

2012–present Senior architect, Micron Technology, Inc.

PUBLICATIONS


CHAPTER 1

Introduction

1.1 High-Level Synthesis

Following Moore’s Law, the number of transistors in a typical integrated circuit (IC) has been doubling approximately every two years since the 1970s. This trend is expected to continue for at least another decade (albeit at a possibly slower speed after 2013, according to ITRS [80]). Such exponential growth allows more and more powerful chips to be built at low costs, but at the same time it causes significant difficulties in design and verification of these chips due to the increased complexity.

Specifying the design at a higher level of abstraction has been an effective way to cope with complexity. It is possible to design an IC by drawing the layout directly; this is called full custom design at the geometry level. Full custom design is extremely tedious; today it is used only when the chip is very small or highly regular (e.g., memory) or has a huge volume and very stringent performance/power/cost requirements (e.g., certain microprocessors) to justify the design effort. With the introduction of automatic layout tools and the transistor/logic gate abstraction, a digital designer can specify at the transistor/gate level, and use layout tools to get geometry-level implementations; this simplifies the design task and allows the designer to focus on the logic part of the circuit. Register-transfer level (RTL) is one level above the gate level. By describing the state transition between sequential elements, the implementation of the combinational logic network can be left to the synthesis tool, and the designers only need to consider how state holding elements like registers or memories are initialized and updated. The
RTL design style has been widely adopted since the 1990s, with the maturity of RTL synthesis tools like Synopsys Design Compiler. Nowadays, the RTL-based approach is the dominating methodology in digital IC designs.

It is believed that raising the design abstraction to another level beyond RTL, i.e., the behavior level, is the next boost to design productivity. A behavior-level specification describes the algorithm to be implemented, without details on the structure of the circuit. High-level synthesis (also known as behavioral synthesis) is the process of generating an RTL model from an untimed behavioral description using a high-level language such as C/C++ or Matlab. High-level synthesis is the key link from the behavior level to the RTL; without a practical high-level synthesis flow, designers will have to translate high-level models to RTL models manually, and such a process is tedious and error-prone. Therefore, high-level synthesis has gained much interest from both academia and industry.

1.2 A Brief History

Compilers for high-level languages have been successful since the 1950s. The idea of automatically generating circuit implementations from high-level behavioral specifications arises naturally with the increasing design complexity of ICs. Early efforts (in the 1980s and early-1990s) on high-level synthesis were mostly research projects, where multiple prototype tools were developed to call attention to the methodology and to experiment with various algorithms.

To our knowledge, the first high-level synthesis tool, Carnegie-Mellon University Design Automation (CMU-DA), was built by researchers at Carnegie Mellon University in the 1970s [44, 128]. In this tool, the design is specified at the behavior level using the “instruction set processor specification” (ISPS) language [7]. It is then translated into an intermediate dataflow representation called the “value trace” [146] before producing an RTL model. Classical code transformation techniques in software compil-
ers, including dead-code elimination, constant propagation, redundant sub-expression elimination, code motion, and common subexpression extraction were performed in CMU-DA. The synthesis engine also included many steps familiar in later high-level synthesis tools, such as datapath allocation, module selection, and controller generation. CMU-DA supported hierarchical design and included a simulator of the original ISPS language. Although many of the methods used were very preliminary, the innovative flow and the design of toolsets in CMU-DA quickly generated considerable research interest.

In the subsequent years of the 1980s and early 1990s, a number of similar high-level synthesis tools were built, mostly for research and prototyping. Examples of academic efforts include MIMOLA [109], ADAM [63, 81], HAL [130], Hercules/Hebe [41, 42, 92], and Hyper/Hyper-LP [24, 132]. Industry efforts include Cathedral and its successors [40], BSSC [168], and Yorktown Silicon Compiler [19], among many others. Like CMU-DA, these tools typically decompose the synthesis task into a few steps, including code transformation, module selection, operation scheduling, datapath allocation, and controller generation. Many fundamental algorithms addressing these individual problems were also developed. For example, the list scheduling algorithm [1] and its variants were widely used to solve scheduling problems with resource constraints [127]; the force-directed scheduling algorithm developed in HAL [129] was used to optimize resource requirements under a performance constraint. The path-based scheduling algorithm in the Yorktown Silicon Compiler was proposed to optimize performance for designs with conditional branches [20]. The Sehwa tool in ADAM was able to generate pipelined implementations and explore the design space by generating multiple solutions [82, 126]. The relative scheduling technique developed in Hebe was an elegant way to handle operations with unbounded delay [90]. Conflict graph coloring techniques were developed and used in several systems to share resources in the datapath [99, 130].

These early high-level tools often used custom languages for design specification.
Besides the ISPS language used in CMU-DA, a few other languages were notable. HardwareC is a language designed for use in the Hercules system [91]. Based on the popular C programming language, it supports both procedural and declarative semantics and has built-in mechanisms to support design constraints and interface specifications. This is one of the earliest C-based hardware synthesis languages for high-level synthesis. The Silage language used in Cathedral was specifically designed for the synthesis of digital signal processing hardware [40]. It has built-in support for customized data types, and allows easy transformations [24, 132]. The Silage language, along with Cathedral, represented an early domain-specific approach in high-level synthesis.

These early research projects helped to create a basis for high-level synthesis with many innovations, and some were even used to produce real chips. However, these efforts did not lead to wide adoption among designers. A major reason was that the methodology of using RTL synthesis was not yet widely accepted at that time and RTL synthesis tools were not mature. Thus, high-level synthesis, built on top of RTL synthesis, did not have a sound foundation in practice. In addition, simplistic assumptions were often made in these early systems—many of them were technology-independent, and inevitably led to suboptimal results.

With improvements in RTL synthesis tools and the wide adoption of RTL-based design flows in the 1990s, industrial deployment of high-level synthesis tools became more practical. Proprietary tools were built in major semiconductor design houses including Philips [105], Siemens [12], IBM [11], and Motorola [93]. Major EDA vendors also began to provide commercial high-level synthesis tools. In 1995, Synopsys announced the Behavioral Compiler [88], which generates RTL implementations from behavioral hardware description language (HDL) code and connects to downstream tools. Similar tools include Monet from Mentor Graphics [48] and Visual Architect from Cadence [89]. These tools received wide attention, but failed to widely replace RTL design. This is partly ascribed to the use of behavioral HDLs as input languages, which are not popular among algorithm and system designers and require steep learning
curves.

Since around 2000, a new generation of high-level synthesis tools has been developed in both academia and industry. Unlike many predecessors, most of these tools focus on using C/C++ or C-based languages to capture the design. This makes the tools much more accessible to algorithm and system designers compared to previous tools that only accept HDLs. It also enables hardware and software to be built using a common language, thus facilitating software/hardware co-design and co-verification. Furthermore, the use of C-based languages makes it easy to leverage the newest technologies in software compilers for parallelization and optimization in high-level synthesis tools.

In fact, there has been an ongoing debate on whether C-based languages are proper choices for high-level synthesis [46, 142]. Despite the many advantages of using C-based languages, opponents often criticize C/C++ as languages only suitable for describing sequential software that runs on microprocessors. Specifically, the deficiencies of C/C++ include the following.

1. Standard C/C++ lack built-in constructs to explicitly specify bit accuracy, timing, concurrency, synchronization, hierarchy, and others, which are critical to hardware design.

2. C and C++ have complex language constructs, such as pointers, dynamic memory management, recursion, and polymorphism, which do not have efficient hardware counterparts and lead to difficulty in synthesis.

To address these deficiencies, most C-based high-level synthesis tools have introduced additional language extensions and restrictions to make design inputs more amenable to hardware synthesis. Common approaches include both restriction to a synthesizable subset that discourages or disallows the use of dynamic constructs (as required by most tools) and introduction of hardware-oriented language extensions (HardwareC [91], SpecC [55], Handel-C [3]), libraries (SystemC [77]), and compiler
directives to specify concurrency, timing, and other constraints. For example, Handel-C allows the user to specify clock boundaries explicitly in the source code. Clock edges and events can also be explicitly specified in SpecC and SystemC. Pragmas and directives along with a subset of ANSI C/C++ are used in many commercial tools. An advantage of this approach is that the input program can be compiled using standard C/C++ compilers without change, so that a module can be easily implemented as either software running on processors or dedicated hardware accelerators. In addition, co-simulation of hardware and software can be performed without code rewriting. At present, most commercial high-level synthesis tools use some form of C-based design entry, although tools using other input languages (e.g., BlueSpec [121], Esterel [47], and MATLAB [68]) also exist.

With the increasing popularity of field-programmable gate arrays (FPGAs), many recent high-level synthesis tools are targeting implementation on FPGA, in addition to the traditional ASIC platform. FPGAs have continually improved in capacity and speed in recent years, and their programmability makes them an attractive platform for many applications in signal processing, communication, and high-performance computing. There has been a strong desire to make FPGA programming easier, and high-level synthesis is regarded as part of the solution. Many high-level synthesis tools are designed to specifically target FPGAs, including Streams-C [60], CASH [18], SPARK [66, 67], Impulse C [131], ASC [113], Trident [154], Handel-C Compiler [3], DIME-C [120], GAUT [38], ROCCC [65, 158], C2H [4], and LegUp [22]. ASIC tools also commonly provide support for targeting an FPGA tool flow in order to enable system emulation.

As of 2012, major commercial C-based high-level synthesis tools include Cadence’s C-to-Silicon Compiler, Forte’s Cynthesizer [114], Calypto’s Catapult C (formerly, Mentor Catapult C) [15], NEC’s Cyber Workbench [159], Synopsys Synphony C (formerly, Synfora’s PICO Express, originating from the PICO project in HP Labs [144]), and Xilinx Vivado high-level synthesis (formerly AutoESL’s AutoPilot [169], originating from the xPilot project in UCLA [29]).
1.3 Motivation

Despite the limited success of early generations of commercial high-level synthesis tools, we believe that the recent wave of research and development will likely lead to a much wider acceptance of high-level synthesis in the near future. The market needs for high-level synthesis have been growing in the nanoscale IC technologies. The reason for this is the following.

- **Larger silicon capacity and cheaper transistors.** In the 1990s, when the first generation of commercial high-level synthesis tools was introduced, the design complexity was still manageable at the RT level, especially with other productivity boosters such as IP reuse. With continued scaling, these techniques become insufficient, and a paradigm shift toward a higher level of abstraction is increasingly desirable. At the same time, as transistors become cheaper, designers are more likely to accept a slight overhead in chip area when design productivity is enhanced by switching from manual RTL coding to high-level synthesis.

- **Need for platform-based software/hardware co-design and exploration.** With the coexistence of micro-processors, digital signal processors (DSPs), memories and custom logic on a single chip (this is usually called system-on-a-chip, or SoC), more software elements are involved in the process of designing a modern embedded system. A widely accepted methodology is to specify the design functionality for both embedded software and customized hardware logic, so that the system can be verified at an early design stage. Given the functionality models, an automated high-level synthesis flow is desirable because it allows system designers to experiment with different hardware/software boundaries and explore various area/power/performance tradeoffs from a single common functional specification. Since C-based languages are very popular for early system modeling, C-based high-level synthesis tools fit naturally in this flow.
• **Trend toward extensive use of accelerators and heterogeneous SoCs.** In recent years, while transistor dimensions continued scaling, energy efficiency per transistor did not improve as fast as transistor density, thus creating a utilization wall due to power limitations. This implies that a large fraction of the chip will be “dark silicon,” i.e., components that cannot run at a full speed [153]. It is estimated in [49] that more than 50% of a fixed-size chip must be powered off in a projected 8nm process. Thus, many SoCs and chip multiprocessors move toward the inclusion of many application-specific or domain-specific accelerators, which are only used for specific tasks, in order to improve energy efficiency. According to the ITRS prediction [80], the number of on-chip accelerators will reach 3000 by 2024. Such a trend toward more customized silicon makes high-level synthesis an intriguing technology to easily generate application-specific or domain-specific accelerators.

• **Rise of FPGAs.** FPGAs have become popular in recent years due to the increasing ASIC design and manufacturing cost, which is well over $1M in nanometer technologies [80]. Compared to ASIC alternatives, FPGA designs typically face less pressure on verification (as in-system simulation can be done quickly and inexpensively) but more pressure on time-to-market; thus the productivity benefit of high-level synthesis is particularly desirable. In addition, FPGA designers are more likely to accept suboptimal quality of results (QoR) produced by an immature synthesis tool, as long as the resulting design fits into the target FPGA device and meets the desired performance specifications. QoR can be improved by subsequent design optimizations in the behavioral description and synthesis directives at a low cost in FPGA designs.

At the same time, it is easier than ever to build a high-quality tool chain for high-level synthesis, because the technical foundation for high-level synthesis has improved over the years in the following aspects.
• **Maturity of RTL synthesis flow.** When early commercial high-level synthesis tools were introduced in the mid-1990s and early-2000s, the industry was still struggling with timing closure between logic and physical designs. There was no dependable RTL-to-GDSII flow, which made it difficult to consistently measure, track, and enhance results generated by high-level synthesis. Highly automated RTL-to-GDSII solutions became available in the 2000s (e.g., provided by the IC Compiler from Synopsys [151]). Thus, it is much easier to get desired results without tweaking the RTL generated by a high-level synthesis tool.

• **Availability of high-quality compiler infrastructures.** Early high-level synthesis tools mostly used in-house languages, and thus a significant amount of time needed to be spent on the compiler frontend. With the recent shift toward C-based languages as design input, high-level synthesis tools can easily take advantage of existing high-quality compiler infrastructures (e.g., SUIF [69], LLVM [101], etc.) for parsing and code transformations. Many techniques for code optimization and parallelization in software compilation (e.g., dead-code elimination, code motion) can be directly used or slightly altered in a high-level synthesis tool; customized transformations specifically for hardware design (e.g., bitwidth optimization) can also be integrated in the these compiler frameworks with ease. In addition to code optimization, the compiler infrastructures can also provide other features useful in a commercial synthesis tool, such as the support for source-level debugging and instrumentation.

• **Active development of synthesis techniques.** Over the years, many improvements have been made in the high-level synthesis tools. Detailed platform characterization is now widely used in commercial synthesis tools; new target architectures have been introduced (e.g., [2, 30, 32, 103, 137]); and new algorithms for compilation (e.g., [33, 34, 125, 148], scheduling (e.g., [36, 83, 136, 171]), and binding (e.g., [26, 31, 35, 86]) are proposed for QoR optimization.
In sum, we believe that high-level synthesis shows promise in the design of nanoscale ICs, given the increasing market needs and the technical development in the tool flow. On the other hand, nanoscale technologies give rise to a number of new challenges that are not adequately addressed in traditional tools. Specifically, our research is motivated by the following new challenges in high-level synthesis.

- **Multiple design objectives/constraints.** Traditionally, performance and area were the only concerns for RTL designers. Thus, many existing algorithms for high-level synthesis focus exclusively on performance and area. In nanoscale IC technologies, many other design objectives and constraints, such as power, heat, reliability, and yield, need to be considered together with the traditional metrics when optimizing the RTL model. Thus, new synthesis algorithms are needed to flexibly support multiple design objectives/constraints and perform effective optimization.

- **Power.** Power dissipation, including both dynamic power (switching power and short-circuit power) and static power (leakage power), has been an increasingly critical issue. This is not just a concern when designing ICs for battery-powered devices like mobile phones and tablets; instead, power is a universal problem in almost every nanoscale design, due primarily to limitations of the continued scaling of silicon CMOS technology. The device scaling has arrived at a dimension where leakage power caused by gate tunneling currents and subthreshold leakage currents are very significant, thus limiting the continued shrinking of gate oxide thickness ($T_{ox}$) and channel length. There has been a significant slowdown of threshold voltage scaling in recent technology generations, despite the innovations in process such as silicon-on-insulator and high-$\kappa$ metal gate. At the same time, to maintain acceptable performance, the supply voltage scaling has also slowed down greatly. The power per transistor still decreases, but not at a rate as fast as the transistor density increases. Therefore, the general trend is that power density of nanoscale ICs increases exponentially every technology generation.
Excessive power sets limits to the degree of integration, challenges the voltage supply network reliability and the cooling/packing system, results in excessive energy consumption and a short battery life. The flexibility in choosing alternative RTL structures during high-level synthesis is a promising way to reduce power; therefore, it is mandatory for a high-level synthesis tool to optimize for lower power.

- **Interconnect.** Problems caused by interconnects, particularly delay and congestion, are also getting worse with technology scaling. The wire resistance increases quickly when narrower and thinner metal lines are used; at the same time, interconnect capacitance tend to decrease. Overall, interconnect delay per unit length increases every technology generation. While local connections may become shorter and maintain its delay with scaling, global and semi-global connections do not often become shorter and thus can incur larger delay. Since the late 1990s, circuit performance has been increasingly determined by the delay of interconnects with technology scaling. Another related problem is routing congestion, often caused by an excessive amount of wires and relatively scarce routing resources. Despite extensive efforts for timing closure and routability enhancement in RTL synthesis, interconnect remains a big challenge in IC design.

In this dissertation we propose a spectrum of new techniques to address these challenges systematically. Some of these techniques were developed to address real-world challenges in the AutoPilot high-level synthesis tool [169], in collaboration with AutoESL Design Technologies, Inc.

### 1.4 Overview of Our Research

In this section we give a brief overview of our work as detailed in subsequent chapters of this dissertation.
Scheduling with Soft Constraints

While classic algorithms for scheduling typically focus on the tradeoffs between resource usage and performance [1, 129], they are often adapted to consider new objectives and constraints that emerge in nanoscale IC designs, typically by imposing constraints on the schedule of operations. The mechanism is often flawed in practice because many of these constraints are actually soft constraints which are not necessary; moreover, the constraint system may become inconsistent when many hard constraints are added for different purposes. In Chapter 3, we propose a scheduling algorithm that distinguishes soft constraints from hard constraints when exploring the design space. We identify a class of tractable scheduling problems with soft constraints, called difference-preference scheduling, where (1) both hard constraints and soft constraints are in the integer-difference form, and (2) the penalty function for violating each soft constraint is convex. For this model, we show that the constraint matrix in an integer-linear programming formulation is totally unimodular, and thus the problem can be solved optimally in polynomial time without expensive branch-and-bound procedures. Compared to traditional methods, the proposed scheduler allows easier expression of various design intentions and optimization directions, and, at the same time, gives the scheduler freedom to make global trade-offs optimally. We show that the formulation is useful in supporting a variety of design considerations in high-level synthesis, including register optimization, delay budgeting, operation grouping, and incremental scheduling. Thus it can be used as the main scheduling engine in a high-level synthesis tool.

Behavior-Level Observability Analysis and Operation Gating

Many techniques for power reduction in advanced RTL synthesis tools rely explicitly or implicitly on observability don’t-care conditions. In Chapter 4 we propose a systematic approach to maximize the effectiveness of these techniques by generating power-
friendly RTL descriptions in from high-level synthesis. This is done using operation gating, i.e., explicitly adding predicate to an operation based on its observability condition, so that the operation, once identified as unobservable at run time, can be avoided using RTL power-optimization techniques such as clock gating.

We introduce the concept of behavior-level observability in the context of high-level synthesis by generalizing the concept of observability don’t-care condition from Boolean networks to dataflow graphs. We then propose an efficient procedure to compute an approximated behavior-level observability of every operation in a dataflow graph. Unlike previous techniques which work at the bit level in Boolean networks, our method is able to perform analysis at the word level, and thus avoids most computation effort with a reasonable approximation. Our algorithm exploits the observability-masking nature of some Boolean operations, as well as the select operation, and allows certain forms of other knowledge to be considered for stronger observability conditions. The approximation is proven exact for (acyclic) dataflow graphs when non-Boolean operations other than select are treated as black boxes. The behavior-level observability condition obtained by our analysis can be used to guide the operation scheduler to optimize the efficiency of operation gating. To the best of our knowledge, this is the first time that behavior-level observability analysis and optimization are performed during high-level synthesis in a systematic manner. We believe that our idea can be applied to compiler transformations in general.

**Layout-Friendly High-Level Synthesis**

Most interconnect problems that occur during layout can be avoided or mitigated by adopting a better microarchitecture. Since different microarchitectures can be explored efficiently during high-level synthesis, huge opportunities for interconnect optimization are present. We address the problem of managing interconnect timing and congestion in high-level synthesis by generating a layout-friendly microarchitecture.
A metric called spreading score is proposed to evaluate the layout-friendliness of microarchitecture-level netlist structures. For a piece of connected netlist, spreading score measures how far the components can be spread from each other with bounded length for every wire. The intuition is that components in a layout-friendly netlist (e.g., a mesh) can spread over the layout region without introducing long interconnects. A semidefinite programming relaxation can be used to allow efficient estimation of spreading score. We also study another metric based on neighborhood population. The idea is that when all wires are short, for any component in the netlist, the number of components reachable within $k$ steps should be $O(n^2)$. On a number of test cases, the normalized spreading score shows a stronger bias in favor of microarchitectures that have shorter wire length after layout, compared to previous metrics like cut size and multiplexer inputs. We also justify our metric and motivate further study by relating spreading score to other metrics and problems for layout-friendly synthesis.
CHAPTER 2

Preliminaries

In this chapter we discuss the application context and the typical structure of a high-level synthesis tool. The xPilot system is used as an example. In fact, our work as described in subsequent chapters in this dissertation is implemented on the infrastructure of xPilot (or AutoPilot, a commercialized version of xPilot developed by AutoESL Design Technologies, Inc.). However, concepts and techniques proposed in this dissertation can be applied to other high-level synthesis tools as well.

