SYNTHESIS TECHNIQUES FOR POWER-EFFICIENT INTEGRATED CIRCUITS

A Dissertation

by

CHAOFAN LI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jiang Hu
Committee Members,	Duncan M. Walker
	Gwan S. Choi
	Tie Liu
Head of Department,	Miroslav Begovic

August 2018

Major Subject: Computer Engineering

Copyright 2018 Chaofan Li

ABSTRACT

In the past few years, power efficiency has been increasingly important for integrated circuits. As the Moore's law effects slows down, the improvement of power consumption through scaling of silicon process technology is hitting the limits. At the same time, IC chips are more often embedded into mobile devices, which usually have no outer continuous power supply. The power efficiency is even more critical due to the limited electricity stored in batteries of these mobile devices. Besides, the high-performance ICs used in server farms or data centers also require improved power efficiency to alleviate the heat dissipation of the chips, which causes additional cost to lower the temperature of the facilities. The profit of crypto-currency mining is even directly affected by the electrical energy consumption of the mining hardware including ASICs, GPUs and FPGAs, which accounts for the largest part of the cost. Thus, more techniques for power efficiency were exploited in recent years to achieve further power reduction in addition to that achieved by silicon process advancements.

Among the techniques for improving power efficiency, approximate computing has been recognized as an effective low power technique for applications with intrinsic error tolerance, such as image processing and machine learning. Existing efforts on this are mostly focused on approximate circuit design, approximate logic synthesis or processor architecture approximation techniques. Chapter 2 of this research aims to make good use of approximate circuits at system and block levels. In particular, approximation aware scheduling, functional unit allocation and binding algorithms are developed for data intensive applications. Simple yet credible error models, essential for precision control in the optimizations, are investigated. The algorithms are further extended to include bitwidth optimization in fixed point computations. Experimental results, including those from Verilog simulations, indicate that the proposed techniques facilitate desired energy savings under latency and accuracy constraints.

With their flexibility in allowing reconfiguration for different applications, hardware such as FPGAs have become increasingly preferred over ASICs as a platform for high-performance com-

puting like accelerators. However, this advantage is partially defeated by the time-intensive high-level synthesis (HLS) process and the poor controllability for the synthesized architecture. We propose a fast mapping-based high level synthesis technique friendly to local incremental change. It exploits the SSA (Static Single Assignment) form with array SSA extension and ϕ-function based flow control. It first maps the SSA form based IR to a fully pipelined circuit, then alters the circuit to a partially pipelined or nonpipelined circuit by resource sharing in an optional phase of resource optimization. Pipeline interlocking to address the pipeline hazards is also provided, which has better power-efficiency.

Adaptive Supply Voltage (ASV) is another power-efficient approach to achieving resilience against process variation and circuit aging. Fine-grained ASV offers further power efficiency gains, but entails relatively complex control circuit, which has not been well studied yet. Chapter 4 of this research presents two control design techniques: one is rule-based control derived from network flow optimization and the other is finite state machine control. For the FSM control, a graph-based algorithm that automates the control vector generation is proposed. This research also presents an iterative greedy heuristic for delay sensor deployment in ASV designs. The effectiveness of these techniques is confirmed by experiments performed on ICCAD 2014 benchmark circuits. The results show that our techniques achieve around 20% leakage power reduction compared to coarse-grained ASV, while maintain about the same timing yield.

DEDICATION

To my family

ACKNOWLEDGMENTS

I am very grateful to my advisor Prof. Jiang hu, who guided me during my PhD and led me to the field of computer-aided design for integrated circuits. I appreciated it very much that Prof. Hu gave me this opportunity to explore the frontiers of CAD, especially the high-level synthesis. I am also very grateful to Prof. Sachin Sapatnekar, who gave me many advices for the research. They always shared their insights and experience with me during research.

I want to thank my dissertation committee members, who made their comments on my research so that I could refine my dissertation.

I am very thankful to Wei Luo, who was a master student when I was working on chapter 2 of this research. He was very diligent and helped me finish the experiments on time. Also, I want to thank Ang Lv who shared with me his work on gate clustering so that I could continue the fine-grained control synthesis of adaptive supply voltage systems following his work.