2.1 System Design Using High-Level Synthesis

The need to implement complex functionalities on a single chip arises from such considerations as power, performance, cost, and reliability. Enabled by the huge silicon capacity, System-on-Chip (SoC) designs, which integrate various components in a single chip, have become a very popular design style in various embedded systems.

Figure 2.1 shows the components in a typical multimedia SoC, where microprocessors, memories, various coprocessors and accelerators are integrated in a single chip. The prevalent methodology for SoC design relies heavily on IP reuse to manage complexity. Standard components, including microprocessors, memories, some coprocessors (e.g., GPU) as well communication modules, are often designed separately and appear in the system as IP blocks. On the other hand, other blocks, like application-specific accelerators used for multimedia encoding/decoding, encryption/decryption, and signal processing, often need to implement customized or proprietary algorithms.
in order to achieve the best efficiency and to differentiate the product. In such a case, where an off-the-shelf IP block is unavailable or unsatisfactory, high-level synthesis can fit naturally in the design process by generating the required block from algorithmic specifications in high-level languages.

Figure 2.2 illustrates a typical flow to implement an application-specific accelerator from a behavioral description. Functionality of the block is described as a C program. The designer may perform simulation and refinement on the program to ensure correctness and possibly to improve the design. The behavioral model is then partitioned into the hardware part and the software part, each part possibly represented as a function in the C program. The software function is compiled using a compiler for the microprocessor, and the hardware function goes through high-level synthesis and subsequent
synthesis steps. Communication between the software program and the hardware module is often implemented using predefined interface protocols (using the IO logic in hardware and the device driver in software). System designers can explore different ways to partition the task between processor and accelerator, and in the extreme case, can choose to implement the desired functionality entirely in the accelerator.

![Diagram of Hardware/software co-design using high-level synthesis.](image)

Figure 2.2: Hardware/software co-design using high-level synthesis.

Many target applications of high-level synthesis, such as digital signal processing, involve communication between functional blocks and are driven by streams of regular data samples. Dataflow models of computation are often used for the analysis and design of such systems. A Kahn Process Network (KPN) is a dataflow model of computation where blocks are modeled as actors (processes) and actors only communicate via unidirectional first-in-first-out (FIFO) point-to-point channels [84]. The only operations an actor can perform on its associated channels are reading an input channel (which is blocking) and writing an output channel (which is required to be non-blocking by KPN). When every actor is monotonic (i.e., the sequences of tokens on its output channels depend only on the sequences of tokens on its input channels), the KPN is determinate, in the sense that the system functionality is independent of the timing of
the actors and communication channels.

The KPN model is flexible enough to model many tasks. However, KPN cannot be scheduled statically, and potentially requires unbounded buffers to ensure non-blocking write operations; thus it is hard to get an efficient and reliable implementation. A *synchronous dataflow graph* (SDF) is a restricted KPN: it requires each actor to consume/produce a fixed number of data tokens on each of its input/output channels per firing [102]. With such a restriction, the feasibility of an SDF, i.e., whether the SDF can be implemented with finite channel sizes, can be checked at compile time by investigating the topology matrix of the SDF. A static schedule of the actors can also be derived statically. SDF allows efficient implementation, and is supported in our high-level synthesis tool. While the traditional C/C++ programming languages lack the concept of actors, it is easy to add such support via synthesis directives or libraries (like in SystemC).

### 2.2 The Anatomy of xPilot

In this section we briefly describe the typical structure of a high-level synthesis tool, using xPilot as an example.

The basic synthesis flow of xPilot is illustrated in Figure 2.3. The behavioral model specified in C/C++/SystemC is first parsed into an intermediate representation, where various transformations are performed. These transformations try to improve the quality of the final implementation in various ways, e.g., removing unnecessary operations (dead code elimination), reducing the bitwidth of variables (range analysis and bitwidth optimization), replacing expensive operations with cheaper ones (operator strength reduction, constant propagation), increasing parallelism and data locality (function inlining, if-conversion, code motion, loop transformations), improving data layout (array partitioning/reshaping), extracting dataflow processes (dataflow extraction), etc. The LLVM compiler infrastructure is used in xPilot, with a lot of customized transforma-
A control/data flow graph (CDFG) is created from after code transformations. A CDFG is a graph $G = (V, E)$, where each node $v \in V$ represents an operation, and each directed edge $e \in E$ represents a data flow or a control flow.

Operation scheduling then assigns operations in the CDFG to control states. The result of scheduling is a finite-state machine with datapath (FSMD) [54], on which binding is applied to map operations to functional units, variables to storage elements (memory or registers), and data transfers to interconnects. These datapath resources can be shared in a time-multiplexed manner. After binding and state-machine encoding, an RTL model can be generated.

The synthesis engine, i.e., scheduler and binder, is the key process of generating a structural model from a behavioral model. It has vast influence on various aspects of the resulting design, and is thus the topic of interest in this dissertation.
CHAPTER 3

Scheduling with Soft Constraints

3.1 Introduction

Scheduling problems arise naturally in many fields, e.g., flight scheduling, tournament scheduling, multiprocessor scheduling, process scheduling, and instruction scheduling. The problems, even those from the same domain, often have a huge number of variants, with different application backgrounds, cost models, constraints and optimization goals. Thus, instead of developing a new algorithm for “yet another new scheduling problem,” researchers have been trying to classify problems based on their underlying characteristics. For example, a widely used classification system (described in [17]) categorizes based on machine environment, task characteristic, and objective function, resulting in a (much smaller) number of interesting abstract scheduling models. Results on the abstract scheduling models, especially those with effective polynomial-time algorithms, form the foundation of many practical schedulers in various fields.

In the context of high-level synthesis, scheduling is the process of mapping operations onto control states. Different from most other scheduling problems where the machine environments are fixed, the scheduling problem in high-level synthesis is unique in that its machine environment—that is—the target hardware generated by a high-level synthesis tool, is optimized together with the schedule. This allows a customized machine environment designed specifically for an instance of the scheduling problem, and thus leads to a more efficient hardware architecture compared to a predefined one such as a microprocessor. On the other hand, this added flexibility results in an enormous
solution space of scheduling. Almost all the important metrics for QoR, including performance (throughput, latency and clock frequency), power, area, and reliability are all significantly impacted by scheduling. Not surprisingly, scheduling has been the focus of high-level synthesis research for decades, and many approaches have been proposed in a vast literature.

Several interesting abstract scheduling models have been presented for high-level synthesis. In [160], the following models are presented.

- **Resource-constrained scheduling.** This model assumes that the number of available functional units of each type is given as a constraint, and a typical goal is to optimize performance measured by latency.

- **Time-constrained scheduling.** This model assumes that performance (latency) is given as a constraint, and a typical goal is to minimize the required number of functional units.

- **Time-and-resource-constrained scheduling.** This is a combination of the above two, where both resource constraints and performance constraints are given.

Common to these models are dependency constraints (or precedence constraints in general) and the heterogeneous machine model.

List scheduling, a technique originally used for compilation targeting microprocessors [1], is widely employed for resource-constrained scheduling. A list scheduler processes operations in a control-step by control-step fashion. It maintains a list of “ready” operations whose predecessors have all been scheduled, and visits each operation in the list in some heuristic order. If a functional unit that matches the current operation type is unoccupied, the operation is scheduled at current step, and the “ready” list is updated. The algorithm proceeds to the next step when no more operations can be scheduled in the current step.

Force-directed scheduling [129] is an algorithm designed primarily to solve the
time-constrained scheduling problem for resource minimization. It maintains the feasible schedule range for each operation according to dependency constraints and performance requirements, and estimates the required number of functional units by building distribution graphs based on the feasible schedule ranges. In each iteration, the force (indicating the estimated resource increase) of scheduling each operation to each of its feasible steps is evaluated, and the operation-step pair with smallest force is committed. The force is calculated based on an analog to the effect of a spring; this tends to distribute operations evenly among the steps.

In addition to list scheduling and force-directed scheduling, other methods have also been proposed. Exact approaches include integer-linear programming [56, 76], branch-and-bound search [138], constraint programming [94, 95], and other formal methods using BDD [133] or finite automata [71, 72]. Metaheuristics such as simulated annealing, genetic algorithms and ant colony optimization [161] can also be applied.

While the most basic forms of available scheduling models (e.g., resource-constrained latency minimization for acyclic dataflow graphs where each operation takes unit time) are well studied, variants of the problems arise in practice when deviations occur in constraints, optimization goals, or the models for evaluating quality of results. For example, the assumption that each operation takes an integer number of clock cycles is invalid on many target architectures of high-level synthesis nowadays. Multiple operations with data dependencies can be executed in the same cycle, and this is often referred to as chaining. Those variants are often solved using the same algorithm as that for the basic form of the problem, often by preprocessing and introducing additional constraints. Take operation chaining as an example: Suppose Operation A uses the result of Operation B, and the overall delay of the two operations exceeds the clock period, an additional precedence constraint that the starting step of operation B is at least 1 cycle after the finishing step of Operation A. This constraint is similar to the dependency constraint, and can be handled by most existing algorithms. Many other variants of the scheduling problem can be handled similarly by introducing additional
constraints [107, 117]; however, the approach has several limitations.

1. Inconsistency can occur in the constraint system. When many constraints are added for different purposes, they will probably contradict each other. For example, scheduling two operations in a certain order will lead to opportunities for efficient clock gating, but the order may conflict with other requirements like performance in less obvious ways (discussed in Chapter 4). When inconsistency happens, many scheduling algorithms will either fail or disregard some constraints completely.

2. Design space is often not well characterized with the artificially added constraints. While these constraints eliminate infeasible/inferior solutions from the search space, they can also forbid the exploration of some feasible and possibly good solutions due to low accuracy of estimation models at such a high level. For the example of operation chaining, if the estimated overall delay for two operations in a combinational path is 10.5 ns when the required cycle time is 10 ns, a simple approach would forbid the chaining of the two operations. However, it is probable that a solution with a slight nominal timing violation can still meet the frequency requirement after various timing optimization procedures in later design stages, like logic refactoring, retiming, threshold voltage assignment, gate/wire sizing, placement/routing, etc. In this case, artificial constraints eliminate the possibility of improving other aspects of the design with some reasonable estimated violations. Some may argue that when the timing analyzer is aggressive enough by taking into account all possible subsequent timing optimizations, the added constraints will not restrict the design space. However, in such a case, we may miss some constraints that are indeed necessary.

3. Many abstract scheduling models are intrinsically difficult to solve. Most of the problems involving dependency and resource constraint/objective are proven NP-hard [17]. Known exact methods like integer-linear programming are not scalable
to moderately large designs. Heuristics like list scheduling have some approximation ratio for only a limited category of problems (e.g., makespan minimization with identical machines and identical tasks can be solved by list scheduling within a factor of 2 [62]). For many other variants of the problem, such as those with multiple types of functional units, with consideration of registers, no result is known about the optimality of the algorithms. While algorithms like list scheduling or force-directed scheduling work well in many practical instances of the problem, their effectiveness is questionable on other variants of the problem, especially when a lot of artificial constraints are introduced to allow the application of the algorithms.

Both Items 1 and 2 stem from the rigid way in which constraints are handled. So, we take a different approach. We notice that many of these constraints are actually preferences (also referred to as soft constraints) rather than essential requirements (referred to as hard constraints). Unlike hard constraints, soft constraints are supposed to be followed when possible, but not necessarily. Thus, soft constraints will neither limit the design space nor cause inconsistencies even if conflicting soft constraints are present.

The conceptual structure of a scheduler with support for soft constraints is illustrated in Figure 3.1. The scheduler accepts a functional specification and a number of QoR targets along with characterizations of the target platform. A number of hard/soft constraint generators produce hard/soft constraints based on the specifications. Hard constraint generators usually impose essential requirements of the design, like dependency between operations or performance requirement. A typical soft constraint generator considers only one aspect of the design, and formulates some favorable situations as soft constraints. The cost of violating a soft constraint can also be specified. Note that since conflicting soft constraints are allowed, a soft constraint generator does not need to consider trade-offs with other design requirements or global feasibility of the constraint system. This makes it very easy to design a soft constraint generator and al-
lows more accurate estimation models to be used. It is also possible to expose the inter-
face of a soft constraint generator to the designer, so that the designer can gain detailed
control over part of the design by manually adding soft constraints when necessary,
without worrying about problem feasibility. The optimization engine then accepts all
hard/soft constraints, as well as the optimization goal, and makes intelligent decisions
to obtain the solution. The process can possibly be iterative by allowing adjustments
on soft constraints after an initial result is available.

![Figure 3.1: Structure of a scheduler with soft constraints.](image)

Although the concept of soft constraints makes it easier to express various design
intentions and to guide the scheduler, it brings major challenges for the implementation
of the optimization engine. Classical algorithms like list scheduling and force-directed
scheduling are not designed to work with soft constraints. Some scheduling algorithms,
like the iterative modulo scheduling for software pipelining [136], adopt iterative search
strategies and allow earlier decisions to be adjusted later, so that they are able to work
on highly constrained problems. However, we are not aware of any algorithm in this
category that explicitly supports soft constraints. The soft scheduling algorithm [171],
despite the name, actually adds hard constraints iteratively. Similar approaches in-
clue [25, 117]. Exact methods for scheduling with soft constraints have been developed in the artificial intelligence community using branch and bound [118], integer-linear programming [156], or constraint satisfaction [147]. This class of methods works well for planning and temporal reasoning problems with very complex hard/soft constraints on a small number of tasks; however, exponential time complexity prohibits the application of these methods to problems of a practical size in high-level synthesis.

In this chapter we identify a class of scheduling problems with soft constraints that can be solved optimally in polynomial time, and investigate its applications in high-level synthesis. The contribution is the following.

- We propose to perform scheduling guided by soft constraints in high-level synthesis. This method enables a methodological change on how the direction of optimization can be specified and how design space can be explored. To the best of knowledge, this is the first systematic way to formulate and support scheduling with soft constraints in the high-level synthesis domain.

- We identify a class of soft constraints that can be solved in polynomial time with guaranteed optimality. This leads to a new abstract scheduling model for high-level synthesis, referred to as difference-preference scheduling. A mathematical-programming formulation is proposed to solve the problem with proven optimality. Techniques to handle variants of the formulation are also proposed.

- We show that the difference-preference scheduling model is useful in high-level synthesis. In particular, we discuss its modeling ability for register optimization, operation grouping, time budgeting, incremental synthesis, and operation ordering (in Section 4.4).
3.2 Problem Formulation

Recall that in a CDFG $G = (V, E)$, each node $v \in V$ represents an operation and each directed edge $e \in E$ represents data or control dependency between operations. For each operation $v$, an integer-valued scheduling variable $s_v$ is introduced to represent the time slot in which operation $v$ is performed. The FSMD model can be constructed once the scheduling variable for every operation is decided [36]. The task of scheduling is thus to decide $s_v$ for every operation $v$.

Using the scheduling variables described above, the system of difference constraints (SDC) formulation was proposed to solve scheduling problems [36]. The method uses a special form of constraints in a linear program, so that it can avoid expensive branch and bound procedures in traditional ILP formulations for scheduling [56,76] while still optimizing globally. SDC is flexible enough to model a wide variety of constraints and objectives in high-level synthesis, and is extended in [83] to solve a time budgeting problem.

A special class of constraints, called integer-difference constraints, is used to model various design constraints in SDC.

**Definition 3.1** (integer-difference constraint). An integer-difference constraint is a constraint of the form $s_u - s_v \leq d$, where $d$ is a constant integer.

Using a system of integer-difference constraints, [36] is able to model dependency constraints and timing constraints precisely, and model resource constraints heuristically. The advantage of integer-difference constraints is that they can be solved very efficiently using linear programming, without expensive branch-and-bound procedures required in most ILP formulations.

In this chapter we show that the “soft version” of an integer-difference constraint, referred to as an integer-difference soft constraint, can also be handled efficiently in the optimization engine.
**Definition 3.2** (integer-difference soft constraint). An integer-difference soft constraint is a soft constraint in the form of \( s_u - s_v \leq d \), where \( d \) is a constant integer.

The scheduling problem with both hard and soft integer-difference constraints can be written as:

\[
\begin{align*}
\text{min} & \quad c^T s & \text{(linear objective)} \\
\text{s.t.} & \quad s_{u_i} - s_{v_i} \leq p_i, \quad i = 1, \ldots, m, & \text{(hard constraints)} \\
& \quad s_{k_j} - s_{l_j} \leq q_j, \quad j = 1, \ldots, n, & \text{(soft constraints)}
\end{align*}
\]

(3.1)

Here \( u_i, v_i, k_j \) and \( l_j \) are nodes in the CDFG.

### 3.3 Handling Soft Constraints

In this section we describe our method for handling soft constraints using a penalty method in a mathematical programming formulation. Our method extends the approach in SDC by supporting integer-difference soft constraints, while still guaranteeing efficient and optimal solutions.

#### 3.3.1 A Penalty Method for Soft Constraints

To ease discussion, we can write the formulation in Equation 3.1 in vector and matrix form as

\[
\begin{align*}
\text{min} & \quad c^T s & \text{linear objective} \\
\text{s.t.} & \quad Gs \leq p & \text{(hard constraints)} \\
& \quad Hs \leq q & \text{(soft constraints)}
\end{align*}
\]

(3.2)

Let \( H_j \) be the \( j \)th row of \( H \). Since a soft constraint may be violated, for each soft constraint \( H_j s \leq q_j \), we introduce a violation variable \( v_j \) to denote the amount of violation. Then the soft constraint can be transformed to two traditional constraints as

\[
\begin{align*}
H_j s - v_j & \leq q_j, \\
-v_j & \leq 0.
\end{align*}
\]

(3.3)
There is usually a cost when a soft constraint \( H_j s \leq q_j \) is violated. We first consider a simple case where the total cost is separable; that is, the cost for violating a soft constraint depends only on the amount of violation, and when multiple soft constraints are violated, the overall cost is the sum of the corresponding costs for all the violated soft constraints. Thus, a penalty term depending on the amount of violation, \( \phi_j(v_j) \), can be added in the objective function to denote the cost for violating the \( j \)th soft constraint. Then the problem is formulated as a traditional form of mathematical programming as follows.

\[
\min \quad c^T s + \sum_{j=1}^{n} \phi_j(v_j) \\
\text{s.t.} \quad G s \leq p \\
H s - v \leq q \\
- v \leq 0.
\]

(3.4)

The constraints can also be written in matrix form as

\[
\begin{bmatrix} G & O \\ H & -I \\ O & -I \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} \leq \begin{bmatrix} p \\ q \\ 0 \end{bmatrix}.
\]

(3.5)

3.3.2 Total Unimodularity and Implications

In the SDC formulation, total unimodularity is exploited to avoid branch and bound procedures while still guaranteeing integral solutions.

**Definition 3.3** (total unimodularity). A matrix \( A \) is totally unimodular if every square submatrix of \( A \) has a determinant either 0, 1 or -1.

Clearly, a totally unimodular matrix can only have entries 0, 1, or -1. Total unimodularity plays an important role in combinatorial optimization; this is due to the result in Lemma 3.1.

**Lemma 3.1** (Hoffman and Kruskal [74]). If \( A \) is totally unimodular and \( b \) is a vector of integers, every extreme point of polyhedron \( \{ x : Ax \leq b \} \) is integral.
Lemma 3.1 implies that an integer linear programming problem

\[
\min \ c^T x \\
\text{s.t.} \ Ax \leq b \\
x \in \mathbb{Z}^n
\]

can be solved by solving a linear-programming relaxation

\[
\min \ c^T x \\
\text{s.t.} \ Ax \leq b
\]

if \( A \) is totally unimodular and \( b \) is integral, because the solution of the linear program can always be found at an extreme point of the polyhedron \( \{ x : Ax \leq b \} \).

In [36], the following result is presented.

**Lemma 3.2** (Cong and Zhang [36]). With only integer-difference constraints, the constraint matrix of the scheduling problem is totally unimodular.

By exploiting the total unimodularity, the SDC formulation of scheduling can be efficiently solved using linear programming. Integral solutions are guaranteed without the expensive branch-and-bound procedure in typical integer-linear programming solvers.

The following lemma is useful for showing the total unimodularity of the constraint matrix in Equation 3.4.

**Lemma 3.3** (Raghavachari, [134]). If a \( m \times n \) matrix \( A \in \{-1, 0, 1\}^{m \times n} \) has a row (or a column) with at most one nonzero element, \( A \) is totally unimodular if and only if the resulting matrix, after removing the row (or column) from \( A \), is totally unimodular.

**Theorem 3.1.** With only integer-difference constraints and integer-difference soft constraints, the constraint matrix in the optimization program in Equation 3.4 is totally unimodular.
Proof. Using Lemma 3.3, we can first remove the last \( n \) rows of the matrix in Equation 3.5, and then remove the last \( n \) columns of the resulting matrix, without changing total unimodularity.

\[
\begin{bmatrix}
G & O \\
H & -I \\
O & -I
\end{bmatrix} \Rightarrow \begin{bmatrix}
G & O \\
H & -I
\end{bmatrix} \Rightarrow \begin{bmatrix}
G \\
H
\end{bmatrix}.
\]

Then we only need to show that \( \begin{bmatrix}
G \\
H
\end{bmatrix} \) is totally unimodular. Note that each row of the matrix represents the difference of two variables, so the matrix is in the same form as the constraint matrix in the SDC formulation, which is thus totally unimodular according to Lemma 3.2.

Actually, by using Lemma 3.3, it is easy to see that unary constraints in the form of \( s_i \leq d, d \in \mathbb{Z} \), either hard or soft, can be added to the problem in Equation 3.2 without destroying the total unimodularity of the constraint matrix. Constraints in this form are useful for specifying the latency bound, a prefixed schedule for an operation, etc., as discussed later in this chapter.

### 3.3.3 Form of the Penalty Term

#### 3.3.3.1 Convex Separable Penalty

The penalty term for violation in the objective function, \( \phi_j(v_j) \), is supposed to model the cost of violating the \( j \)th soft constraint. Typically, \( \phi_j \) is 0 when there is no violation, and it is a non-decreasing function of \( v_j \). That is,

\[
\phi_j(v_j) = 0, \text{ when } v_j = 0,
\]

\[
\phi_j(v_j) \geq \phi_j(v'_j) \geq 0, \text{ when } v_j \geq v'_j \geq 0.
\]

Note that it is impossible to have a negative \( v_j \) because the constraints in our formulation forbid this (Equation 3.3).
If the cost is linear to the amount of violation for each soft constraint, i.e.,

\[ \phi_j(v_j) = \alpha_j v_j, \] (3.8)

where \( \alpha_j \) is a nonnegative constant, the problem in Equation 3.4 is then a linear program with a totally unimodular constraint matrix and integral right-hand side. According to Lemma 3.1, we can get an optimal integral solution efficiently.

In fact, our formulation allows \( \phi_j(v_j) \) to be a nonlinear convex function of \( v_j \) without introducing significant complexity based on the following result.

**Lemma 3.4** (Hochbaum and Shanthikumar [73]). For the integer convex separable optimization problem

\[
\begin{align*}
\min & \quad \sum_{i=1}^{n} f_i(x) \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \in \mathbb{Z}^n,
\end{align*}
\] (3.9)

where \( f_i \) is convex, \( A \) is totally unimodular and \( b \) is integral, the optimal solution can be found in polynomial time.

It is shown in [116] that the problem in Equation 3.9 can be translated into a linear program by piece-wise linearizing \( f_i \) at integral points and removing the integral constraints. The resulting linear program, if feasible, is guaranteed to have an integral solution. Therefore, we have the following theorem.

**Theorem 3.2.** When only integer-difference constraints and integer-difference soft constraints are used, and every \( \phi_j \) is convex, the problem can be solved optimally with an integral solution in polynomial time.

Theorem 3.2 allows a rich category of penalty functions to be used in our formulation, including linear, quadratic, exponential and log barrier functions, as illustrated in Figure 3.2. The selection of the penalty form and its parameters can be performed on each individual soft constraint based on characteristics of the constraint. For some
cases, the simple linear function models the cost precisely; for some others, the cost increases exponentially with the amount of violation. Sometimes the log-barrier form is also useful to keep the amount of violation within a certain range.

Figure 3.2: Some typical penalty functions. An offset is added to the exponential penalty to satisfy Equation 3.6. The binary penalty is nonconvex.

### 3.3.3.2 Nonconvex Binary Penalty

Notably, although the cost of violating a soft constraint is often convex with respect to the amount of violation, it is not always the case. For example, the cost for violating some soft constraints can be a constant for any amount of non-zero violation (denoted as the binary penalty function). Let $b(x)$ be the binary penalty function with unit coefficient, defined as

$$
b(x) = \begin{cases} 
1 & \text{if } x > 0, \\
0 & \text{if } x \leq 0.
\end{cases}
$$

Unfortunately, introducing such a penalty function directly to the formulation in Equation 3.4 makes it difficult to solve. In fact, we can show that the problem is NP-hard.
Theorem 3.3. With binary penalty functions, the problem in Equation 3.4 is NP-hard.

Proof. Here we formulate the problem of minimum feedback arc set using integer-difference soft constraints and binary penalty functions. Note that the decision version of minimum feedback arc set is among Karp’s 21 NP-complete problems [85].

The reduction process is the following. For a directed graph $G = \{V, E\}$, where $E \subseteq (V \times V)$ is the set of directed edges, the minimum feedback arc set problem asks for a minimum set of edges whose removal leaves the remaining graph acyclic. Since a direct acyclic graph defines a partial ordering of its nodes, we can assign a label $l_j$ to every node $v_j$, so that for each edge $e_i = (v_{src}, v_{dst}) \in E$, we have

$$l_{src} - l_{dst} \leq -1.$$  \hspace{1cm} (3.11)

For the directed graph $G$, we can add soft constraints as in Equation 3.11 for every edge. If $G$ is acyclic, all these soft constraints can be satisfied; otherwise, some of these constraints will be violated. The minimum feedback arc set problem asks us to find the minimal set of soft constraints to violate. The objective function can be modeled as the sum of binary functions over the violation variable for all soft constraints. We can then reach the following formulation.

$$\min \sum_{i=1}^{\mid E \mid} b(v_i)$$

s.t. $l_{src} - l_{dst} - v_i \leq -1 \quad i = 0, 1, \ldots, \mid E \mid$

$$-v_i \leq 0 \quad i = 0, 1, \ldots, \mid E \mid$$

Here, $b(v)$ is the binary function defined in Eqn. 3.10. This reduction shows that the problem with the binary penalty function is at least as hard as the minimum feedback arc set problem, and is thus NP-hard. \hfill \Box

Instead of trying to obtain the optimal solution, we take a different approach to get an approximation. We first check every soft constraint with binary penalty and eliminate the constraint if it obviously conflicts with the existing hard constraints. For the remaining soft constraints, we iteratively solve a sequence of subproblems with
slightly different convex penalty functions to gradually approximate the binary penalty. This is motivated by the sparsity optimization techniques used in signal processing [21, 155], as well as the method for linear wire length optimization using a quadratic model in the GordianL placer [145]. At iteration $k$, we use a linear function $b^{(k)}(x)$ to approximate $b(x)$ near $x^{(k-1)}$ ($x^{(k-1)}$ denotes the value of $x$ in the solution of the previous iteration).

$$b^{(k)}(x) = \frac{x}{\max(1, x^{(k-1)})}$$  \hspace{1cm} (3.12)

The idea is that when the process converges after $k - 1$ iterations, we will have $x^{(k)} = x^{(k-1)}$, and $x^{(k)}$ is a nonnegative integer; thus $b^{(k)}(x^{(k)}) = b(x^{(k)})$. For an intermediate iteration, when the violation is larger than one, the coefficient in the penalty term is scaled down in the next iteration, effectively reducing the effort to satisfy the corresponding soft constraint. In the extreme case where the violation is very large, the optimization engine will tend to ignore the soft constraint. Note that this iterative reweighting method does not guarantee the optimality of the final solution. Nevertheless, similar techniques are shown to perform well in signal processing [21]; our experimental evaluation also confirms that it is a good heuristic.

### 3.3.3.3 Penalty on the Maximum Violation

In the above discussion, we have assumed that the cost is separable; i.e., the overall cost for violating multiple soft constraints is the sum of the cost for violating each individual soft constraint. However, this does not cover all practical situations. A typical exception is the case where a group of soft constraints are imposed for the same intention and the overall cost is decided by the maximum violation among all the soft constraints in the group. An example is the case for register lifetime minimization, where the lifetime of a variable is determined by its last user. Consider a group of soft constraints $G = \{g_1, g_2, \ldots\} \subseteq \{1, \ldots, n\}$, the overall cost for $G$ is

$$\text{cost}_G = \phi_G(\max_{j \in G} v_j),$$  \hspace{1cm} (3.13)
instead of
\[
\text{cost}_G = \sum_{j \in G} \phi_j(v_j).
\] (3.14)

The objective function of the form in Equation 3.13 is not separable when \(|G| > 1\), and the linearization method in [116] cannot be applied.

The \(\max\) operation on violation variables can be handled by a reformulation technique commonly used in convex programming [16]. We introduce auxiliary variable \(v_G\) to measure the maximum violation in the group directly. For each \(j \in G\), we replace \(v_j\) with \(v_G\) in Equation 3.3 to get the following constraints.

\[
\begin{align*}
H_j s & -v_G \leq q_j, \\
-v_G & \leq 0.
\end{align*}
\] (3.15)

Clearly, \(v_G \geq v_j, \forall j \in G\). We add \(\phi_G(v_G)\) to the objective function. Since \(\phi_G\) is a univariate penalty function satisfying requirements in Equation 3.6 and 3.7, the optimization solver will tend to minimize \(v_G\), and when the optimal solution is achieved, we can have \(v_G = \max_{j \in G} v_j\).

After the above reformulation, we get a separable objective, however, the structure of the constraint matrix is changed, because \(v_G\) appears in multiple constraints. The resulting constraint matrix for the linear program is

\[
\begin{bmatrix}
G & 0 \\
H & -P \\
O & -I
\end{bmatrix}.
\] (3.16)

Here \(P\) is no longer an identity matrix (it is not even a square matrix as each \(v_G\) appears in multiple constraints). Thus, the result in Lemma 3.3 is not applicable.

In the following, we show that the constraint matrix is still totally unimodular if the group of soft constraints conforms a special structure.

**Definition 3.4** (anchoring variable). *In a group of integer-difference soft constraints \(G\), an anchoring variable \(a_G\) is a scheduling variable that occurs in every soft constraint in the group with the same coefficient, either 1 or -1.*
For example, the group of soft constraints

\[
\begin{align*}
  s_1 - s_2 &\leq -1 \\
  s_1 - s_3 &\leq -2 \\
  s_1 - s_4 &\leq 0
\end{align*}
\]

(3.17)

has an anchoring variable \( s_1 \), while the group of soft constraints

\[
\begin{align*}
  s_1 - s_2 &\leq -1 \\
  s_2 - s_3 &\leq -2
\end{align*}
\]

(3.18)

does not have an anchoring variable. The operation (or node) corresponding to the anchoring variable is called anchoring operation (or anchoring node in the CDFG).

The following result is useful for determining if a general matrix is totally unimodular.

**Lemma 3.5** (Ghouila-Houri, [59]). A is totally unimodular if and only if each subset \( J \) of the columns can be partitioned into two subsets \( K \) and \( J - K \), such that for each row \( i \) we have

\[
\left| \sum_{j \in K} A_{ij} - \sum_{j \in J - K} A_{ij} \right| \leq 1.
\]

Using this lemma, we can prove that the constraint matrix can still be totally unimodular.

**Theorem 3.4.** For a problem with integer-difference constraints and integer-difference soft constraints, when the reformulation technique in Equation 3.15 is used to handle the max operation on violation variables, if each group of soft constraints (which are penalized based on the maximum violation) have an anchoring variable, the resulting constraint matrix in Equation 3.16 is totally unimodular.

**Proof.** Using Lemma 3.3, we do not need to consider rows with only one nonzero element, so we will only show that the matrix

\[
\begin{bmatrix}
  G & O \\
  H & -P
\end{bmatrix}
\]

is totally unimodular.
Note that after the reformulation to handle \( \max \) operation, each row has at most one violation variable.\(^1\) Thus, the rows of this matrix can be divided into non-overlapping groups \( \{G_0, G_1, \ldots\} \), so that constraints in \( G_0 \) do not involve any violation variable (hard constraints), and \( G_i \) (\( i = 1, \ldots \)) contains all the constraints that share a maximum violation variable \( v_{G_i} \).

Let \( S \) denote the set of columns in the matrix corresponding to scheduling variables (columns covering submatrices \( G \) and \( H \)), and let \( V \) denote the set of the other columns (corresponding to violation variables), with its subset \( V_+ \) denoting those corresponding to soft constraint groups whose anchoring variables have coefficient 1, and \( V_- \) denoting those corresponding to soft constraint groups whose anchoring variables have coefficient -1.

Given any subset of columns \( J \), we construct a subset \( K \) of \( J \) by adding columns in \( J \cap V \) to \( J \cap S \). Let \( A(v) \) denote the anchoring variable of the soft constraint group corresponding to \( v \). We construct

\[
K = J \cap S + \{v | ((v \in V_+ \cap J) \wedge A(v) \in J) \lor ((v \in V_- \cap J) \wedge A(v) \notin J) \}
\]

The basic idea of the above construction is that all scheduling variables covered by \( J \) should be in one subset \( K \); if an anchoring variable with coefficient 1 and its corresponding violation variable are both covered by \( J \), the violation variable should be added in \( K \), so that their coefficients cancel out each other in the sum; if an anchoring variable with coefficient -1 and its corresponding violation variable are both covered by \( J \), they should be in different subsets, so that their coefficients cancel out in the subtraction.

We can verify that with the partition of \( J \) into \( K \) and \( J - K \), for each row \( r \) in

\(^1\)Even if the violation variable of a soft constraint appears in multiple \( \max \) functions in the objective, distinct constraints will be introduced for each group.
We consider the following cases.

1. If \( r \) corresponds to a hard constraint (i.e., the coefficient is 0 for any violation variable), we have \( \sum_{j \in J - K} r_j = 0 \). The coefficients of the two scheduling variables involved are 1 and -1. If none of them are covered by \( J \), or both of them are covered by \( J \), the left-hand side of Equation 3.19 is 0; if exactly one of them is covered by \( J \), the left-hand side of Equation 3.19 is 1.

2. If \( r \) corresponds to a soft constraint that has a violation variable \( v_r \notin J \), the left-hand side of Equation 3.19 can take 1, -1, or 0, depending on how the two scheduling variables involved are covered by \( J \), as in the case of the hard constraint.

3. If \( r \) corresponds to a soft constraint that has a violation variable \( v_r \in J \cap V_+ \), when \( A(v_r) \in J \) (and thus \( A(v_r) \in K \)), we have \( v_r \in K \), and \( \sum_{j \in J - K} r_j = 0 \); when \( A(v_r) \notin J \), we have \( v_r \in J - K \), and \( \sum_{j \in J - K} r_j = -1 \). In both cases, the left-hand-side of Equation 3.19 can be either -1 (when the scheduling variable with coefficient -1 is in \( J \)) or 0 (otherwise).

4. If \( r \) corresponds to a soft constraint that has a violation variable \( v_r \in J \cap V_- \), when \( A(v_r) \in J \) (and thus \( A(v_r) \in K \)), we have \( v_r \in J - K \), and thus \( \sum_{j \in J - K} r_j = -1 \). The left-hand side of Equation 3.19 is either 0 (when the scheduling variable with coefficient 1 is in \( J \)) or -1 (otherwise). When \( A(v_r) \notin J \), we have \( v_r \in K \), and thus \( \sum_{j \in J - K} r_j = 0 \). The left-hand side of Equation 3.19 is either 1 (when the scheduling variable with coefficient 1 is in \( J \)) or 0 (otherwise).

All the above cases satisfy Equation 3.19.
3.3.4 Overall Flow

Since our optimization engine is highly efficient because it takes advantage of the special constraint structure, it is feasible to run the optimization engine repeatedly in an iterative manner. After each iteration, we can adjust the soft constraints and solve again for better QoR. There are many ways soft constraints can be adjusted: a soft constraint can be added or deleted, and the penalty function of a soft constraint can be changed (e.g., the technique discussed in Subsection 3.3.3 that deals with binary penalty). The detailed strategy for adjusting should be designed considering characteristics of the problem.

When various soft constraints designed for different aspects of QoR are present, it is important that the penalty terms be carefully designed to allow a reasonable tradeoff. Orthogonal to the selection of the form of the penalty function (among which are linear, quadratic, exponential), the coefficient (weight) in the penalty term can always be adjusted in the objective function. Note that the cost of violating a soft constraint can usually be interpreted as overhead in certain QoR metrics like performance/area/power. Thus, adjusting the coefficients for different types of soft constraints enables different tradeoffs. For example, if tradeoffs between power and area are desirable, we may want to select the weights so that the penalty of a 10% power increase is comparable with that of a 10% area increase. To do this for a soft constraint considering area, we can normalize the area penalty for unit violation ($\delta_A$) to the estimated total area ($A$), and multiply it by a manually selected constant $\alpha$. Then the penalty term is

$$\phi(v) = \alpha \frac{\delta_A}{A} p(v),$$

where $p(v)$ is a convex function as discussed in Subsection 3.3.3. The same can be done for a soft constraint for power consideration. In practice, a robust approach is to run the optimization multiple times with different combinations of weights to get a sequence of Pareto optimal solutions—and then the designer can select a subset of solutions to go through later design processes.

40
3.4 Example Applications

In this section we provide examples of soft constraint generators that are designed for various intentions in the proposed framework.

3.4.1 Register Optimization

3.4.1.1 Register Reduction

When a variable is generated in one state and used in another, its value usually needs to be stored in one or more storage elements (typically registers). Reducing the number of registers is usually effective in reducing both area and power.

Consider an operation $u$ whose result is used by operations $v_1, v_2, \ldots, v_m$. If all of these operations can be scheduled in the same control step as $u$, we do not need to store the result of $u$ in a register. We can then get a soft constraint to express this preference as

$$s_{v_i} - s_u \leq 0, \forall i \in \{1, 2, \ldots, m\}.$$ (3.20)

Note that another soft constraint $s_u - s_{v_i} \leq 0$ is unnecessary here because it is already added as a hard constraint for data dependency.

It is important to note that the effort to reduce the number of registers is performed when hard constraints regarding clock frequency are already satisfied. A hard constraint is added to separate operations $u$ and $v$ when chaining them together would lead to a path with an estimated delay exceeding the target clock period, as described in [36].

For each operation, we can have a group of soft constraints in the form of Equation 3.20, and relate the cost to the maximum violation in the group. The cost function with regard to the maximum violation can either be a binary function, or a linear one. A formulation using the linear penalty is not just an approximation, because it effectively optimizes the total lifetime of the variables, which is also a meaningful optimization.
objective considering pipelining and register sharing.

In a previous work by Beidas, Mong and Zhu [8], a method to optimize register pressure is proposed. The basic observation of [8] is similar to the one mentioned here; yet their model for register lifetime is inaccurate because they effectively evaluate the cost for violating different soft constraints independently, without considering correlations within a group of soft constraints. When a value is used by multiple downstream operations, the lifetime of the value is decided by the schedule of its last user, and thus the maximum violation among the group of soft constraints (shown in Equation 3.20) for avoiding the register should be used in the penalty. In other words, if any of the constraints are violated by a large amount, the lifetime of the variable will be large, and the register cannot be avoided by meeting other soft constraints in the same constraint group. Thus, compared to [8], our approach is more general and allows more accurate modeling.

3.4.1.2 Register Insertion

While we often want to reduce the number of registers, there are cases where an additional level of register is useful. For example, in order to accommodate long interconnect delay and ease layout, a designer may want to register the input of the module, or the input from a memory block when the throughput/latency target permits doing so. For this, we can create a soft constraint that try to separate a “read” operation with all of its successors. That is, if the result of operation $u$ is preferably registered, and the result is used by $v_1, v_2, \ldots, v_m$, we have soft constraints

$$s_u - s_{v_i} \leq -1, \forall i \in \{1, 2, \ldots, m\}.$$  \hspace{1cm} (3.21)

We can either penalize on the maximum violation or each individual one. Since the soft constraints in Equation 3.21 can be violated by at most one, a linear penalty is effectively the same as a binary penalty here.
3.4.2 Delay Budgeting

Some schedulers assume that the number of clock cycles needed to complete an operation is known before scheduling. Essentially, the assumption is that the type of functional unit used to implement the operation is decided before scheduling. While this makes the scheduling problem similar to that in software compilation and thus allows adoption of some classical instruction scheduling techniques used in software compilers, the practice of module selection before scheduling is likely to result in suboptimal solutions in high-level synthesis. This is because an operation can often be implemented in less expensive ways (in terms of area, power, reliability, etc.) when the delay budget is increased, but it is often unclear how much slack is available for the operation without scheduling information.

The same applies when a group of operations (often operations that form a submodule) are considered together. When scheduling operations in a module, the submodule can be regarded as a single operation. If we assume that the implementation of a submodule is fixed when scheduling the top module, we lose the opportunity to explore alternative implementations for better quality of results.