Last but not the least, I want to thank my parents and my aunt, who supported me to study in America for my PhD. Without their support, I couldn't have the opportunity to pursue this PhD degree.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

The chapter 2 was advised by Prof. Jiang Hu and Prof. Sachin Sapatnekar of University of Minnesota. Part of the experiments were performed by Wei Luo. The work was published in [1].

The chapter 3 was advised by Prof. Jiang Hu, Prof. Sachin Sapatnekar. Hongbo Rong of Intel Corporation also provided many comments.

The chapter 4 was also advised by Prof. Jiang Hu and Prof. Sachin Sapatnekar. The work was published in [2].

Funding Sources

Graduate study was supported by a research assistantship from Prof. Jiang Hu. The research of chapter 4 was partially supported by NSF (CCF-1255193, CCF-1525749, CCF-1525925) and SRC (2013-TJ-2421).

NOMENCLATURE

ASV	Adaptive Supply Voltage
ASAP	As-Soon-As-Possible
ALAP	As-Late-As-Possible
ASIC	Application-Specific Integrated Circuit
CTS	Clock Tree Synthesis
CUDA	Compute Unified Device Architecture
CPU	Central Computing Unit
DAG	Directed Acyclic Graph
DSE	Design Space Exploration
DVFS	Dynamic Voltage Frequency Scaling
EDA	Electronic Design Automation
FU	Functional Unit
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GCC	GNU Compiler Collection
GDSII	Graphic Database System II
GNU	GNU is Not Unix
GPU	Graphic Processing Unit
HLS	High-Level Synthesis
HDL	Hardware Description Language
IR	Intermediate Representation
ILP	Integer Linear Programming

LLVM	Low-Level Virtual Machine
LP	Linear Program
MIP	Mixed Integer Programming
P&R	Place & Route
RTL	Register Transfer Level
SSA	Static-Single Assignment
STA	Static Timing Analysis
SDK	Software Development Kit

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGMENTS v

CONTRIBUTORS AND FUNDING SOURCES vi

NOMENCLATURE vii

TABLE OF CONTENTS ix

LIST OF FIGURES x

LIST OF TABLES xi

1. INTRODUCTION 1

1.1 Motivation 1
1.2 Common Synthesis Techniques 2
1.3 High-Level Synthesis for Power-Efficiency 4
1.3.1 HLS for Approximate Circuits 7
1.3.2 Mapping-Based HLS for Pipelined Circuits 7
1.4 Adaptive Circuits 8
1.4.1 Control Circuits Synthesis for Adaptive Supply Voltage 9
1.5 Summary of Contributions 9

2. HIGH-LEVEL SYNTHESIS FOR APPROXIMATE COMPUTIN 11

2.1 Introduction 11
2.1.1 Error Control 13
2.2 Preliminaries 14
2.3 Analytic Error Models 15
2.4 Knapsack-Based HLS for Approximate Computing 21
2.4.1 Knapsack-Based Precision Optimization 21
2.4.2 Approximation-Aware HLS 22
2.4.2.1 Conventional List Scheduling 22
2.4.2.2 Iterative List Scheduling 23
2.5 ILP-Based HLS for Approximate Computing 27
2.6 Experiment 28

2.7 Conclusion 32

3. MAPPING-BASED HIGH-LEVEL SYNTHESIS FOR PIPELINED CIRCUITS 33

3.1 Introduction 33
3.2 Backgrounds 35
3.2.1 Distributed Memories 35
3.2.2 SSA Form 37
3.3 Phase I: Mapping 39
3.3.1 Scheduling 39
3.3.2 Storage Binding for Scalar Variables 41
3.3.3 Datapath Control 42
3.3.4 Synthesis of Array Datapaths 43
3.3.5 Loops 46
3.4 Phase II: Resource Optimization 46
3.4.1 Iterative Resource Sharing 47
3.4.2 Pipeline Interlock Synthesis 49
3.4.3 Sharing for Loops 51
3.5 Support for parallelization 52
3.5.1 Array SSA and Parallelization 53
3.5.2 Structural Recursion 53
3.6 Experimental Results 54
3.7 Conclusions and Future Works 62