Such problems call for a synthesis flow where flexibility in choosing the right implementation is preserved during scheduling. Under a given performance target, the process of deciding the latency of operations (or operation groups) is referred to as delay budgeting. Some of the above problems, particularly those related to single operations, have been addressed in a number of previous research efforts, using techniques like iterative refinement [78, 149], integer-linear programming [79], network-flow formulation [57, 152] and convex programming [83]. Here we show that the delay budgeting problem can be handled using soft constraints.

To get a unified description, we use the term “region” to refer to an operation or a group of operations. We introduce two additional scheduling variables: \( s_{\text{begin}} \) denotes the step when the region is scheduled to start execution, and \( s_{\text{end}} \) denotes the step...
when the region is scheduled to finish execution. There are multiple implementations of the region, each with different latency and cost. Here, consider the case where the cost function with respect to latency is convex. This is common in practice due to diminishing returns when increasing the latency budget. Figure 3.3 shows a typical tradeoff curve between area and latency. In this case, when the latency budget is greater than or equal to 5, we get the implementation with minimum cost. Thus we add a soft constraint

\[ s_{\text{begin}} - s_{\text{end}} \leq -5. \]  

Since Equation 3.22 is an integer-difference soft constraint, the constraint system is still totally unimodular. With the violation variable \( v \), the constraint becomes

\[ s_{\text{begin}} - s_{\text{end}} - v \leq -5. \]

Here \( v \) can only take a value in \{0, 1, 2, 3\}, and it is easy to see the associated penalty function \( \phi(0) = 0, \phi(1) = 100, \phi(2) = 300, \phi(3) = 700 \). The penalty function is illustrated in Figure 3.4. With this piecewise-linear convex penalty, the problem can be solved optimally according to Lemma 3.4. In the case where the penalty is not convex, the convex hull can be used in an iterative approximation procedure.
It should be noted that when applied to the problem of time budgeting for individual operations, our approach is equivalent to a few previous approaches. [83] introduced a mathematical programming formulation which has a similar form to ours. [57] proposed a formulation using min-cost network flow; and the same technique is used in [152] for a voltage assignment problem. The min-cost flow problem is equivalent to the dual of our formulation with convex penalty functions. Compared to the existing work, our framework using soft constraints is more generic, and it allows multiple types of hard/soft constraints to be added in more flexible ways.

### 3.4.3 Operation Grouping

Many subsequent optimizations in logic synthesis and mapping can only happen when a group of operations are scheduled to match a certain pattern. This is referred to as *operation grouping*. Here we give a few examples.

- Mapping to DSP blocks. Many FPGA devices contain dedicated DSP blocks that are capable of implementing some groups of operations commonly used in DSP
applications, e.g., multiply-accumulate. These DSP blocks are much more efficient than their general-purpose counterparts (such as SRAM-based look-up tables) in terms of power, area, and performance. Figure 3.4.3 illustrates a DSP48A slice in a Xilinx Spartan-3A DSP FPGA. There are five major ports, labeled A, B, D (18-bit inputs), C (48-bit input), and P (48-bit output). The DSP48A slice is capable of implementing the following family of arithmetic functions,

\[ P = C \pm A \times (D \pm B). \] (3.23)

Two conditions must be satisfied to allow efficient mapping: (1) the group of operations need to match the functionality and the bitwidths of the DSP block; (2) the operations need to be scheduled to match the register structure in the DSP block.

• Mapping to SIMD units. It is recognized that SIMD units (like vector addition/subtraction/multiplication) often lead to better area and power efficiency when used in high-level synthesis [135]. This is not only because SIMD units are implemented by taking advantage of the datapath regularity, but also because better resource sharing can be achieved when large groups of operations are considered together. Some SIMD units can be configured dynamically to operate in different modes, e.g., a 64-bit adder can be configured to operate as two 32-bit
SIMD adders, or as four 16-bit SIMD adders. This resource model has an advantage over the traditional one, where a 64-bit addition can share an adder with a 16-bit addition, but not a group of four 16-bit additions in the same clock cycle. To allow a group of operations to be mapped to a SIMD unit, they must be scheduled to start in the same clock cycle.

Soft constraints are often useful when describing the preferred relative schedule within such a group. The operations that match a template (like the one shown in Equation 3.23) can be identified easily by traversing the CDFG [37, 107]. Some schedulers use a complex operation to substitute a group of operations that can be mapped to a composite functional unit before scheduling [107]; however, this actually imposes a hard constraint on the scheduler, and opportunities to explore alternative schedules for other purposes will no longer be available. To overcome the limitation, soft constraints can be introduced between pairs of operations to guide the optimization engine toward a solution with the desired schedule, while still keeping possibilities to optimize other metrics by not obeying the constraint. The penalty function associated with this type of soft constraint is binary, and the coefficient is estimated according to the gain in performance/area/power when the desired schedule is achieved.

3.4.4 Incremental Synthesis

In incremental synthesis or ECO synthesis, most parts of the design are probably already tuned, and only a small set of operations need to be rescheduled. Thus it is desirable that the schedules for most operations do not change after incremental synthesis. On the other hand, when some operations are rescheduled, it is impossible and unreasonable to fix the schedule of all other operations with hard constraints due to the existing constraints and optimization goals. Using soft constraints, it is easy to express the intention to fix the absolute schedule of an operation or the relative schedule between two operations.
3.4.4.1 Fix the Absolute Schedule

Suppose that for an operation $v$ scheduled at $t$ before incremental synthesis, we hope that $v$ is scheduled at $t$ after the incremental synthesis, and yet we do not want to enforce a hard constraint to force the schedule. We can impose the following soft constraints.

\[ s_v \leq t \]  
\[ -s_v \leq -t \]

According to Lemma 3.3, adding the above constraints preserves total unimodularity. We then add a convex penalty term (e.g., linear, quadratic, exponential) on each associated violation variable to the objective function. The resulting problem is solvable.

3.4.4.2 Fix the Relative Schedule

Sometimes the absolute schedule of an operation is not important, but instead the relative schedule between two operations is preferably fixed. Suppose we have operations $v_i$ and $v_j$ with a preferable distance of $d$, we can add the following integer-difference soft constraints.

\[ s_i - s_j \leq -d \]  
\[ s_j - s_i \leq d \]

Similar to the case of fixing the absolute schedule, a convex penalty term of each violation variable can be added to the objective function, and the problem can be solved efficiently and optimally. With proper weights on the penalty terms, the resulting solution will probably preserve the distance between $v_i$ and $v_j$ when appropriate.

3.4.5 Resource Sharing

Two operations of the same type can often share a functional unit if they are scheduled in different control steps. If it is desirable that a group of operations $\{v_1, v_2, \ldots, v_m\}$
share a functional unit, we can derive a linear order of the operations considering a reference schedule or the ASAP/ALAP steps of the operations like done in [36]. If such an order is satisfied in the final schedule, the group of operations can share. Without loss of generality, we can assume that the operations are already ordered. Then soft constraints can be added as follows.

\[ s_i - s_{i+1} \leq -1, i = 1, \ldots, m - 1 \]  \hspace{1cm} (3.28)

Another way is to add soft constraints for ordering between every pair of nodes in the group.

\[ s_i - s_j \leq -1, \forall 1 \leq i < j \leq m \]  \hspace{1cm} (3.29)

Note that soft constraints for the purpose of resource sharing depend on a predetermined order, and a suboptimal order can influence the solution negatively. Thus, it is best used within an iterative optimization procedure, where the ordering of operations can be adjusted considering the solution in the previous iteration and other optimization goals.

### 3.5 Experimental Result

Here we evaluate the effectiveness of our approach with the application of register reduction. Results on another application for power reduction will be reported in Chapter 4.

A soft constraint generator is implemented for register reduction as described in Section 3.4.1. A soft constraint is generated for every variable, and a linear penalty function is used. The coefficient in the penalty function is decided considering the bitwidth of the variable. It is expected that not all of these soft constraints can be satisfied unless a combinational circuit can be generated; yet we still generate all the soft constraints, and rely on the optimization engine to decide which subset of soft constraints to satisfy in the final implementation.
To evaluate the effectiveness of our approach, we compare the results with and without this soft constraint generator. Several designs in arithmetic and multimedia applications are used in our experiments. Characteristics of these designs are given in Table 3.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>#node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>88</td>
<td>address space translation unit</td>
</tr>
<tr>
<td>test2</td>
<td>333</td>
<td>Gaussian noise generator</td>
</tr>
<tr>
<td>test3</td>
<td>351</td>
<td>floating-point multiplier</td>
</tr>
<tr>
<td>test4</td>
<td>1306</td>
<td>motion compensation (MPEG4 decoder)</td>
</tr>
<tr>
<td>test5</td>
<td>621</td>
<td>motion estimation (MPEG4 encoder)</td>
</tr>
</tbody>
</table>

The designs are synthesized toward the Xilinx Virtex-6 platform using the ISE design suite, and the number of logic slices, look-up tables, flip-flops are reported after placement and routing in Table 3.2. Some of these designs also utilize other resources in the FPGA, such as clock buffers, embedded memory blocks, I/O buffers, etc. Since the use of these resources are not influenced by the scheduling strategy, it is not shown in the table.

From the results, we see an obvious advantage of our approach. An average of 27% reduction is observed in the number of flip-flops. The FPGA platform is known to be abundant in flip-flops: in a logic slice in the Virtex-6 FPGA, every look-up table is followed by a flip-flop. If the result of logic mapped to the look-up table is registered, the flip-flop can be used without using resources in other logic slices; otherwise, if the result is not registered, the flip-flop is left unused. Thus, a reduction in the number of flip-flops does not necessarily lead to a reduction in the number of logic slices in general. Despite this, a 17% reduction in logic slices is achieved in our experiments. This justifies the use of our soft constraint generator on the FPGA platform. A marginal increase in the use of look-up tables (about 1%) is observed in our experiments. We consider this to be noise due to perturbations in the synthesized RTL.
Table 3.2: Experimental results for register reduction.

<table>
<thead>
<tr>
<th>design</th>
<th>cycle</th>
<th>without register reduction</th>
<th>with register reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#slice</td>
<td>$N_s$</td>
</tr>
<tr>
<td>test1</td>
<td>4.0</td>
<td>392</td>
<td>1.33</td>
</tr>
<tr>
<td>test2</td>
<td>13.3</td>
<td>2377</td>
<td>1.16</td>
</tr>
<tr>
<td>test3</td>
<td>5.5</td>
<td>3975</td>
<td>1.17</td>
</tr>
<tr>
<td>test4</td>
<td>7.0</td>
<td>722</td>
<td>1.12</td>
</tr>
<tr>
<td>test5</td>
<td>5.0</td>
<td>946</td>
<td>1.28</td>
</tr>
<tr>
<td>geomean</td>
<td></td>
<td>1.21</td>
<td>0.99</td>
</tr>
</tbody>
</table>
CHAPTER 4

Behavior-Level Observability Analysis and Power Optimization

4.1 Introduction

In recent years, power dissipation has become an increasingly critical issue in IC design. A number of techniques for power reduction have been developed in advanced RTL synthesis tools. While some techniques try to replace power-hungry devices with their power-efficient counterparts at possible costs in performance and/or area, other orthogonal approaches reduce power by avoiding unnecessary operations, using techniques such as operand isolation, clock gating and power gating. Observability don’t-care (ODC) conditions, introduced by the logic synthesis community (for example, [43, 70, 115]), play an important role in the identification of unnecessary operations in a Boolean network. Isolation cells can be inserted at inputs of a functional unit when its result is not observable [119]. For clock gating, the simplest approach is based on stability conditions [53]: when the value stored in a register does not change, its clock can be gated. It is recognized that exploiting ODC conditions in clock gating can lead to significantly more power reduction by avoiding unobservable value changes in registers [6, 9, 10, 53, 122]. Figure 4.1 shows a simple case where ODC conditions in an RTL design are used for clock gating: when $A$ is greater than 5, the output of the multiplexer is equal to $A$, and thus $B^2$ computed by the multiplier is unobservable; then we can get the implementation shown on the right where clock is gated for some registers and activity of the multiplier can be avoided.
Figure 4.1: An example of clock gating based on ODC conditions in RTL synthesis.

The problem of computing ODC conditions in a sequential RTL model has been approached in a number of ways. Some prior work views the circuit as a finite-state machine (FSM) and calculates the exact ODC condition for every bit using formal methods [9, 10]. However, the number of states in an FSM can be exponential to the number of registers. Thus, the exact approach can be prohibitively expensive for moderately large designs. Methods developed in practical systems are often conservative but more scalable, without a thorough analysis of the FSM. The work in [119] assumes that every value stored in a register is observable and only performs analysis on combinational parts of the circuit. Similar assumptions are used in many commercial products. The approach in [122] relies on special patterns in the hardware-description language (HDL) code to compute ODC conditions, and thus the quality of result depends on the coding style. The algorithm in [6] detects ODC conditions based on datapath topology, using backward traversal and levelization techniques. A more recent work [53] shows that more ODC conditions can be uncovered in the results of [6] by propagating ODC conditions that are already utilized in other parts of the design (possibly discovered manually by the designer). [162] proposes to perform observability analysis at the behavior level, in order to discover opportunities for power optimization in an existing RTL design. The approach relies on the branching structure of the program, but ignores correlation between Boolean values. All these methods are shown to be very effective in practice. However, it is not clear how much opportunity for power optimization still exists due to obvious pessimism when computing ODC conditions.
Even with a powerful tool that could calculate the exact ODC condition for every signal in an RTL model efficiently, the opportunity for power saving is still limited by the existing RTL design. In the example shown in Figure 4.2, the comparison is performed later than the multiplication, and the clock gating technique in Figure 4.1 cannot be applied. This suggests that huge opportunities remain unexploited at a higher level where there is freedom in choosing a good RTL structure among many alternatives of the same functionality.

A more sophisticated example is shown in Figure 4.3, where different schedules in Figure 4.1 and Figure 4.5 with the same latency imply different opportunities for avoiding operations. Note that there is a \texttt{select} instruction in the behavior code (corresponding to a multiplexer in the circuit), and thus the evaluation of some values are not necessary depending on which value is selected as the output. In the first schedule (Figure 4.1), two multiplications (v1 and v2) are always executed. In the second one (Figure 4.5), when v7 is evaluated as false in the first step, v9 will be equal to v3, and values including v1, v2, v5 and v6 are not needed because they will not influence the output. By scheduling instructions intelligently and imposing guarding conditions (\textit{predicates}) to operations based on ODC conditions in the resulting RTL design (this is referred to as \textit{operation gating}), we can effectively restructure the control flow and get different equivalent C++ codes; the resulting implementation can have different power under the same performance constraint. A powerful high-level synthesis tool could explore such higher-level opportunities to generate and redistribute ODC conditions.
whereas an RTL synthesis tool is unable to explore this design space—it can at most
take advantage of available ODC conditions for a fixed schedule.

```cpp
unsigned module(unsigned a, unsigned b,
                unsigned c, unsigned d) {
  unsigned v1 = a * a;
  unsigned v2 = b * b;
  unsigned v3 = c * d;
  unsigned v4 = a + b;
  unsigned v5 = v1 + v2;
  bool v6 = (v5 == 100);
  bool v7 = (a == c);
  bool v8 = v6 & v7;
  unsigned v9 = v8 ? v3 : v4;
  return v9;
}
```

Figure 4.3: A dataflow graph described as a C++ function.

A common problem associated with optimization at a higher level is that an ac-
curate estimation model is often absent. Fortunately, the problem does not exist for
ODC-based power optimization. In fact, all data flows and executing conditions can
be analyzed statically in a behavior description and optimized during scheduling. In
such a case, observability can be analyzed value by value (instead of register by reg-
ister when different values share one register in a time-multiplexing manner). The
high-level synthesis tool can explicitly specify the optimized ODC condition for every
register in the RTL description it generates, so that an RTL synthesis tool using simple
and conservative ODC analysis algorithms can work without losing opportunities for
power optimization.

It then becomes an interesting problem how a power-friendly RTL model can be
unsigned
module(unsigned a, unsigned b,
unsigned c, unsigned d) {
    if (a * a + b * b == 100) {
        unsigned v3 = c * d;
        if (a == c)
            return v3;
    }
    return a + b;
}

Figure 4.4: A schedule for the dataflow graph in Figure 4.3 and the implied control structure after operation gating.

generated in order to maximize the effectiveness of ODC-based power-management techniques. In this dissertation, we study this problem systematically. Our contributions include the following.

- We present a formal framework for observability analysis at the behavior level. We introduce several observability measures and explain their relations.

- We describe an efficient method to compute observability at the behavior level. We first present an abstraction of a dataflow graph using only black boxes and generalized select operations. Then, a method is developed to compute the smoothed behavior-level observability based on several theorems. The method is exact for dataflow graphs with black box abstraction. We also allow certain forms of knowledge about inputs and other instructions to be considered.

- We present a high-level synthesis flow for power optimization using operation gating, guided by behavior-level observability, and demonstrate the effectiveness of our approach in real-word designs. To the best of our knowledge, this is the
unsigned module(unsigned a, unsigned b, 
    unsigned c, unsigned d) {
    if (a == c) {
        unsigned v3 = c * d;
        if (a * a + b * b == 100)
            return v3;
    }
    return a + b;
}

Figure 4.5: An alternative schedule of the dataflow graph in Figure 4.3 and the implied 
control structure after operation gating.

first time that high-level synthesis is guided by a comprehensive observability 
analysis for power optimization.

4.2 Behavior-Level Observability

4.2.1 Observability for a General Function

The observability of a function \( f(x, y) \) with respect to part of its variables \( x \) is a 
Boolean-valued function of the rest of the variables \( y \); the observability is true for 
values of \( y \) which makes it possible that changes in \( x \) are observable at the output.

**Definition 4.1** (observability). *For a function \( z = f(x, y) : \mathbb{X} \times \mathbb{Y} \rightarrow \mathbb{Z} \), where \( x \in \mathbb{X}, \) 
\( y \in \mathbb{Y} \), an observability function of \( f \) with respect to \( x \) is a function \( \mathcal{O}_x f : \mathbb{Y} \rightarrow \{0, 1\}, \) 
so that \((\exists x_1, x_2 \in \mathbb{X}, f(x_1, y) \neq f(x_2, y)) \Rightarrow \mathcal{O}_x f(y). \)

Informally, \( \mathcal{O}_x f \) is a necessary condition about \( y \) for \( x \) to be observable. It is clear 
that Definition 4.1 is compatible with the definition of the observability condition for
Boolean functions.

When only part of the variables can be used for observability computation, we usually need to make conservative decisions about other unknown variables by using a necessary condition of the exact observability. This can be done using projection.

**Definition 4.2** (projection). For a Boolean-valued function $h(y_1, y_2): \mathbb{Y}_1 \times \mathbb{Y}_2 \rightarrow \{0, 1\}$, the projection of $h$ onto $y_1$ is a function $\mathcal{P}_{y_1} h: \mathbb{Y}_1 \rightarrow \{0, 1\}$, so that

$$\mathcal{P}_{y_1} h(y_1) = \begin{cases} 1 & \text{if } \exists y_2 \in \mathbb{Y}_2, \ h(y_1, y_2) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Informally, $\mathcal{P}_{y_1} h(y_1)$ is weaker than $h(y_1, y_2)$, but it is the strongest necessary condition for $h(y_1, y_2)$ with respect to $y_1$.

**Lemma 4.1.** For $g: \mathbb{Y}_1 \times \mathbb{Y}_2 \times \mathbb{Y}_3 \rightarrow \{0, 1\}$,

$$\mathcal{P}_{y_1}(\mathcal{P}_{\{y_1,y_2\}} g) \iff \mathcal{P}_{y_1} g.$$  

For a dataflow graph $g$, let $x \in X$ be the value whose observability is being considered. We cut the edges from the operation that computes $x$, and treat $x$ as a primary input. Let $V \in \mathbb{V}$ be the set of all the other primary inputs in $g$, among which $B \in \mathbb{B}$ is the set of Boolean values, and $C \in \mathbb{C}$ is the set of other values. Let $OUT \in \mathbb{OUT}$ be the output. We view the program as a function $g: X \times B \times C \rightarrow OUT$.

**Definition 4.3** (BL-observability). For the program $g$ as described above, the behavior-level observability condition of $x$ is $BLO(x) \equiv \mathcal{O}_x g$.

According to its definition, $BLO(x)$ is a function: $\mathbb{B} \times \mathbb{C} \rightarrow \{0, 1\}$. Since only Boolean values are allowed in a predicate expression in our target architecture, we define the Boolean behavior-level observability condition of $x$ as the projection $BLO(x)$ onto $B$, so that we get an observability condition that uses only Boolean values.

---

1For two logic conditions $A$ and $B$, if $A \Rightarrow B$, we say that $B$ is weaker than $A$, and that $A$ is stronger than $B$.  

58
Definition 4.4 (B-BL-observability). For the program \( g \) as described above, the Boolean behavior-level observability condition of \( x \) is \( \text{BBLO}(x) \equiv \mathcal{P}_B \text{BLO}(x) \).