4. CONTROL CIRCUIT SYNTHESIS FOR ADAPTIVE SUPPLY VOLTAGE DESIGNS .. 64

4.1 Introduction 64
4.2 Backgrounds on ASV 66
4.3 Problem Formulation 68
4.4 Rule-Based Control 69
4.5 Finite State Machine Control 70
4.5.1 Phase I: Initial Response 71
4.5.2 Phase II: Incremental Responses for FSM 76
4.6 Delay Sensor Deployment 77
4.7 Experiments 80
4.7.1 Adaptive Design Flow 80
4.7.2 Experimental Results 82
4.8 Conclusions 84

5. CONCLUSION 88

REFERENCES 90

APPENDIX A. TRANSISTOR-LEVEL HDL FOR AN APPROXIMATE ADDER 101

APPENDIX B. TRANSISTOR-LEVEL HDL FOR AN ACCURATE ADDER 103

LIST OF FIGURES

FIGURE Page

1.1 Overview of typical IC design flow ... 3

1.2 Example of task graph for high-level synthesis ... 5

1.3 The ASAP scheduling result of task graph in figure 1.2 ... 5

1.4 The ALAP scheduling result of task graph in figure 1.2 ... 6

2.1 The comparison of the accurate and the approximate adder [3] ... 12

2.2 An error from an approximate adder may cancel itself along reconvergent paths. ... 19

2.3 (a)Task graph; (b) Conventional list scheduling results in two multipliers; (c) The two multiplications can share one multiplier. ... 24

2.4 (a)Task graph; (b) Conventional list scheduling results in 3 adders for latency constraint of 3; (c) Two adders are sufficient. ... 24

2.5 Overview of the experiment flow ... 28

2.6 Energy normalized with respect to All-Prcs results. ... 30

2.7 Error standard deviation normalized with respect to standard deviation constraints... 30

2.8 Energy-error tradeoff from ILP result. The MSE results are from Verilog simulations. 31

2.9 Runtime (in log scale) comparison. ... 32

3.1 The storage binding for code 3.2. The circuit is divided into pipeline stages indicated by dashed horizontal lines. Each black font name corresponds to one register in code 3.2 and each gray font name indicates a newly added register for pipeline synchronization or control. Every variable has one pipeline register storing its value. The registers in dashed rectangles, except `Phi`, are for data synchronization in the pipeline. On FPGA, these registers can be implemented by configuring look-up tables (LUT) to shift registers and thus have low cost [4]. ... 40

3.2 Details of the shaded region in Figure 3.1 (stage 2 and 3). The AND operations require that the enable signals such as %if.then.3 are replicated to have the same bit-width as the arithmetic operations. The dashed arrows and boxes indicate an alternative implementation of the Φ function where an enable signal resets the register in a dashed box (with synchronous active-low RST) such that the value is zero and the corresponding AND operation can be avoided. 42

3.3 Example of array SSA: arrays are propagated to different pipeline stages. 45

3.4 Example of resource sharing: the enable signals, similar to the %entry in Figure 3.1, are initialized to 0. (a) The original circuit with no sharing. (b) Two operations sharing the same one adder. This shared adder selects inputs according to the enable signal. The enable signal asserts when the corresponding input is valid. If the enable signals do not assert consecutively, there is no structural hazard. A bubble needs to be inserted between two consecutive valid inputs to avoid hazards. (c) All operations sharing the same operator, equivalent to a loop adding 1 for five times. 55

3.5 Example of interlocked pipelines for the same example in figure 3.4: (a) two operations sharing one operator. (b) another operation added to the periodic sharing list sharing the same operator with the two operations in (a). The $IN(a)$ and $IT(a)$ are the same as the IT and IN in (a) and are updated with Algorithm 3. 56

3.6 Diagram of the basic blocks in code 3.4: the basic blocks in red are part of the loop, which can be shared among different loop cycles. The back edges would also cause pipeline stall, similar to the back edges in figure 3.4. 57

3.7 Bitonic sorter: another example of structural recursion. Each line represents a number. The arrows mean swapping the two numbers. The bitonic sorter is a common structure for sorting in parallel. Figure courtesy of Magnus Manske. 57