4.2.2 Dataflow Graph Abstraction

Different from a Boolean network where each vertex is a simple Boolean operator and each edge is a single-bit signal, a dataflow graph represents program behavior at the word level. Operations in a dataflow graph can be either Boolean operators like those in a Boolean network (such as \text{and}, \text{or}, \text{not}), or more complex ones at the word level (such as \text{add}, \text{mul}, \text{div}). A special operation is \text{select}, whose output is equal to one of its data inputs, depending on the control input. While it is theoretically possible to decompose all complex word-level operations into Boolean networks and compute \( \text{BLO}(x) \) using techniques for observability computation in Boolean networks, the approach is often computationally intractable due to the large size of the network. Moreover, even if we have an efficient way to compute observability in the large Boolean network, the observability condition is likely to be very complex, involving bits from word-level signals. This is not quite useful for operation gating, because only Boolean values can be used in predicates in a typical synthesis tool.

To get reasonable observability conditions efficiently at the word level without elaborating all details, we propose to model complex operations (i.e., all operations that take a non-Boolean value as an input, excluding \text{select}) in the dataflow graph as black boxes. A black box has fixed input/output bitwidths, and it implements some non-Boolean function whose semantics is being ignored in our analysis. In other words, a black box can be instantiated as any function with the specified input/output bitwidths. Under this abstraction, no knowledge about the complex operations can be used, and the goal is to obtain observability conditions that hold regardless of the instantiations of black boxes. Consider the general case where a dataflow graph has \( m \) black boxes \( B_1, B_2, \ldots, B_m \). The observability condition of a value \( x \) depends on the instantiation of each black box. Let \( \text{BLO}_{f_1, \ldots, f_m}(x) \) be the observability condition of \( x \) when \( B_i \) is
instantiated as a function \( f_i \). We are interested in the strongest necessary condition for \( BLO(x) \), without knowing anything about \( f_i, i = 1, \ldots, m \). The result is defined as the **smoothing** of \( BLO(x) \) with respect to all the black boxes. Denoting the result of smoothing as \( S_{\{f_1, \ldots, f_m\}} BLO(x) \), or simply \( SBLO(x) \), we have

\[
SBLO(x) = \sum_{f_1, \ldots, f_m} BLO_{f_1, \ldots, f_m}(x). \tag{4.1}
\]

Please recall that the smoothing of a Boolean function \( g(x_1, \ldots, x_i, \ldots, x_n) \) over \( x_i \) is defined as \( S_{x_i} g \equiv g|_{x_i=1} + g|_{x_i=0} \) in [115]. Note that our definition of smoothing over black-box functions in Equation 4.1 is compatible with the original definition in [115]. In fact, they are the same if we view variable \( x_i \) as the output of a 0-input-1-output black-box function.

Clearly, \( SBLO(x) \) is weaker than the exact \( BLO(x) \) due to the absence of knowledge about operations that are modeled as black boxes. However, the result of such an operation typically depends on all of its operands (with rare exceptions, such as the case when one of the inputs of a \texttt{mul} operation is 0), and correlations between different non-Boolean values are difficult to analyze and represent in a thorough way at the word level. Thus, we consider the black-box modeling a reasonable abstraction for behavior-level observability analysis.

We also generalize the \texttt{select} operation to facilitate observability analysis. A \((k, l)\)-\texttt{select} operation takes \( k \) control inputs \((b_1, \ldots, b_k)\), \( l \) data inputs \((d_1, \ldots, d_l)\) and generates one output \( z \). All control inputs are Boolean variables, and all data inputs are of the same bitwidth.

\[
z = \begin{cases} 
    d_1 & \text{when } s_1(b_1, \ldots, b_k) = 1 \\
    d_2 & \text{when } s_2(b_1, \ldots, b_k) = 1 \\
    \vdots \\
    d_l & \text{when } s_l(b_1, \ldots, b_k) = 1 
\end{cases} \tag{4.2}
\]
Or,

\[ z = \sum_{i=1}^{l} s_i(b_1, \ldots, b_k) d_i \]  \hspace{1cm} (4.3)

Here \( s_1, \ldots, s_l \) is a set of orthonormal Boolean bases; that is,

\[ s_i s_j = 0, \forall i \neq j \in \{1, \ldots, l\} \]  \hspace{1cm} (4.4)

\[ \sum_{i=1}^{l} s_i = 1. \]  \hspace{1cm} (4.5)

It is easy to note that the traditional \select operation is a (1, 2)-\select with \( s_1(b_1) = b_1 \) and \( s_2(b_1) = \overline{b_1} \). This generalized \select allows selecting from multiple data inputs, and absorbs Boolean functions.

With the approach of black-box modeling and the above generalization of \select, there are only two types of operations remaining in the dataflow graph—black box and generalized \select. We make the following restrictions to facilitate discussion in the following parts of this chapter. Note that a valid dataflow graph can always be normalized to a form that conforms to these restrictions.

1. If a Boolean variable is used by a \select operation as a control input, it cannot be used by black boxes or by any \select operation as a data input. In other words, values are divided into two categories: either used purely as control signals (by \select operations), or purely as data inputs (by black boxes and \select operations).

   - If a Boolean variable \( b \) is a primary output, introduce a (1, 2)-\select that selects constant 1 if \( b \) is true and constant 0 if \( b \) is false.
   - If a \select operation takes Boolean variables as data inputs, we can always replace the \select operation with a Boolean expression (which is eventually absorbed in another \select).

2. For each \select operation, all of its inputs are distinct.
• If a Boolean variable appears more than once in the list of control inputs of a select operation, the number of control inputs can be reduced, and the selecting logic can be simplified.

• If the same data value is selected in more than one case, the number of data inputs can be reduced, and the cases where the same value is selected can be merged.

3. Each data input of a select comes from a black box.

• Primary inputs and constants are regarded as outputs of black boxes.

• When the result of a select \( n_1 \) is used by another select \( n_2 \), we can simply replace \( n_2 \) with a combination of \( n_1 \) and \( n_2 \), so that the result of \( n_1 \) is not used by another select. Note that \( n_1 \) may still be present after the transformation if it has other uses.

For the example dataflow graph shown in Figure 4.3, operations that evaluate \( v_1, v_2, v_3, v_4, v_5, v_6 \) and \( v_7 \) are all modeled as black boxes; the operation that evaluates \( v_8 \) is absorbed in the \( (2, 2) \)-select operation \( v_9 \); the selection function for the \( (2, 2) \)-select is \( s_{v_3}(v_6, v_7) = v_6v_7 \), \( s_{v_4}(v_6, v_7) = \overline{v_6}v_7 \). The dataflow graph is transformed as shown in Figure 4.6.

4.3 Behavior-Level Observability Analysis

As mentioned previously, computing the exact observability condition requires nontrivial effort, essentially breaking all values into individual bits and applying techniques for Boolean networks. In this section, we describe an algorithm to compute \( SBLO \), i.e., observability under the abstraction using black boxes described in Section 4.2. The algorithm propagates and manipulates \( SBLO \) directly, and thus it avoids the trouble of considering the instantiations of black boxes.
Figure 4.6: The black-box abstraction of the dataflow graph in Figure 4.3, where v9 is a generalized select operation, and black boxes are filled with shade.

### 4.3.1 Review of Observability Computation in a Boolean Network

Our method for computing $S_{BLO}$ is based on a technique for observability analysis in Boolean networks [115]. Here we give a brief review of the algorithm. For simplicity, we consider the case when the Boolean network has only one primary output. The observability of a node in the Boolean network with regard to multiple primary outputs can be computed by summing up its observability conditions with regard to each individual primary outputs. The algorithm labels the observability conditions on nodes and edges in reverse topological order. It proceeds with three kinds of actions:

**Initialize.** For the primary output $z$ under consideration, set $BLO(z) = 1$. For all other primary output $w$, set $BLO(w) = 0$.

**Propagate.** Compute the observability at each input edge of a node from node observability. For each node $z = f(x_1, \ldots, x_n)$, we have $BLO(x_i) = BLO(z) \frac{\partial f}{\partial x_i}$, $i = 1, \ldots, n$.

**Merge.** Compute the observability of a node from the edge observabilities of its fanout edges. If a value $y$ is used $m$ times as $y_1, \ldots, y_m$, when $y$ is visited, we already
have the edge observability conditions $BLO(y_i), i = 1, \ldots, m$. These edge observability conditions are computed independently from downstream operations; thus the correlation that $y_1 = y_2 = \cdots = y_m$ needs to be considered when merging edge observability conditions. Equation 4.6 is used to derive node observability.

$$BLO(y) = \bigoplus_{i=1}^{m} BLO(y_i)|_{y_{i+1} = \cdots = y_m = y'}$$ (4.6)

4.3.2 Observability Analysis with Black Boxes

In this subsection, we show that for a dataflow graph composed of black boxes and generalized `select` operations, if the requirements in Section 4.2.2 are satisfied, we can compute and propagate the smoothed observability $SBLO$ directly, using an approach similar to that for Boolean networks. For simplicity, we first consider the case with only one output (an output is an operation whose result may be used outside the current dataflow graph, or by the next loop iteration); the observability with regard to multiple outputs can be obtained by considering these outputs one by one and summing up the smoothed observability conditions with regard to different outputs for the same value.

Similar to the approach for observability analysis in a Boolean network, our algorithm also uses the three types of actions—initialization, propagation and merging, among which the initialization is trivial. In the following, we develop theorems that give rules for propagation through black boxes and generalized `select` operations, as well as rules for merging of control signals and data signals.

**Theorem 4.1** (propagate through a black box). For a black box $z = f(x_1, \ldots, x_m)$, the edge observability satisfies $SBLO(x_i) = SBLO(z), \forall i$.

**Proof.** Without loss of generality, we only consider $x_1$. According to the propagation
rule in a Boolean network, we have

\[ BLO(x_1) = BLO(z) \frac{\partial z}{\partial x_1} \quad (4.7) \]

\[ = BLO(z)(f(0, x_2, \ldots, x_m) \oplus f(1, x_2, \ldots, x_m)) \quad (4.8) \]

Let \( S_f \) be the smoothing operator over all possible instantiations of black box \( f \), and let \( S \) be the smoothing operator over all black boxes in the network. We have

\[ S_f BLO(x_1) = S_f(BLO(z)(f(0, x_2, \ldots, x_m) \oplus f(1, x_2, \ldots, x_m))) \quad (4.9) \]

\[ = BLO(z) S_f(f(0, x_2, \ldots, x_m) \oplus f(1, x_2, \ldots, x_m)) \quad (4.10) \]

\[ = BLO(z) \quad (4.11) \]

Note that from Equation 4.10 to Equation 4.11, we use the fact that there exists an instantiation of \( f \) so that \( f(0, x_2, \ldots, x_m) \oplus f(1, x_2, \ldots, x_m) = 1 \); one such instantiation is \( f(x_1, x_2, \ldots, x_m) = x_1 \). Thus,

\[ SBLO(x_1) = S(S_f BLO(x_1)) = SBLO(z) \quad (4.12) \]

\[ \square \]

We introduce the following lemma, which is useful to prove Theorem 4.2.

**Lemma 4.2.** Let \( \odot \) be an arbitrary binary Boolean operator. For a set of orthonormal Boolean basis \( s_i, i = 1, \ldots, l \),

\[ \sum_{j=1}^{l} (f_i \odot g_i) s_i = \left( \sum_{i=1}^{l} f_i s_i \right) \odot \left( \sum_{i=1}^{l} g_i s_i \right). \quad (4.13) \]

**Proof.** According to the definition of orthonormal Boolean basis, exactly one of \( \{s_1, \ldots, s_l\} \) is true in any scenario. When \( s_i \) is true, both sides of Equation 4.13 equal \( f_i \odot g_i \). \( \square \)

**Theorem 4.2** (propagate through select). For a \((k, l)\)-select instruction \( z = g(b_1, \ldots, b_k, d_1, \ldots, d_l) \), with \( s_i(b_1, \ldots, b_k) \) being the condition under which \( d_i \) is selected. We have \( SBLO(d_i) = SBLO(z) s_i(b_1, \ldots, b_k) \), \( SBLO(b_i) = SBLO(z) \sum_{j=1}^{l} \frac{\partial s_i}{\partial b_j} \).
Proof. According to the rule for propagation in a Boolean network without black boxes, and the definition of observability,

\[
BLO(d_j) = BLO(z) \left( \sum_{i=1}^{l} s_i d_i \bigg|_{d_j=0} \bigoplus \sum_{i=1}^{l} s_i d_i \bigg|_{d_j=1} \right) \tag{4.14}
\]

\[= BLO(z) \left( s_j (0 \oplus 1) + \sum_{i=1,i \neq j}^{l} s_i (d_i \oplus d_i) \right) \tag{4.15}
\]

\[= BLO(z) s_j \tag{4.16}
\]

Here from Equation 4.14 to Equation 4.15, Lemma 4.2 is needed. Then it is clear that

\[
SBLO(d_j) = S(BLO(z) s_j) = SBLO(z) s_j. \tag{4.17}
\]

For a control input \(b_j\),

\[
BLO(b_j) = BLO(z) \left( \sum_{i=1}^{l} s_i d_i \bigg|_{b_j=0} \bigoplus \sum_{i=1}^{l} s_i d_i \bigg|_{b_j=1} \right) \tag{4.18}
\]

\[= BLO(z) \left( \sum_{i=1}^{l} s_i|_{b_j=0} d_i \bigoplus \sum_{i=1}^{l} s_i|_{b_j=1} d_i \right) \tag{4.19}
\]

\[
S_{\{d_1, \ldots, d_l\}} BLO(b_j) = BLO(z) S_{\{d_1, \ldots, d_l\}} \left( \sum_{i=1}^{l} s_i|_{b_j=0} d_i \bigoplus \sum_{i=1}^{l} s_i|_{b_j=1} d_i \right) \tag{4.20}
\]

\[= BLO(z) S \left( \sum_{i=1}^{l} s_i|_{b_j=0} d_i \bigoplus \sum_{i=1}^{l} s_i|_{b_j=1} d_i \right) \tag{4.21}
\]

In the following we show that

\[
\sum_{i=1}^{l} (s_{i|b_j=0} \oplus s_{i|b_j=1}) = S \left( \sum_{i=1}^{l} s_{i|b_j=0} d_i \bigoplus \sum_{i=1}^{l} s_{i|b_j=1} d_i \right). \tag{4.22}
\]

1. If \(\sum_{i=1}^{l} (s_{i|b_j=0} \oplus s_{i|b_j=1}) = 1\), \(\exists s_i, s_{i|b_j=0} \oplus s_{i|b_j=1} = 1\). In the smoothing, consider the case when \(d_i = 1\), and \(d_j = 0, j \neq i\), we have \(\sum_{i=1}^{l} s_{i|b_j=0} d_i \oplus \sum_{i=1}^{l} s_{i|b_j=1} d_i = s_{i|b_j=0} \oplus s_{i|b_j=1} = 1\), and thus \(S \left( \sum_{i=1}^{l} s_{i|b_j=0} d_i \bigoplus \sum_{i=1}^{l} s_{i|b_j=1} d_i \right) = 1\).
2. If \( \sum_{i=1}^{l} s_i|b_j=0 \oplus s_i|b_j=1 = 0 \), \( \forall s_i, s_i|b_j=0 = s_i|b_j=1 = s_i \). Then \( \sum_{i=1}^{l} s_i|b_j=0 d_i = \sum_{i=1}^{l} s_i|b_j=1 d_i, S \left( \sum_{i=1}^{l} s_i|b_j=0 d_i \oplus \sum_{i=1}^{l} s_i|b_j=1 d_i \right) = 0 \).

Thus,
\[
SBLO(b_j) = S \left( BLO(z) \sum_{i=1}^{l} (s_i|b_j=0 \oplus s_i|b_j=1) \right) \tag{4.23}
\]
\[
= S \left( BLO(z) \sum_{i=1}^{l} \frac{\partial s_i}{\partial b_j} \right) \tag{4.24}
\]
\[
= SBLO(z) \sum_{i=1}^{l} \frac{\partial s_i}{\partial b_j} \tag{4.25}
\]

To derive the rule for merging \( SBLO \) of multiple edges, we introduce the following lemmas.

Denote a single-input-single-output black box as \( \lambda \). That is, \( \lambda(x) \) could be instantiated in the following four ways:

1. \( \lambda(x) = 1 \)
2. \( \lambda(x) = 0 \)
3. \( \lambda(x) = x \)
4. \( \lambda(x) = \bar{x} \)

**Lemma 4.3** (equivalence under addition of \( \lambda \)). *For a network with black boxes, consider a black box \( B : z = f(x_1, \ldots, x_m) \), perturb the network by adding a \( \lambda \) after an input, or after the output; the resulting network is equivalent to the original network in the sense that the \( SBLO \) condition for any signal is unchanged.*

**Proof.** The lemma is trivial noting that after the addition of \( \lambda \), the set of functions the combination of the two black boxes can be instantiated as is the same as that of the original black box. \( \square \)
Lemma 4.4.  
\[ S_{x_1, \ldots, x_k} \left( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) \right) = \sum_{i=1}^{k+1} f_i(z). \]

Proof. We show the following two cases based on the value of the right-hand side.

1. If \( \sum_{i=1}^{k+1} f_i(z) = 0 \), we have \( f_i(z) = 0 \), \( \forall i = 1, \ldots, k + 1 \). Then \( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) = (\oplus_{i=1}^{k} 0) \oplus 0 = 0 \), thus \( S_{x_1, \ldots, x_k} \left( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) \right) = 0 \).

2. If \( \sum_{i=1}^{k+1} f_i(z) = 1 \), then \( \exists j, f_j(z) = 1 \). For the left-hand side, we can get an assignment of \( x_i, i = 1, \ldots, k \), so that \( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) = 1 \). There are two scenarios.

(a) If \( f_{k+1}(z) = 1 \), we can assign \( x_i = 0, i = 1, \ldots, k \), then \( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) = f_{k+1}(z) = 1 \).

(b) If \( f_{k+1}(z) = 0 \), then \( \exists j \neq k + 1, f_j(z) = 1 \). We can assign \( x_j = 1, \) and \( x_i = 0, \forall i \neq j \). Thus \( \left( \oplus_{i=1}^{k} x_i f_i(z) \right) \oplus f_{k+1}(z) = f_j(z) = 1 \).

\[ \square \]

Theorem 4.3 (merge edge observability for data). For a value \( x \) that is used by black boxes or by select instructions as data inputs, suppose \( x_1, \ldots, x_p \) are the edges beginning from \( x \). We have

\[ SBLO(x) = \sum_{i=1}^{p} SBLO(x_i). \]

Proof. There are two cases.

1. If \( x \) is used purely by black boxes, \( B_1, \ldots, B_p \). According to Lemma 4.3, we can always add \( \lambda \), so that each \( B_i : z_i = \lambda_i(x_i) \). We have \( BLO(x_i) = BLO(z_i) \frac{\partial \lambda_i}{\partial x_i} \).

\[ BLO(x) = \bigoplus_{i=1}^{p} BLO(x_i)|_{x_{i+1} = \ldots = x_p = x'} \quad (4.26) \]

\[ = \bigoplus_{i=1}^{p} BLO(z_i) \frac{\partial \lambda_i}{\partial x_i} |_{x_{i+1} = \ldots = x_p = x'} \quad (4.27) \]
Clearly, \( \frac{\partial \lambda_i}{\partial x_i} \) is a constant—either 0 or 1 in this case, depending on instantiations of \( \lambda_i \).

\[
S_{\{B_1,\ldots,B_p\}} BLO(x) = S_{\{B_1,\ldots,B_p\}} \left( \bigoplus_{i=1}^p BLO(z_i) \frac{\partial \lambda_i}{\partial x_i} \bigg|_{x_{i+1} = \ldots = x_p = x'} \right) \tag{4.28}
\]

\[
= S_{\{B_1,\ldots,B_p\}} \left( \bigoplus_{i=1}^p BLO(z_i) \right) \tag{4.29}
\]

\[
= \sum_{i=1}^p BLO(z_i) \tag{4.30}
\]

Lemma 4.4 is used from Equation 4.29 to Equation 4.30. Thus

\[
SBLO(x) = S \left( S_{\{B_1,\ldots,B_p\}} BLO(x) \right) \tag{4.31}
\]

\[
= S \left( \sum_{i=1}^p BLO(z_i) \right) \tag{4.32}
\]

\[
= \sum_{i=1}^p SBLO(z_i) \tag{4.33}
\]

\[
= \sum_{i=1}^p SBLO(x_i) \tag{4.34}
\]

Theorem 4.1 is used from Equation 4.33 to Equation 4.34.

2. When \( x \) is used by select instructions as data input, and possibly by black boxes. The idea is to transform to the first case. For every select instruction that uses \( x \), since the result of select is either used by black boxes, or is a primary output, so we can add a black box after the output of the select.

We further move the 1-input-1-output black box to every data input of the select instruction, using the following rule.

\[
\lambda(select(b_1, \ldots, b_k, d_1, \ldots, d_l)) = select(b_1, \ldots, b_k, \lambda(d_1), \ldots, \lambda(d_l)) \tag{4.35}
\]

Then, the case is reduced to the first case.
**Theorem 4.4** (merge edge observability for control). For a node $x$ that is used as control inputs in $m$ select instructions, $f_1, \ldots, f_m$, each having $l_i$ data inputs, suppose $x_1, \ldots, x_m$ are the edges beginning from $x$. We have

$$SBLO(x) = \sum_{i=1}^{m} \left( SBLO(f_i)|_{x_{i+1}=\ldots=x_m=x'} \sum_{j=1}^{l_i} \frac{\partial s^j_i}{\partial x_i} \right),$$

where $s^j_i$ is the selection function for data input $j$ in the $i$th select operation.