3.8 Example of structural recursion. The inputs are recursively decomposed. Each box represents a recursive structure except the upper four terminating box. 58

3.9 The throughput-power comparison among different methods. Each point represents a case. The no-array-SSA results are from modifying our HLS by removing the use of array SSA. 61

4.1 Fine grained ASV using canary flip-flop as delay sensors for detecting process variations and circuit aging. Dashed lines indicate timing paths. 65

4.2 Schematic of voltage interpolation [5]. 67

4.3 Network flow model for assigning sensors to adaptivity blocks. The edge cost between block and sensor vertex is inversely proportional to the overlap between the sensor fanin cone and the block. The red edges indicate a flow solution. 69

4.4 Dominance graph for matrix (4.1) and its pruning, on the assumption that the power overhead of $b3$ and $b6$ combined is less than that of $b2$. 73

4.5 Partial FSM transition diagram for control matrix (4.1). 76

4.6 The paths to FF2 are largely covered by paths to FF1 and FF3. 78

4.7 Impact of ω on the preference metric with horizontal axis for ω and vertical axis for p. If $\omega > 2.6571$, $p_3 > p_2 > p_1$; if $\omega < 2.4857$, $p_1 > p_2 > p_3$. 79

4.8 The flow of experiments 81

4.9 The layout of testcase b19: The power domains are divided according to the gate clustering results 82

4.10 Results for ICCAD 2014 benchmarks. 85

4.11 Timing yield vs. leakage power for circuit b19 with different timing constraints. For each curve, results on the right are from tighter timing constraints. 86

4.12 Comparison among different sensor deployment methods on b19. 86

4.13 Impact of the number of sensors on b19. 87

LIST OF TABLES

TABLE Page

2.1 Existing error metrics 13

2.2 Notations for HLS with approximate computing 16

2.3 Verilog simulation results. 31

3.1 Storage options on Xilinx Virtex 7 series FPGAs: depth and width is when the RAM is configured with maximal bandwidth. 36

3.2 The comparison of post-layout results obtained from our mapping phase alone without resource optimization and a state-of-the-art commercial HLS tool. The Sscan and Pscan are designs for sequential scan and parallel scan as described in [6]. Bsort is a Bitonic sorter of 16 32-bit numbers [7]. SHA256 is 256-bit secure hash algorithm fully unrolled except the outer-most loop. Designs with arrays are marked with *. The average percentages are our mapping results compared with the commercial tool. 59

3.3 The post-route results obtained from our HLS after the resource optimization phase, with two variants: pipelined circuits without interlock (marked with "-") and interlocked pipelined circuits (marked with "+"). The percentages are compared with the mapping results in Table 3.2. The last column is the total HLS runtime (Phase I + Phase II) compared with the mapping results (Phase I). 60

4.1 Area and wire length comparison of two cases for conventional design and adaptive design 81

4.2 Testcases; total number of gates; number of adaptivity blocks; percentage of gates in adaptivity blocks; area overhead due to adaptivity (%A) and average number of blocks at high VDD (#H) for rule-based and FSM control. 83

1. INTRODUCTION

1.1 Motivation

With the billions of transistors integrated on a chip, large-scale integrated circuits rely heavily on automated synthesis techniques to build robust designs. In recent years, the ICs are often used in mobile devices such as cell phones, tablets, and sensors. These devices are critically sensitive to the power consumption as a result of limited battery capacity. Even for ICs with continuous power supply such as FPGAs, the power density is an increasingly crucial constraint for the number of transistors per unit die area. However, the demands for these high-performance ICs are soaring with the development of applications such as deep learning and high-concurrency computing. As the VLSI technology scaling slows down, the ICs have almost reached their performance ceilings. Design methods for resilient or reprogrammable integrated circuits with better power efficiency are increasingly important to achieve both high performance and low power simultaneously.