**Proof.** Since the result of every select operation is only used by black boxes (or is a primary output), we can safely add a 1-input-1-output black box after each select, without changing the smoothed observability. Thus, we only need to consider the case with exactly one black box following the output of every select operation.

Let $B_i$ be the black box following $f_i$ and let $z_i$ be the output of $B_i$. Denote $u_i = \frac{\partial z_i}{\partial f_i}$, then

$$BLO(f_i) = BLO(z_i) \frac{\partial z_i}{\partial f_i} = u_i BLO(z_i).$$

(4.36)

$$BLO(x_i) = BLO(f_i) \frac{\partial f_i}{\partial x_i} = u_i BLO(z_i) \frac{\partial f_i}{\partial x_i}$$

(4.37)

$$BLO(x) = \bigoplus_{i=1}^{m} BLO(x_i)|_{x_{i+1}=\ldots=x_m=x'}$$

(4.38)

$$= \bigoplus_{i=1}^{m} u_i \frac{\partial f_i}{\partial x_i} \left( BLO(z_i)|_{x_{i+1}=\ldots=x_m=x'} \right)$$

(4.39)

$$S_{\{B_1,\ldots,B_m\}} BLO(x) = S_{\{u_1,\ldots,u_m\}} \left( \bigoplus_{i=1}^{m} u_i \frac{\partial f_i}{\partial x_i} \left( BLO(z_i)|_{x_{i+1}=\ldots=x_m=x'} \right) \right)$$

(4.40)

$$= \sum_{i=1}^{m} \frac{\partial f_i}{\partial x_i} \left( BLO(z_i)|_{x_{i+1}=\ldots=x_m=x'} \right)$$

(4.41)
\[ SBLO(x) = S \left( S_{\{u_1, \ldots, u_m\}} \mathcal{BLO}(x) \right) \]  
\[ = S \left( \sum_{i=1}^{m} \frac{\partial f_i}{\partial x_i} \left( \mathcal{BLO}(z_i) \big|_{x_{i+1}=\ldots=x_m=x'} \right) \right) \]  
\[ = \sum_{i=1}^{m} S \left( \frac{\partial f_i}{\partial x_i} \left( \mathcal{BLO}(z_i) \big|_{x_{i+1}=\ldots=x_m=x'} \right) \right) \]  

Let \( D^j_i \) be the black box before the \( j \)th data input of the select instruction \( f_i \). Clearly, since \( B_i \) is downstream to \( D^j_i \), \( \mathcal{BLO}(z_i) \) is independent of \( D^j_i \) in the backward propagation process. Thus, we have

\[ S \left( \frac{\partial f_i}{\partial x_i} \left( \mathcal{BLO}(z_i) \big|_{x_{i+1}=\ldots=x_m=x'} \right) \right) = S \left( \sum_{j=1}^{l_i} \frac{\partial s^j_i}{\partial x_i} \mathcal{BLO}(z_i) \big|_{x_{i+1}=\ldots=x_m=x'} \right) \]  
\[ = \sum_{j=1}^{l_i} \frac{\partial s^j_i}{\partial x_i} \mathcal{BLO}(z_i) \big|_{x_{i+1}=\ldots=x_m=x'} \]  
\[ = \mathcal{BLO}(f_i) \big|_{x_{i+1}=\ldots=x_m=x'} \sum_{j=1}^{l_i} \frac{\partial s^j_i}{\partial x_i} \]  

Here \( l_i \) is the number of data inputs for \( f_i \). From Equation 4.47 to Equation 4.48, the result in Equation 4.22 is used.

Using the result of Equation 4.50 in Equation 4.44, we get

\[ SBLO(x) = \sum_{i=1}^{m} \left( \mathcal{BLO}(f_i) \big|_{x_{i+1}=\ldots=x_m=x'} \sum_{j=1}^{l_i} \frac{\partial s^j_i}{\partial x_i} \right). \]  

Note that the above theorems are proved for the case where data values are individual bits, but the results can be generalized easily to word-level data values. The reason is that with the black-box modeling, the expression of the smoothed observability contains only Boolean values that are used exclusively as control signals (see Theorems...
4.1, 4.2, 4.3, 4.4); thus every bit in the same word will have the same smoothed observability in propagation and merging, and the above theorems hold for word-level data.

Based on the above theorems, we develop a process called observability analysis to compute the smoothed observability conditions for all operations. The input program is thoroughly optimized using classic compiler optimizations including control-flow optimization and if-conversion (using the \texttt{select} instructions); a dataflow graph is formed for each acyclic region, and preprocessed into a graph with black boxes and generalized \texttt{select} operations. The algorithm for observability analysis is shown in Algorithm 1.

\begin{algorithm}
\caption{observability analysis for a dataflow graph with a single output.}
\begin{algorithmic}
\ForAll{operation $I$ in reverse topological order}
\If{$I$ directly generates the primary output}
\State \textbf{Initialize}: $SBLO(I) = \text{predicate}(I)$, $\text{predicate}(I)$ standing for the executing condition of $I$.
\ElsIf{$I$ directly generates the primary output}
\State \textbf{Merge}: calculate $SBLO(I)$ from the smoothed observability conditions of edges starting from $I$, which have been calculated when visiting downstream operations. The equation in Theorem 4.3 is used if the value is used as data; the equation in Theorem 4.4 is used otherwise.
\EndIf
\State \textbf{Propagate}: calculate the smoothed observability for every incoming edge $SBLO(\text{src}_i(I))$ based on the type of $I$. If $I$ is a black box, use the equation in Theorem 4.1; otherwise ($I$ is a generalized \texttt{select}) use the equation in Theorem 4.2.
\EndFor
\end{algorithmic}
\end{algorithm}

Clearly, the algorithm performs $O(|V|)$ merging operations and $O(|E|)$ propagations. Based on Theorems 4.1, 4.2, 4.3, 4.4, we have the following.
Theorem 4.5. Algorithm 1 computes the correct smoothed observability condition for each value in the dataflow graph.

For the example dataflow graph with black boxes and the generalized select operations shown in Figure 4.6, after applying Algorithm 1, we get the smoothed behavior-level observability condition shown in Table 4.1.

Since the expression of $S_{BLO}$ contains only Boolean control signals, $S_{BLO}$ is also a $BBLO$ (and is also a $BLO$), according to definition. However, it is not necessarily the strongest $BBLO$ possible, due to the black-box abstraction. The output of a black box is completely unknown, and correlations between values are completely lost after a black box. While we consider this black-box abstraction very useful to enable analysis at a higher level, certain knowledge about values in the dataflow graph, once uncovered, can be employed to strengthen the condition. To do that, we are mostly interested in correlations between Boolean values, e.g., $(x == 3)$ implies $(x < 10)$. Although capturing exact relations between Boolean values is nontrivial, at least some knowledge can be discovered and exploited. Such techniques have been developed in compilers [5], and can be directly applied in our algorithm.

For the example in Figure 4.3, let us assume that we know input $c$ is always an odd number. By analyzing the observability-propagating instructions, it can be asserted that the two Boolean values, $v6 = (a * a + b * b == 100)$ and $v7 = (a == c)$ cannot be true simultaneously, because the set of integer values of $a$ that satisfies $a * a + b * b == 100$ is $\{0, 6, 8, 10\}$, all elements of which are even, so $v7 = (a == c)$ will be false if $v6$ is true. Thus we have the knowledge $v6 \land v7 = \text{true}$, and use it to simplify the conditions. For example, we have $BLO(v3) = v6v7$; with that knowledge, we have $BLO(v3) = v6v7 \land \neg v6v7 = \text{false}$; i.e., we find that $v3$ is always unobservable when $c$ is odd.
Table 4.1: Smoothed behavior-level observability computed by Algorithm 1.

<table>
<thead>
<tr>
<th>value</th>
<th>SBLO</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_9$</td>
<td>true</td>
<td>initialization for primary output ($v_9$)</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$SBLO(v_9)v_6v_7 = v_6v_7$</td>
<td>propagation from select ($v_9$) to input</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$SBLO(v_9)v_6v_7 = v_6v_7$</td>
<td>propagation from select ($v_9$) to input</td>
</tr>
<tr>
<td>$v_6$</td>
<td>$SBLO(v_9) \left( \frac{\partial (v_6v_7)}{\partial v_6} + \frac{\partial (v_6v_7)}{\partial v_7} \right)$</td>
<td>propagation from select ($v_9$) to condition</td>
</tr>
<tr>
<td>$v_7$</td>
<td>$SBLO(v_9) \left( \frac{\partial (v_6v_7)}{\partial v_6} + \frac{\partial (v_6v_7)}{\partial v_7} \right)$</td>
<td>propagation from select ($v_9$) to condition</td>
</tr>
<tr>
<td>$c_{v_3}$</td>
<td>$SBLO(v_3) = v_6v_7$</td>
<td>propagation from black box ($v_3$) to input</td>
</tr>
<tr>
<td>$d$</td>
<td>$SBLO(v_3) = v_6v_7$</td>
<td>propagation from black box ($v_3$) to input</td>
</tr>
<tr>
<td>$a_{v_4}$</td>
<td>$SBLO(v_4) = \overline{v_6}v_7$</td>
<td>propagation from black box ($v_4$) to input</td>
</tr>
<tr>
<td>$b_{v_4}$</td>
<td>$SBLO(v_4) = \overline{v_6}v_7$</td>
<td>propagation from black box ($v_4$) to input</td>
</tr>
<tr>
<td>$v_5$</td>
<td>$SBLO(v_6) = v_7$</td>
<td>propagation from black box ($v_6$) to input</td>
</tr>
<tr>
<td>$a_{v_7}$</td>
<td>$SBLO(v_7) = v_6$</td>
<td>propagation from black box ($v_7$) to input</td>
</tr>
<tr>
<td>$c_{v_7}$</td>
<td>$SBLO(v_7) = v_6$</td>
<td>propagation from black box ($v_7$) to input</td>
</tr>
<tr>
<td>$v_1$</td>
<td>$SBLO(v_5) = v_7$</td>
<td>propagation from black box ($v_5$) to input</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$SBLO(v_5) = v_7$</td>
<td>propagation from black box ($v_5$) to input</td>
</tr>
<tr>
<td>$a_{v_1}$</td>
<td>$SBLO(v_1) = v_7$</td>
<td>propagation from black box ($v_1$) to input</td>
</tr>
<tr>
<td>$b_{v_2}$</td>
<td>$SBLO(v_2) = v_7$</td>
<td>propagation from black box ($v_2$) to input</td>
</tr>
<tr>
<td>$a$</td>
<td>$SBLO(a_{v_1}) + SBLO(a_{v_4})$</td>
<td>merge edge observability for $a$</td>
</tr>
<tr>
<td></td>
<td>$+ SBLO(a_{v_7}) = true$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$SBLO(b_{v_2}) + SBLO(b_{v_4})$</td>
<td>merge edge observability for $b$</td>
</tr>
<tr>
<td></td>
<td>$= \overline{v_6} + v_7$</td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>$SBLO(c_{v_3}) + SBLO(c_{v_7}) = v_6$</td>
<td>merge edge observability for $c$</td>
</tr>
</tbody>
</table>

Here $c_{v_3}$ denotes the edge observability for the edge from $v_3$ to $c$. If a node has only one outgoing edge, the edge observability and node observability are the same.
4.4 Scheduling for Operation Gating Optimization

In this section we discuss how the behavior-level observability obtained by Algorithm 1 can be used to improve the effectiveness of operation gating in scheduling.

4.4.1 Observability under a Given Schedule

Observability conditions associated with different levels of abstractions can be different. In the scheduling process of transforming a CDFG into an FSMD, behavior-level observability conditions are translated into FSMD-observability conditions (observability under a given schedule, more precisely defined later). In the example in Figure 4.3, a behavior-level observability condition for \( v_1 \) is \( v_6 \lor v_7 \lor v_8 \); i.e., \( v_1 \) is observable only when \( v_6 \), \( v_7 \) and \( v_8 \) are all true. However, the evaluation of \( v_1 \) can never be avoided in the first schedule, because the observability of \( v_1 \) is always unknown when it is evaluated and conservative decisions have to be made to guarantee correctness. The second schedule is better in the sense that we could avoid evaluating \( v_1 \) when \( v_7 \) is known to be false—because \( v_6 \lor v_7 \lor v_8 \) will then be false even when \( v_6 \) and \( v_8 \) are not yet evaluated. Here we say the FSMD-observability condition of \( v_1 \) is \( v_7 \). Clearly, different schedules imply different ODC conditions on their associated FSMDs, and it is not always possible to postpone every instruction until its behavior-level observability condition is known, due partly to performance constraints.

For a given schedule \( s \) of program \( g \), let \( A_s(x) \) be the set of values available when value \( x \) is evaluated (i.e., the set of values generated by operations scheduled to finish before the evaluation of \( x \) starts), we define FSMD-observability as follows.

**Definition 4.5 (FSMD-observability).** An FSMD-observability condition of \( g \) with respect to \( x \) under a given schedule \( s \), \( FSMDO_s(x) \equiv P A_s(x)BLO(x) \).

**Definition 4.6 (B-FSMD-observability).** A Boolean FSMD-observability condition (B-FSMD-observability) of \( g \) with respect to \( x \) under a given schedule \( s \), \( BFSMDO_s(x) \equiv \)
\( \mathcal{P}_{BFSMDO_s}(x) \).

\( BFSMDO_s(x) \) is the condition we can use as the predicate of the operation that computes \( x \). The conceptual difference between \( BLO, BBLO, FSMDO \) and \( BFSMDO \) lies in the set of values that are used to evaluate observability. All values in the program can be used for behavior-level analysis, while only available values are meaningful when the schedule is fixed. Theoretically, both Boolean and non-Boolean values can be used, while in practice most architectures support only a Boolean expression as the predicate of an instruction.

Using Lemma 4.1, we have

**Theorem 4.6.** \( BFSMDO_s(x) \iff \mathcal{P}_{A_s(x)}BBLO(x) \).

Figure 4.7: Relations between observabilities. The red arc shows the way we obtain \( BFSMDO \).

Theorem 4.6 uncovers relations between \( BLO, BBLO, FSMDO \) and \( BFSMDO \); it gives a way to compute \( BFSMDO \) under a given schedule by projecting a \( BBLO \) condition onto available values. In Figure 4.7, the black arcs illustrate the definition of observabilities by projection, and the red arc illustrates the way to compute \( BFSMDO \) from \( BBLO \) by projection, as stated in Theorem 4.6. Using Algorithm 1, we obtain \( SBLO \) as a \( BBLO \), which is subsequently projected as a \( FSMDO \) for operation gating.
The problem considered here can be described as follows: Given a CDFG and profiling information, as well as the cost (average power) for executing each instruction, find a schedule that leads to the smallest expected total cost after operation gating, subject to data-dependency constraints and a latency constraint.

4.4.2 Previous Work

Considering the fact that only Boolean values already evaluated can be used in predicates for operation gating, the impact of scheduling on ODC-based power management is obvious. To our knowledge, the work in [117] presents the first algorithm designed to create more opportunities for ODC-based power management. This method works as a post-processing step on an existing schedule: it examines multiplexers (select instructions) one by one and tries to move the instruction by computing the Boolean operand earlier if possible. Authors of [117] noticed that their results depended on the order in which multiplexers were examined, and used reverse topological order in their implementation. [25] proposes an improved optimization technique using priority and soft dependency edges.

Both [117] and [25] use a very simple method for observability analysis. They do not generalize the select operation; thus the dataflow graph contains Boolean operations such as and/or/not, which generate the control signals for select. However, those Boolean operations are essentially modeled as black boxes just like add/sub. Hence, knowledge about Boolean operations are unexploited, resulting in weaker observability conditions compared to the proposed method in Section 4.3. For example, in the schedule illustrated in Figure 4.5, the evaluation of v3 can be avoided when v7 is false. This may look straightforward in the original code, but it is nontrivial in the if-converted form, where the first operand of the select instruction, v8, is computed later than v3. The method in [117] and [25] will not find such an opportunity. Although the method can possibly be extended by viewing and/or as degenerated select, it still cannot capture the information that either operand can mask the observability of the
other. Such information is essential for control-flow restructuring when the scheduler exploits different possible speculation/predication schemes under a latency constraint.

### 4.4.3 Optimization Using Soft Constraints

When the schedule is optimized for operation gating, along with other objectives such as latency, different algorithm frameworks can be used. As the problem is intrinsically difficult even without the consideration of operation gating, it is often solved using heuristics like list scheduling [1] or force-directed scheduling [129]. The post-processing technique in [117] and the approach of [25] can be viewed as natural adaptations of previous heuristics to the problem with consideration of operation gating.

Here we propose to use integer-difference soft constraints to express the design intention for operation gating, and rely on the scheduling engine described in Chapter 3 to make global tradeoffs among operation gating and other design intentions.

Recall that in our scheduling formulation, every operation \( v \) in the CDFG is associated with an integer-valued scheduling variable \( s_v \). When it is preferred that a Boolean value \( c \) is computed before another value \( v \) so that \( v \) can be avoided when \( c \) takes a certain value, an integer-difference soft constraint can be added as

\[
s_c - s_v \leq -b - d_c + 1, \tag{4.52}
\]

where \( d_c \) is the number of clock cycles operation \( c \) spans, and \( b \) is the number of clock cycles needed to separate operations \( c \) and \( v \). The value of \( b \) depends on the power management technique and the target platform: a typical value of \( b \) is 1 if clock gating or operand isolation is used; it means that the condition should be available at least 1 cycle before it can be used as a predicate for clock gating. For power gating, the number \( b \) is probably larger than 1.

When the soft constraint in Equation 4.52 is violated, the possibility of avoiding \( v \) based on value of \( c \) is eliminated, regardless of the amount of violation. Thus, the binary penalty function defined in Equation 3.10 is used. The coefficient in the penalty
function is the estimate of potential energy saving if the soft constraint is satisfied based on profiling information.

4.5 Experimental Result

The proposed techniques have been implemented in AutoPilot. The scheduler introduces soft constraints and formulates the problem using techniques described in Chapter 3. We make comparisons to three other approaches:

1. a baseline scheduler using the SDC formulation without operation gating,
2. the iterative algorithm described in [25] for operation gating,
3. an integer-linear programming (ILP) formulation to handle binary penalty exactly for optimal operation gating.

We do not compare our approach with the original work on operation gating in [117], because [25] is algorithmically similar to [117], but with an improved strategy. All these approaches are implemented in C++, and the programs run on a workstation with four 2.4GHz 64-bit CPU and 8G primary memory.

The ILP formulation (for the purpose of optimality study) is briefly described as follows. In addition to all variables and constraints in the mathematical programming formulation for the scheduling engine based on soft constraints, a variable $m_j$ is introduced for each violation variable $w_j$. We add constraints

$$w_j - N \times m_j \leq 0,$$

$$m_j \in \{0, 1\},$$

where $N$ is a large constant number so that the constraint in Equation 4.53 can always be satisfied when $m_j = 1$. Then $m_j$ is introduced to the objective with a coefficient reflecting the cost of violating soft constraint $j$. We also explicitly enforce the constraint
that every variable is an integer. It is easy to verify that in the solution of the ILP formulation, we have

\[ m_j = \begin{cases} 
1 & \text{when } w_j > 0, \\
0 & \text{otherwise.} 
\end{cases} \]

After scheduling, a binding algorithm described in [31] is performed. The RTL code generated by the behavioral synthesis tool is fed to the Magma Talus RTL-to-GDSII toolset. Gate-level simulation under typical input vectors is performed using the Aldec Riviera simulator to obtain power dissipation. All designs are implemented using a TSMC 90nm standard cell library. In this experiment the actual operation gating is carried out by the clock gating on the output registers of the gated operations. Further power savings can potentially be achieved if we apply additional low-power techniques (e.g., operand isolation, feeding sleep vectors for leakage reduction).

Several designs in arithmetic and multimedia applications are used in our experiments. Characteristics of these designs are given in Table 4.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>#node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>88</td>
<td>address space translation unit</td>
</tr>
<tr>
<td>test2</td>
<td>333</td>
<td>Gaussian noise generator</td>
</tr>
<tr>
<td>test3</td>
<td>351</td>
<td>floating-point multiplier</td>
</tr>
<tr>
<td>test4</td>
<td>1306</td>
<td>motion compensation (MPEG4 decoder)</td>
</tr>
<tr>
<td>test5</td>
<td>621</td>
<td>motion estimation (MPEG4 encoder)</td>
</tr>
</tbody>
</table>

Results of the four approaches are reported in Table 4.3. Here, area and power after gate-level implementation are reported for each approach. Since the Magma Talus synthesis tool meets the clock cycle time constraint for all cases, we do not report the frequency for each individual approach. We also normalize the power values to those generated by the approach with soft constraints. For some larger designs, the exact ILP formulation (solved by Cbc [52], a state-of-the-art open-source ILP solver) fails to find
a solution within 7200 seconds. All the three other methods finish within 60 seconds for all cases.

From the results, it is clear that operation gating is a useful technique to create opportunities for power management at the RT level without significant overhead in area. Compared to the SDC scheduling algorithm without considering operation gating, all of the three other methods that optimize for operation gating improve the power dissipation. On average, the method in [25] reduces power by 20.1%, the exact method given by ILP reduces power by 34.6%, and our proposed method reduces power by 33.9%. The reduction tends to be particularly significant when the design has a complex control structure.

Compared to [25], our proposed approach further reduces total power dissipation by an average of 17.1%—a result of being able to consider all opportunities for operation gating simultaneously and optimize globally. The approximation of binary penalty function turns out to work very well—the results generated using our approach are very close to those by the exact formulation, and the observed optimality gap in terms of power is about 1%. At the same time, our method is much more scalable than the exact formulation.
Table 4.3: Experimental results for operation gating.

<table>
<thead>
<tr>
<th>design</th>
<th>cycle</th>
<th>SDC [36]</th>
<th>iterative [25]</th>
<th>ILP</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>area</td>
<td>power</td>
<td>area</td>
<td>power</td>
</tr>
<tr>
<td>test1</td>
<td>2.0</td>
<td>5628</td>
<td>1.16</td>
<td>2.32</td>
<td>5771</td>
</tr>
<tr>
<td>test2</td>
<td>3.5</td>
<td>72127</td>
<td>2.83</td>
<td>1.40</td>
<td>72258</td>
</tr>
<tr>
<td>test3</td>
<td>3.5</td>
<td>143546</td>
<td>7.15</td>
<td>1.39</td>
<td>150335</td>
</tr>
<tr>
<td>test4</td>
<td>3.0</td>
<td>57850</td>
<td>5.03</td>
<td>1.38</td>
<td>57983</td>
</tr>
<tr>
<td>test5</td>
<td>3.0</td>
<td>66341</td>
<td>9.36</td>
<td>1.27</td>
<td>66137</td>
</tr>
<tr>
<td>geomean</td>
<td></td>
<td>1.51</td>
<td>1.21</td>
<td>0.99</td>
<td></td>
</tr>
</tbody>
</table>

Cycle is in ns, area is in $\mu m^2$, and power is in mW. The $N_p$ column is the normalized power.
4.6 Remarks

One might think that after partial dead code elimination in the predicated form [14,141], the predicate of every instruction is equal to its behavior-level observability condition because no redundant instruction will be executed along every control path. However, this is not true. Consider two Boolean values used in a Boolean and instruction, the behavior-level observability condition for either instruction could contain a term about the other. If behavior-level observability conditions are applied as predicates, there will be a cyclic dependency between the two instructions, and the code becomes illegal. Thus, from the perspective of behavior-level observability, one can always find instructions that are unnecessarily executed (unobservable) under certain conditions, even after thorough compiler optimization. Since the execution of unnecessary instructions cannot be avoided completely, profile-guided optimization is needed to minimize the cost of unnecessary execution.

Notably, using behavior-level observability to guide scheduling gives us opportunities to unify both speculative scheduling and control flow restructuring, as shown in Figures 4.1 and 4.5. Previous efforts using predicates in hyperblock scheduling also allow speculative scheduling through predicate promotion [108], and have the ability to simplify program decision logic using knowledge about relations between predicates [5], but a post-processing pass like predicated partial dead code elimination is needed to strengthen some predicates after scheduling to fully realize the equivalent transformation. Behavior-level observability could provide more information to the scheduler than predicate, and can be helpful when various tradeoffs are performed by the scheduler under tight constraints like latency/throughput.

The work in [162] introduced the concept of behavior-level observability and used behavior-level ODC to strengthen conditions for operand isolation and clock gating. However, the algorithm did not consider correlations (between Boolean values, and between data values among a network of select operation), and thus did not capture
opportunities for control-flow restructuring. In addition, the work was not intended to
guide architectural exploration in behavioral synthesis.

Let us, again, consider the example code in Figure 4.3. In practice, an experienced
designer may optimize the design to one in Figure 4.8, where a redundant Boolean
value is introduced in the hope that it can be used in observability computation for
further avoiding multiplications. The Boolean value 
\[(a \land b) \& 0xFFFFFFFFB\] 
\(== \) 0xA is a necessary condition for \(a * a + b * b == 100\). A technique to
add such Boolean guards has been developed in [58] for embedded compilers. We
believe that when this technique is applied, our proposed approach on operation gating
can be more effective in power reduction.

```
unsigned module(unsigned a, unsigned b,
unsigned c, unsigned d) {
  if (a == c) {
    if (a * a + b * b == 100)
      return c * d;
  }
  return a + b;
}
```

```
unsigned module(unsigned a, unsigned b,
unsigned c, unsigned d) {
  if (a == c) {
    if (((a ^ b) & 0xFFFFFFFFB) == 0xA)
      if (a * a + b * b == 100)
        return c * d;
  }
  return a + b;
}
```

Figure 4.8: Transformation to get an even better implementation.
CHAPTER 5

Metrics for Layout-Friendly High-Level Synthesis

5.1 Motivation

Prior to the 1990s, IC designers often ignored the cost of interconnects because transistors were the dominating factor for performance, area and power. However, the situation has changed greatly. Since the 1990s, technology scaling has led to the increasing difficulty in resolving interconnect problems. This is due to relatively scarce routing resources and large interconnect delays. Various efforts to solve these problems have reshaped almost every aspect of the IC industry in the past twenty years. Advanced process technologies have offered new materials and more metal layers. 3-D integration provides a further opportunity to reduce the length of interconnects. In the EDA community, early work mainly focused on timing and congestion optimization during layout [28, 104, 110, 112]. As interconnect problems worsened, repairing failures only during layout was no longer a sufficient remedy. With the prevalence of RTL-based design flows, early interconnect estimation and optimization techniques during logic synthesis have been proposed [96, 100, 106, 123]. However, their capabilities are also limited, especially when the logic is already highly optimized and balanced. It is reported that even with more than 30% of white space, some designs still cannot be routed successfully. Intuitively, some RTL structures are easier to layout than others. If an RTL model is layout-friendly, any reasonable downstream tool will probably get a good result. On the other hand, a highly connected netlist will likely cause headaches to any downstream tool. Thus, to address the interconnect challenges, a well-structured RTL model is needed.
We consider interconnect optimization to be a vital factor to the success of high-level synthesis, because actions the designer could take to circumvent timing and routability failures tend to seriously undermine the advantages of high-level synthesis.

A straightforward approach to fixing timing and routability issues is to manipulate the RTL code directly, but this requires an understanding of the generated code, and is time-consuming and error-prone. Some tools allow specification of explicit clock boundaries and/or sharing decisions. For example, the Handel-C language requires cycle-accurate input [3]. While this approach is useful when fine-tuning is needed, it is questionable whether the level of abstraction is really raised with this method. In the extreme case, input to the high-level synthesis tool is just an RTL model specified in a high-level language. Furthermore, code reuse and design space exploration can become more difficult when decisions on scheduling/sharing are included in the specification. Another common practice is to add an extra timing margin to tolerate excessive interconnect delays. This solves the timing problem to some extent, but not always, because delay of a long interconnect can exceed the target clock period, especially when the synthesis engine is not sophisticated enough concerning interconnect complexity. This approach often leads to unnecessarily long latency and low throughput, and is thus undesirable in many applications, such as signal processing. In addition, a large timing margin can cause overhead in area and power because the synthesis tool needs to use faster components and insert more registers.

Therefore, a high-level synthesis tool needs to manage interconnect complexity intelligently in order to fully realize its advantages. In fact, optimization of global interconnect is most effective when performed during architectural exploration. While downstream designs tools often have a hard time fixing interconnect problems on some difficult netlists, the RTL designer is often able to revise the design and alleviate the problems, e.g., by allowing more time for the data transfer using pipelined multicycle interconnects, or by redesigning the global datapath topology. Such revisions are performed in many RTL-based iterative design flows. As the level of abstraction moves
up to the behavior level, it is possible for a high-level synthesis tool to automatically create layout-friendly RTL structures and reduce unnecessary design iterations.

On the other hand, interconnect optimization during high-level synthesis is challenging due to the absence of information about the gate-level netlist and the layout. Most existing solutions to the problem can be categorized as follows.

1. Use a regular architecture for global interconnects. The chip area is typically divided into regular islands (or clusters), where inter-island data transfers are performed on regular multicycle interconnects, such as a mesh [30, 98, 125]. This style may have been influenced by systolic arrays [98]. The approach is effective for managing global interconnects; yet the regularity is a strong unnecessary constraint and can lead to suboptimal performance and resource usage.

2. Incorporate a rough layout. There are numerous efforts to combine high-level synthesis with floorplanning to help interconnect estimation and optimization [45, 50, 64, 165, 170]. This is quite a natural approach and can potentially work well. However, since layout itself is nontrivial, implementation of a stable and fast layout engine in the inner iteration of microarchitecture optimization is a challenge.

3. Use structural metrics to evaluate netlist structures. Since interconnect timing and routability depend largely on the netlist structure, A high-level synthesis tool can easily explore many different microarchitectures, guided by structural metrics. Such metrics are usually derived from a graph representation of the netlist without performing layout. Widely used structural metrics include total multiplexer inputs [26, 75, 86, 111], number of global interconnects [31, 124], adhesion [96], etc. These metrics generally lead to efficient heuristics. Good correlation with post-layout interconnect complexity is the key for such a metric.

In this dissertation we study the effectiveness of a few structural metrics for modeling interconnect complexity in high-level synthesis. We first introduce a new structural
metric called *spreading score* based on the following intuition: components in a layout-friendly netlist can often be spread apart from each other without introducing many long wires. Spreading score captures these properties in a graph embedding formulation, and it can be estimated efficiently using a semidefinite programming (SDP) relaxation. Compared to the approach of performing a layout, the metric is stable and fast, because the globally optimal solution to the SDP problem can be obtained in polynomial time. Compared to previous structural metrics, it is more layout-oriented. We also present another metric based on neighborhood population density. Our experiments show that spreading score has a better correlation with post-layout wire length, and that weighted neighborhood population and total multiplexer inputs are also reasonable metrics in many cases.

### 5.2 Spreading Score

In high-level synthesis, the synthesis engine performs module selection, operation scheduling and resource sharing, and then generates a microarchitecture-level netlist.¹ The netlist often consists of components (including functional units, registers, memories, I/O ports, multiplexers, pre-synthesized modules, etc.) and wires which connect the components. To simplify the discussion, we first consider the simple case where each component has only one output port.

We construct a directed graph $G = (V, E)$ to model the component-level connectivity, where $V = \{1, 2, \ldots, n\}$ is the set of vertices with each representing a component, and $E \subseteq V \times V$ is the set of directed edges with each representing a wire from the source component to the sink component. Note that an edge is present only when there are data transfers between the two components; if two components are connected in the netlist only because they are both sinks of a net, no edge is created between the corresponding vertex pair. In addition, connections from a component to itself are dis-

¹Unless otherwise noted, the term netlist refers to a microarchitecture-level netlist in this chapter.
carded to avoid self-loop in the graph; this is reasonable because such connections can be regarded as local interconnects within the component.

A layout of the netlist is an embedding of $G$ in the 2-dimensional Euclidean space $\mathbb{R}^2$. Each vertex $i$ is associated with a column vector $p_i = (x_i, y_i)^T$ to represent its position in the embedding. The length of the connection $(i, j) \in E$ can be measured as the Euclidean distance in $\mathbb{R}^2$, i.e., $\|p_i - p_j\| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Consider the following optimization problem.

$$\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} w_i \|p_i\|^2 \\
\text{subject to} & \quad \sum_{i=1}^{n} w_i p_i = 0 \\
& \quad \|p_i - p_j\| \leq l_{ij} \quad \forall (i, j) \in E
\end{align*}$$

(5.1)

Here $w = (w_1, w_2, \ldots, w_n)^T$ is the nonnegative weight vector with $w_i$ being the area of component $i$; $l_{ij}$ is the maximum allowed length for the wire connecting $i$ and $j$. The objective function measures how far components are spread from their weighted center of gravity, using a weighted 2-norm of the distance vector. Thus the problem in Equation 5.1 asks to maximize component spreading, under the constraint that the length of every connection $(i, j) \in E$ does not exceed $l_{ij}$.

With the proper selection of $l_{ij}$, we claim that the optimal value of the above problem can be used to evaluate the layout-friendliness of a netlist. This is based on the following observation: if components in a netlist can be spread over the chip area without introducing long wires, it will be easy for the layout tool to obtain a legal placement of the components without introducing large interconnect delays or using excessive routing resources.

This argument can be supported by examining well-known hand-designed interconnect topologies. For example, mesh [30, 98], ring [87] and counterflow pipeline [150] can all spread without long interconnects, and they are regarded as scalable and layout-friendly topologies; on the other hand, spreading the full crossbar or hypercube on the 2D plane inevitably introduces long interconnects, and these topologies are known to
be much more expensive in interconnect cost. One way to select $l_{ij}$ is based on the estimated sizes of components, $l_{ij} = \sqrt{w_i} + \sqrt{w_j}$.

It is difficult to solve the problem in Equation 5.1 directly, because maximizing a convex function (like the objective function in Equation 5.1) is generally NP-hard (note that minimizing a convex function is easy). We hereby propose a tractable relaxation and use the solution of the relaxed problem to estimate the spreading score.

Consider the graph $G$ with $n$ vertices, we use a $2 \times n$ matrix $P = (p_1, p_2, \ldots, p_n)$ to represent its embedding in $\mathbb{R}^2$, i.e.,

$$
P = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\
y_1 & y_2 & \cdots & y_n \end{pmatrix}.
$$

Let $Q = P^T P$. Then $Q$ is a symmetric semidefinite matrix with rank at most 2, and

$$
Q_{ij} = p_i^T p_j = x_i x_j + y_i y_j.
$$

We can use $Q$ as variables in the formulation in Equation 5.1 without losing any useful information, as indicated by the following result.

**Lemma 5.1.** Given a semidefinite matrix $Q$, the embedding of the graph $G$ can be uniquely decided up to rotation, in the sense that the distance between any pair of vertices is decided.

**Proof.** Since $Q$ is semidefinite, we can perform a Cholesky decomposition and get matrix $U = (u_1, u_2, \ldots, u_n)$, so that $Q = U^T U$. Let $P = (p_1, p_2, \ldots, p_n)$ be another matrix such that $Q = P^T P$. We have $\|p_i - p_j\|^2 = (p_i - p_j)^T (p_i - p_j) = Q_{ii} + Q_{jj} - 2Q_{ij}$, and $\|u_i - u_j\|^2 = (u_i - u_j)^T (u_i - u_j) = Q_{ii} + Q_{jj} - 2Q_{ij}$. Thus $\|p_i - p_j\| = \|u_i - u_j\|$; this means that pairwise distances between vertices are decided given $Q$. \hfill $\square$

Using Equation 5.3, we can rewrite the objective and constraint functions in Equa-
tion 5.1 as follows.

\[ \sum_{i=1}^{n} w_i \|p_i\|^2 = \sum_{i=1}^{n} w_i Q_{ii} = \langle \text{diag}(w), Q \rangle \] (5.4)

\[ \left\| \sum_{i=1}^{n} w_i p_i \right\|^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} w_i w_j Q_{ij} = \langle ww^T, Q \rangle \] (5.5)

\[ \|p_i - p_j\|^2 = Q_{ii} + Q_{jj} - 2Q_{ij} \]

\[ = \langle (e_i - e_j)(e_i - e_j)^T, Q \rangle \]

\[ = \langle K^{ij}, Q \rangle \] (5.6)

Here \( \text{diag}(w) \) is the \( n \times n \) diagonal matrix with \( w \) on its diagonal. \( e_i \) is the \( i \)th standard basis vector in \( \mathbb{R}^n \) and we define matrix \( K^{ij} = (e_i - e_j)(e_i - e_j)^T \) to simplify the equations. \( \langle X, Y \rangle \) is the element-wise inner product (Frobenius inner product) of matrices \( X \) and \( Y \), i.e.,

\[ \langle X, Y \rangle = \sum_{i} \sum_{j} X_{ij} Y_{ij} = \text{trace}(X^T Y) = \text{trace}(Y^T X). \] (5.7)

Then we can rewrite the problem in Equation 5.1 to use \( Q \) as variables.

maximize \( \langle \text{diag}(w), Q \rangle \)

subject to \( \langle ww^T, Q \rangle = 0 \)

\[ \langle K^{ij}, Q \rangle \leq t_{ij}^2 \quad \forall (i, j) \in E \] (5.8)

\[ Q \succeq 0 \]

\[ \text{rank}(Q) \leq 2 \]

Note that the above formulation is equivalent to that in Equation 5.1, and is thus equally hard. Yet if we discard the rank constraint, the resulting problem (shown in Equation 5.9) is much easier.

maximize \( \langle \text{diag}(w), Q \rangle \)

subject to \( \langle ww^T, Q \rangle = 0 \)

\[ \langle K^{ij}, Q \rangle \leq t_{ij}^2 \quad \forall (i, j) \in E \] (5.9)

\[ Q \succeq 0 \]
The above problem is convex; in fact, it is a semidefinite programming (SDP) problem. Like linear programs, SDP problems can be solved optimally in polynomial time, and efficient solvers have been developed in recent years. Regarding the field of convex optimization and SDP, interested readers may refer to books and survey papers such as [16, 157].

The process of deriving the above convex relaxation is a standard technique when dealing with Euclidean distance matrices. The set of $n \times n$ Euclidean distance matrices form a convex set, thus enabling the rank relaxation technique [39]. Similar techniques have been used in problems including sensor network localization [13, 163] and nonlinear dimensionality reduction by maximum variance unfolding [164].

The problem in Equation 5.9 essentially asks for an embedding in $\mathbb{R}^n$ instead of $\mathbb{R}^2$, and thus its optimal value is the lower bound of the spreading score. It would be interesting to see how good the bound is. For this, we refer to the following result.

**Theorem 5.1** (Göring, Helmberg and Wappler [61]). For the relaxed formulation in Equation 5.9 with $p_i \in \mathbb{R}^n$, an optimal embedding always exists in $\mathbb{R}^{tw(G)+1}$, where $tw(G)$ is the tree-width [139] of $G$.

The result implies low distortion when the optimal solution in $\mathbb{R}^n$ is embedded back in $\mathbb{R}^2$. In the extreme case where $tw(G) = 1$ (i.e., the netlist is a tree), an optimal solution always exists in $\mathbb{R}^2$, and the relaxation is exact. This can also be empirically explained as follows: for a vertex $i$ at position $p_i$, the direction from origin (weighted center of gravity) to $p_i$ is the direction of steepest ascent for the objective function, and this direction is within the vector subspace spanned by existing direction vectors, because $p_i = -\frac{1}{w_i} \sum_{j \neq i} w_j p_j$; thus the objective function intrinsically prefers moves that do not increase the dimension of the embedding.

While spreading score characterizes the layout-friendliness of a given netlist, using it directly to compare different netlists tends to favor netlists with larger area, because

---

\[ A \text{ Euclidean distance matrix } D \text{ is a matrix in } \mathbb{R}_{++}^{n \times n}, \text{ with } D_{ij} = ||x_i - x_j||^2, x_i, x_j \in \mathbb{R}^n. \]
more components and larger weights can increase the spreading score naturally. To
avoid this problem, we can normalize the spreading score of a netlist against that of a
uniform mesh with the same area, and use the resulting value in comparison.

Consider a 2-dimensional $m \times m$ uniform mesh with unit component weight and
unit interconnect length between neighboring components, the total weight $W = m^2$.
Assuming $m$ is odd and $r = \frac{m-1}{2}$, we can calculate its spreading score as
\[
\sum_{i=-r}^{r} \sum_{j=-r}^{r} (i^2 + j^2) = \frac{2}{3} r(r + 1)(2r + 1)^2
\]
\[
= \frac{1}{6} (W^2 - W) = O(W^2).
\] (5.10)
This means that the spreading score of a mesh grows quadratically with regard to the
total area. Thus, we can divide the spreading score by $(\sum_{i=1}^{n} w_i)^2$ to get a normalized
value, which is used to compare different netlists without a bias on area.

5.3 Weighted Neighborhood Population

In the graph representation of the netlist as described in Section 5.2, for a pair of nodes
$i, j \in V$, let $\text{dist}(i, j)$ be the distance between $i$ and $j$ in the graph, where edges are
considered undirected and have a unit length. We define the $k$-neighborhood of $i$ as the
set of nodes reachable from $i$ within $k$ distance, i.e., $S_i^k = \{ j \in V | \text{dist}(i, j) \leq k \}$. The
weighted neighborhood population of $i$ can be defined as
\[
\text{wnp}_i = \sum_{k=1}^{n} \frac{|S_i(k)|}{k^2}
\] (5.11)

The intuition behind this definition is that if a netlist can be placed without long
wires, the size of any $k$-neighborhood should grow at most quadratically with $k$ to
maintain a bounded density. The $k^2$ term reflects that the placement is performed on
$\mathbb{R}^2$, as opposed to $\mathbb{R}^n$.