Therefore, this research aims to advance the synthesis techniques for high-performance power-efficient integrated circuits including high-level synthesis for approximate circuits and high performance FPGA computing, and control circuits synthesis for adaptive supply voltage (ASV) systems. This chapter will first introduce the current challenges of common synthesis techniques, especially high-level synthesis. Then, it will introduce approximate circuits, FPGAs, adaptive supply voltage systems and how the synthesis techniques can be used to exploit the power-efficiency of these integrated circuits. The contributions of this research can also be summarized as these three synthesis techniques for power-efficiency:

1. Models for high-level synthesis with approximate circuits and precision constraints at the primary outputs (PO). The approximate circuits can reduce the power while cause errors at the primary outputs. Our proposed HLS model is able to reduce the power with bound error at POs.

2. A fast mapping-based high-level synthesis technique targeting high-throughput pipelined

circuits, with a emphasis on transforming the intermediate representation of compilers directly to fully pipelined and parallelized circuits in Verilog HDL format. This technique can substantially increase the synthesis runtime for fully or partially pipelined circuits, which have better power-efficiency.

3. Fine-grained control circuits synthesis for adaptive supply voltage.

1.2 Common Synthesis Techniques

Synthesis in the context of electronic design automation (EDA), usually refers to automatically generating a form of design in a certain level from a specification in another level subject to a set of constraints. Synthesis techniques have been researched and used for decades. Due to the large number of components involved in different levels, such as registers in the register-transfer level and wires in the transistor level, different efficient synthesis techniques has become necessary for modern integrated circuits design. At the point of writing this dissertation, logic synthesis is the best known type of synthesis techniques, which transform the design from register-transfer level (RTL) to gate-level net-list. High-level synthesis (HLS), although it has also been researched for several decades since 1990s with early publications such as [8, 9], is not so widely-adopted as logic synthesis is. The high-level synthesis aims to transform the programs or algorithms written in high-level languages such as C and Haskell to hardware description languages (HDL) in register-transfer level. Partly because the high level languages are not originally designed for use with hardware description, high-level synthesis is rarely used for ASIC design but attracts more attention and adoption for FPGA synthesis. The quality requirement of synthesized RTL designs for FPGA is a little lower since they can be reprogrammed. So, if there were errors or inefficiencies discovered after deployed, they could be fixed later by simply repeating the synthesis process. The cost is much lower compared with the tape-out cost of ASICs. The synthesis process for FPGAs thus might be repeated many times during the design space exploration (DSE), which wants to find the best design parameters for a FPGA design. Since many developers want to use FPGAs as accelerators and program FPGAs like software, the synthesis runtime for FPGAs is more