Weighted neighborhood population can be regarded as a generalization of direct
fanin/fanout to all distances. Components closer to $i$ have a larger weight, because
there is limited space near \( i \) to place these components. To characterize the netlist with a single number, we use the maximum of the weighted neighborhood population of all nodes.

\[
WNP = \max_{i=1}^{n} wnp_i \tag{5.12}
\]

### 5.4 Previous Metrics

#### 5.4.1 A Metric Based on Cut Size

Cut size has been recognized as an indicator of interconnect complexity in a number of important synthesis steps, such as partitioning, clustering, and placement. Intuitively, smaller cut size implies less global connections, and thus better routability.

The cut-size minimization process is explicitly applied in every partitioning-based placer (e.g., Capo [140]). In fact, if we divide the placement region into unit squares, the total cut size of a placed netlist on the edges of these squares is approximately equal to the total wire length. Thus, the recursive cut-size minimization process in partitioning-based placers can be viewed as an approximate wire length minimization process. Moreover, the cut size across a local cut line directly captures the local congestion, and maintains better congestion information than the total wire length metric [143].

Kudva, Sullivan and Dougherty propose the sum of all-pairs min-cut (SAPMC) to evaluate the adhesion of a gate-level netlist, and use it in logic synthesis [96]. In a netlist represented by graph \( G = (V, E) \), the cut size of the an \( s-t \) min-cut between two distinct vertices \( s \) and \( t \) is the minimum number of nets whose removal disconnects \( s \) and \( t \). The SAPMC is the sum of min-cut sizes for all pairs of distinct vertices in \( V \). Experimental results reported in [96] show a positive correlation between congestion and SAPMC; in addition, the results indicate that the correlation between congestion and the circuit size (as measured by the number of nodes in the netlist) is even stronger,
and thus SAPMC is only used to break ties in optimization. Clearly, SAPMC is also positively correlated with the circuit size, because larger circuits naturally have more distinct node pairs, which lead to larger SAPMC. In an effort to obtain orthogonal metrics, we try to exclude the node count factor from SAPMC, and instead compute the average min-cut (AMC), by dividing SAPMC with \( n(n-1)/2 \), where \( n \) is the number of nodes.

### 5.4.2 Total Multiplexer Inputs

In xPilot, we generate architectures with the multiplexer-based global interconnect structure (instead of tri-state buses). A \( m \)-to-1 multiplexer is introduced before an input port of a component \( c \) when \( m \) upstream components can drive the port (in different control states). This is often the result of resource sharing. Note that resource sharing does not always introduce multiplexers—when the \( m \) upstream operations/variables also share a component, the multiplexer before the input port of \( c \) is not needed, and in such a case, the interconnects are also shared. This is highly desirable, as it reduces area without increasing interconnect complexity. Therefore, total multiplexer inputs are commonly used when comparing different netlists [26, 75, 86, 111].

### 5.5 Experimental Evaluation

In this section we compare the effectiveness of different structural metrics in modeling and guiding the optimization of wire length. Post-layout wire length is measured after feeding the microarchitecture-level netlist to a mixed-size placer, mPL6 [23], which optimizes the total wire length in the layout. In all cases, a square region is used for placement, with a target density of 80%; i.e., 80% of the placement region will be covered by the components, leaving 20% white spaces. We also allow I/O pins to be placed at any location in the layout area. Note that we do not perform logic synthesis in this set of experiments, and thus a component corresponds to a macro cell in placement;
in addition, all pins of a component are assumed to be at the center of the component when wire length is measured. An open-source SDP solver, SDPA 7.3.6 [167], is used when calculating spreading scores.

Several datapath-intensive kernels are extracted as test cases from DSP applications. These designs generally perform many arithmetic operations such as multiplications, additions and subtractions, and thus have plenty of flexibility in choosing from many possible datapath topologies with different resource usage and interconnect complexity.

5.5.1 Correlation with Wire Length

For each design, we generate multiple netlists from the synthesis engine by using different scheduling/binding strategies, and by recording the intermediate solution in incremental optimization. Single-variable linear regressions between each pre-layout structural metric (spreading score, AMC, weighted neighborhood population, total multiplexer inputs) and the post-layout average/total wire length is performed and plotted, as shown in Figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6.

From the regressions, we observe consistently strong negative correlations between spreading score and wire length. Weighted neighborhood population shows strong positive correlations with wire length in test cases 1, 5, and 6, but doesn’t point to the right direction in cases 2 and 3. Total multiplexer inputs generally show positive correlations with wire length as well, but in some cases, the correlation is vague. The average min-cut seems a very weak indicator of wire length and and the correlation is inconsistent for different designs. Overall, we consider spreading score as a preferred measurement of interconnect complexity.
Figure 5.1: Linear regression for test1.
Figure 5.2: Linear regression for test2.
Figure 5.3: Linear regression for test3.
Figure 5.4: Linear regression for test14.
Figure 5.5: Linear regression for test5.
Figure 5.6: Linear regression for test6.
5.5.2 Interconnect Optimization

Here we try to evaluate the effectiveness of the metrics in guiding the optimization engine in high-level synthesis toward a better-structured netlist. We have implemented a simulated annealing algorithm to perform microarchitecture exploration in xPilot. Based on an initial solution, perturbations are performed to generate alternative microarchitectures. Feasibility and cost are evaluated to decide whether the new solution is accepted. Feasibility check includes legality check (for dependency violation, resource hazard, combinational loop, and performance constraint) and timing check (without considering interconnect delay). Area and timing information about components are obtained from a pre-characterized library.

The cost function to be minimized involves both total area and interconnect complexity.

\[
\text{cost} = \text{area} + \alpha \times \text{interconnect\_cost}
\]  

Here area is estimated by adding up the area of datapath components, and interconnect cost is estimated using one of the structural metrics. \(\alpha\) is a constant that can be adjusted to allow different tradeoffs between area and interconnect cost. Also note that for spreading score, larger value implies smaller wire length, and thus \(\alpha < 0\); while for other metrics, \(\alpha > 0\), because a larger value tends to imply a more tightly coupled netlist structure.

We perform the following types of random perturbations on the netlist.

- Move an operation from one functional unit to another.
- Move a variable from one register to another.
- Merge two functional units or registers.
- Reschedule an operation one cycle earlier or later, and update the schedules of related operations if necessary. We do not allow changes in latency for this operation.
In our implementation, we try to reduce the chance of generating an illegal or trivial perturbation (one that leads to essentially the same netlist as the original one) by performing checking in the process of generating perturbations.

For each design, we repeat the experiments multiple times with different $\alpha$, as a way to characterize the Pareto-optimal points in the design space. Note that it is not meaningful to compare different structural metrics with a constant $\alpha$, because values of different metrics are not comparable. Instead, we try to characterize the Pareto-optimal curves generated by using different metrics. The results are shown in Figures 5.7, 5.8, 5.9, 5.10, 5.11, 5.12.

![Figure 5.7: Total area and average wire length of solutions for test 1.](image)

The results indicate a clear tradeoff between area and wire length in most cases. This conforms our intuition that resource sharing, which reduces area, can lead to a more tightly coupled microarchitecture, because connections are often introduced between previously unconnected components to allow resource sharing. In our experiments with vastly different $\alpha$ (absolute value of $\alpha$ ranging from 0.1 to 1000 in our
Figure 5.8: Total area and average wire length of solutions for test 2.

Figure 5.9: Total area and average wire length of solutions for test 3.
Figure 5.10: Total area and average wire length of solutions for test 4.

Figure 5.11: Total area and average wire length of solutions for test 5.
Figure 5.12: Total area and average wire length of solutions for test 6.

experiments), we obtain solutions with larger area when we set larger weights on interconnect cost, particularly when spreading score or total multiplexer inputs are used to estimate interconnect complexity.

When a large weight is applied on interconnect complexity, both spreading score and total multiplexer inputs can lead to a solution with short wire length. However, the metrics of average min-cut and weighted neighborhood population do not consistently do so even with very large weights; they probably consider some inferior interconnect topologies as good ones. Thus, spreading score and total multiplexer inputs are more effective in preventing a solution with average long wire length.

With large weights on interconnect cost, we often get very large area when using the total multiplexer inputs. This is because we can rarely achieve any significant resource sharing without introducing multiplexers; thus a large penalty for multiplexers will effectively prevent sharing. On the other hand, there exist solutions that exploit a moderate amount of sharing opportunities while still leading to short average wire length.
The use of spreading score seems to enable the exploration of such opportunities. In test1, test2, and test 5, the solutions obtained using spreading score achieve the shortest average wire lengths among all solutions, and they are still more than 50% smaller than the solutions with little sharing. Therefore, we consider spreading score superior to total multiplexer inputs, because it helps to avoid the pessimism in interconnect estimation when sharing resources. Spreading score is derived considering the global netlist structure, and this may help explain the fact that it better characterizes interconnect complexity than total multiplexer inputs, which is only a first-order approximation.

5.6 Extension to Interconnect Delay Budgeting

Note that increasing the allowed wire length $l_{ij}$ can often lead to a better spreading score in Equation 5.1, and this explains why an extra timing margin can help layout. However, $l_{ij}$ is limited by timing constraints in practice. To capture first-order timing information, we can extend the formulation in Equation 5.1.

We use $d_{ij}$ to denote the delay of the wire $(i, j) \in E$, and consider it to be a monotone single-variate function of wire length

$$d_{ij} = D(l_{ij}).$$  \hspace{1cm} (5.14)

Two additional variables $t_i$ and $\tau_i$ are attached to each $i \in V$, where $t_i$ denotes the arrival time (after an clock edge) at input ports of $i$, and $\tau_i$ denotes the arrival time at the output port. We then have

$$\tau_i + d_{ij} \leq t_j, \forall (i, j) \in E.$$  \hspace{1cm} (5.15)

If the corresponding component is combinational, we use $d_i$ to denote its delay, and then

$$\tau_i = t_i + d_i.$$  \hspace{1cm} (5.16)
Otherwise, if the component is sequential, $\tau_i$ will be a constant, and $t_i$ should be bounded by the required time at the input of $i$, $T_i$, that is,

$$t_i \leq T_i.$$  \hspace{1cm} (5.17)

For a register $i$, $T_i$ can be regarded as equal to the clock period subtracted by the setup time. A primary output also has a required time depending on the interface timing specification.

We can then treat $l_{ij}$ as variables and optimize spreading under timing constraints.

$$\text{maximize } \sum_{i=1}^{n} w_i \|p_i\|^2$$

subject to

$$\sum_{i=1}^{n} w_i p_i = 0$$

$$\|p_i - p_j\| = l_{ij} \quad \forall (i, j) \in E$$

$$d_{ij} = D(l_{ij}) \quad \forall (i, j) \in E$$

$$\tau_i + d_{ij} \leq t_j \quad \forall (i, j) \in E$$

$$\tau_i = t_i + d_i \quad \forall \text{ combinational } i$$

$$t_i \leq T_i \quad \forall i \in V$$

The formulation in Equation 5.18 effectively combines interconnect slack allocation with node spreading, and captures both structural property and timing property of the netlist.

The graph construction and labeling procedure can be extended easily for practical considerations. For example, for a component with multiple input ports and multiple output ports, if the delay varies significantly between different inputs and outputs, we can create a vertex for each port, so that the delay between each pair of ports can be characterized individually as done in [97]; constraints on the distances between ports can be enforced to keep the geometry of the component. Similar treatment on a very large component can make the estimation of interconnect length aware of port positions, instead of regarding all ports as being located at the center of the component. The required time at a port can be manipulated easily to capture nontrivial situations like multicycle paths, multiple clock domains, or other complex I/O timing requirements.
Using the same reformulation and relaxation technique presented in Section 5.2, we can rewrite the problem in Equation 5.18 to use $Q$ as variables.

$$\begin{align*}
\text{maximize} & \quad \langle \text{diag}(w), Q \rangle \\
\text{subject to} & \quad \langle w w^T, Q \rangle = 0 \\
& \quad \langle K^{ij}, Q \rangle = l_{ij}^2 \quad \forall (i, j) \in E \\
& \quad d_{ij} = D(l_{ij}) \quad \forall (i, j) \in E \\
& \quad \tau_i + d_{ij} \leq t_j \quad \forall (i, j) \in E \\
& \quad \tau_i = t_i + d_i \quad \forall \text{combinational } i \\
& \quad t_i \leq T_i \quad \forall i \in V \\
& \quad Q \succeq 0
\end{align*}$$

(5.19)

When a quadratic delay model is used, that is,

$$D(l_{ij}) = \alpha l_{ij}^2,$$

(5.20)

where $\alpha$ is a constant that depends on technology. We get the following formulation.

$$\begin{align*}
\text{maximize} & \quad \langle \text{diag}(w), Q \rangle \\
\text{subject to} & \quad \langle w w^T, Q \rangle = 0 \\
& \quad \tau_i + \alpha \langle K^{ij}, Q \rangle \leq t_j \quad \forall (i, j) \in E \\
& \quad \tau_i = t_i + d_i \quad \forall \text{combinational } i \\
& \quad t_i \leq T_i \quad \forall i \in V \\
& \quad Q \succeq 0
\end{align*}$$

(5.21)

This problem is still convex, because only linear inequality constraints and positive semidefinite constraints are present, and both types of constraints define a convex feasible region.

The quadratic delay model is used in Equation 5.21 to simplify the relaxation. It is also possible to use the linear delay model, but that leads to more variables and less sparse matrices in the formulation. The quadratic delay model is not an inferior choice because it is more accurate than its linear alternative when no repeaters are
inserted [27]. During microarchitecture optimization, the synthesis engine could ex-
plicitly insert repeaters in the netlist in favor of spreading score, with consideration of
their area/timing overheads.

5.7 Further Discussion

In this section we relate spreading score to other problems, to further study its properties
and to motivate future research.

One may want to measure how far the vertices are spread by looking at pairwise dis-
tances, instead of distances between vertices and their center. Consider the embedding
of \((p_1, p_2, \ldots, p_n)\) centering at \(c = \frac{1}{n} \sum_{i=1}^{n} p_i\), we have

\[
\sum_{i=1}^{n} \|p_i - c\|^2 = \sum_{i=1}^{n} \left( p_i^T p_i + c^T c - 2c^T p_i \right) = \sum_{i=1}^{n} p_i^T p_i - \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} p_i^T p_j,
\]

\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \|p_i - p_j\|^2 = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (p_i - p_j)^T (p_i - p_j) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left( p_i^T p_i + p_j^T p_j - 2p_i^T p_j \right) = n \left( \sum_{i=1}^{n} p_i^T p_i - \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} p_i^T p_j \right).
\]

Thus the two metrics differ only by a constant factor \(n\) in the unweighted version. How-
ever, the weighted sum of square for pairwise distances, i.e., \(\sum_{i} \sum_{j} w_{ij} \|p_i - p_j\|^2\), does
offer more flexibility, because of the larger number of weights \((n^2, \text{ compared to } n\) in
the formulation for spreading score). The reformulation and relaxation techniques can
still be applied to handle the revised formulation with pairwise distances; the only dif-
ference is that the coefficient matrix in the objective function will be dense (as opposed
to diagonal in Equation 5.21). The added flexibility is useful for certain purposes. For
example, when two components are connected by a path with many registers and plenty
of slack, their distance in the embedding is probably long. In such a case, further increasing their distance does not offer a clear advantage, and then we can reduce the corresponding weight in the objective.

We now discuss the relation between the proposed embedding and placement. We consider that our embedding problem is related to the dual of the placement problem in some sense. Roughly speaking, the placement problem asks to minimize wire length, with lower bounds on pairwise distances so that components do not overlap; the embedding problem asks to maximize pairwise distances, with upper bounds on wire length. Such “duality” indicates connections between spreading score and wire length after layout. This further justifies the use of spreading score as an estimator of layout-friendliness.

Kudva, Sullivan and Dougherty propose to use sum of all-pairs min-cut (in contrast to sum of all-pair distance in an embedding) to evaluate the adhesion of a gate-level netlist, as described in Section 5.4.1. Their idea of using a structural metric to guide the generation of a layout-friendly netlist influences our work. However, we use different approaches: their metric is related to techniques used in the analysis of social networks [166], which has roots in classic graph theory; our technique is influenced by the geometric embeddings of graphs and the associated algebraic structures. We consider our metric advantageous in two aspects: (1) it is more layout-oriented;\(^3\) (2) thus it has the potential to capture timing information by relating interconnect delay to distance. On the other hand, the adhesion metric is probably advantageous in another two aspects: (1) it may be evaluated more efficiently;\(^4\) (2) since cut size is used, it may be more closely related to average wire density in layout, and thus help to reduce congestion (which is not measured in this study).

\(^3\)For example, authors of [96] note that a limitation of their adhesion metric is that it cannot consider the impact of relative locations of I/O pins on routability; our embedding formulation can be revised to partly capture such information.

\(^4\)An approximation of adhesion can be obtained in \(O(n^2)\). The complexity of SDP with a fixed error bound is \(O(n^3)\) for the dense case; sparsity and incremental solving can improve scalability for our problem.
Spreading score is indirectly related to cut size as well, as suggested by the following result from [61]. For a simplified version of the problem with \( w_i = 1 \) and \( l_{ij} = 1 \), the dual problem of the relaxation in Equation 5.21 can be transformed to an SDP formulation for calculating \( \frac{|E|}{\hat{a}(G)} \). Here \( \hat{a}(G) \) is the absolute algebraic connectivity of graph \( G \), and it is shown to be related to the node connectivity as well as edge connectivity of \( G \) [51].

A limitation of structural metrics discussed in this chapter is that they do not capture timing related to control signals (from the FSM controller) very well, because the graph representation of the netlist does not have nodes corresponding to the FSM controller. This is because RTL synthesis tools often change the FSM controller drastically in optimization. Capturing controller timing is intrinsically difficult without logic synthesis. In our implementation, we exclude the FSM when constructing the graph; instead, we generate a Moore-style one-hot FSM to alleviate the problem.
CHAPTER 6

Concluding Remarks

Although many previous commercial high-level synthesis tools had only limited success, the repeated efforts after failures are a good indicator of the practical importance of the problem, which cannot be easily handled in alternative ways. The driving force behind high-level synthesis is the trend of IC technology scaling and the need to increase design productivity. As such a trend continues and the tools improve, the benefits of high-level synthesis will become evident to more and more designers.

The recent wave of high-level synthesis tools, by accepting C-based language inputs and applying platform-based approaches, have gained significant success over the past few years. As part of the AutoESL team, I personally witnessed the adoption of high-level synthesis by a number of major semiconductor design houses, and played an active role in several design wins on both ASIC and FPGA platforms. Such experiences have been a major inspiration to the research results presented in this dissertation.

Improving quality of results is a major problem in high-level synthesis, especially in the era of nanoscale ICs where new challenges arise. Designers are often reluctant to switch to high-level synthesis when the tool cannot produce RTL models of a similar quality to hand designs. In fact, a high-level synthesis tool can possibly explore a huge design space automatically, and thus has the potential to beat human designers. To achieve this, the following is needed.

- Effective synthesis engine. Given the complex problem and the huge design space, the tool needs to handle multiple objectives and constraints simultaneously and navigate the solution space efficiently. Simple adaption of a heuristic
for a particular problem may not work well for other problem variants.

- Consideration of new challenges. A synthesis tool targeting nanometer manufacturing technologies need to consider new objectives such as power and yield, in addition to the traditional ones like performance and area.

- Reasonable QoR models. A major difficulty in high-level synthesis is the absence of accurate QoR estimators due to the unknown details at such a high level of abstraction. Without valid models that correlate well with QoR in the final implementation, the optimization engine is often misguided.

Results presented in this dissertation are the outcome of our effort to address the above needs in the past few years. The scheduling optimization engine using soft constraints (Chapter 3) allows flexible specification of various design intentions, and achieves globally optimal solution in polynomial time by exploiting the total unimodularity of the constraint matrix. Our work on behavior-level observability (Chapter 4) is the first thorough treatment of the topic, and the technique is proved to be optimal under the black box abstraction. Behavior-level observability analysis is used to guide operation gating in scheduling, allowing tradeoff between power and other design considerations using soft constraints. The structural metrics (Chapter 5) offer efficient ways to measure interconnect complexity in high-level synthesis, and have good correlations with after-layout wire length.

Along these directions, we consider it promising to blur the boundary between high-level synthesis and RTL synthesis, and perform cross-level optimizations. Many techniques in RTL synthesis, like datapath optimization, logic simplification and technology mapping, could be incorporated in high-level synthesis. The understanding of how the design will be implemented is useful for platform-specific optimizations that lead to best efficiency. Knowledge on the detailed netlist structures also enables more accurate models, and then more effective optimizations can be performed to beat human designers. We believe this will happen soon.
REFERENCES