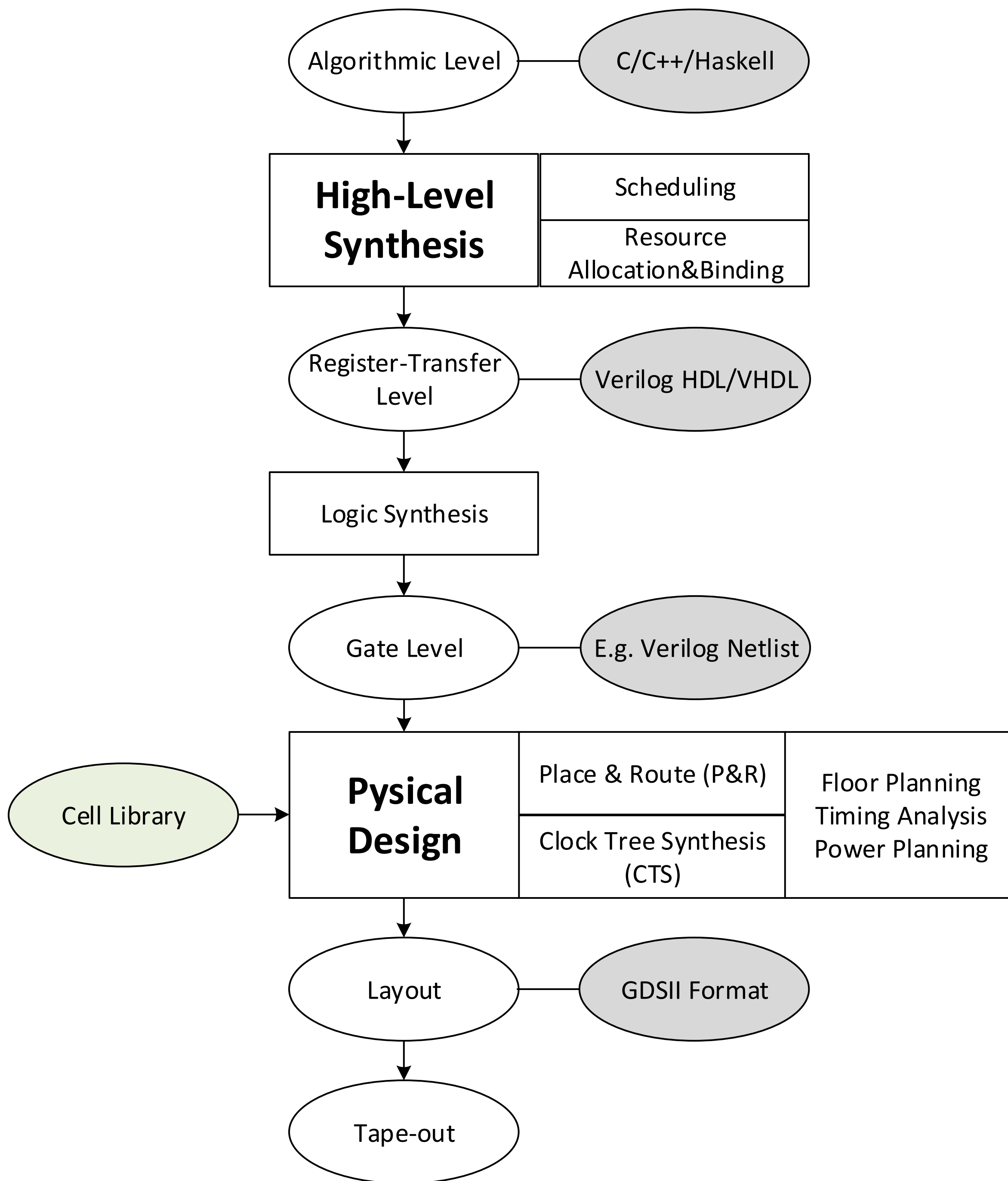


Figure 1.1: Overview of typical IC design flow

important. These challenges and demands emerged in recent years as the general-purpose microprocessors have encounter performance bottlenecks. Alternative computing architectures such as GPU and FPGA are often used to accelerate computing through parallelism. Although FPGAs are more flexible than GPU, the synthesis complexity and runtime became a serious obstacle for FPGAs to be used by developers for a wide range of applications such as deep learning and high-concurrency processing. These applications, if deployed on FPGAs, would be orders of magnitude power-efficient than deployed on servers with microprocessors.

1.3 High-Level Synthesis for Power-Efficiency

High-level synthesis was originally modeled as a optimization problem. The focus of research is on how to achieve optimized RTL design within a set of constraints. After more than two decades, there is still no common input form or specification for HLS as Verilog HDL and VHDL for logic synthesis. Often C or C++ are used as inputs for HLS but some other forms such as Haskell, SystemVerilog and SystemC are also used or researched [10, 11, 12]. Therefore, many research works often start with task graphs extracted from any high-level languages as inputs. The task graph is modeled as a directed acyclic graph (DAG). Each node in the DAG represents an inseparable computing task such as addition and multiplication. Each edge then represents the precedence relationships between these computing tasks. Figure 1.2 is just such a task graph.

Starting from the directed acyclic graph, the HLS can be divided into several steps including scheduling, resource allocation and binding. The scheduling divides the algorithmic level input into several control steps and assign different operations to appropriate control steps. Early studies of the scheduling proposed several simple yet effective algorithms such as as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling algorithms. Figure 1.3 and figure 1.4 show the results of ASAP and ALAP scheduling of task graph in figure 1.2 respectively. More complex list scheduling is also proposed along with ASAP and ALAP [13]. ASAP and ALSP can be seen as special variants of topological ordering.

More advanced but time-consuming models for HLS such as Integer Linear Programming (ILP) are also proposed later. As specified in [8], scheduling in HLS is modeled as a Integer

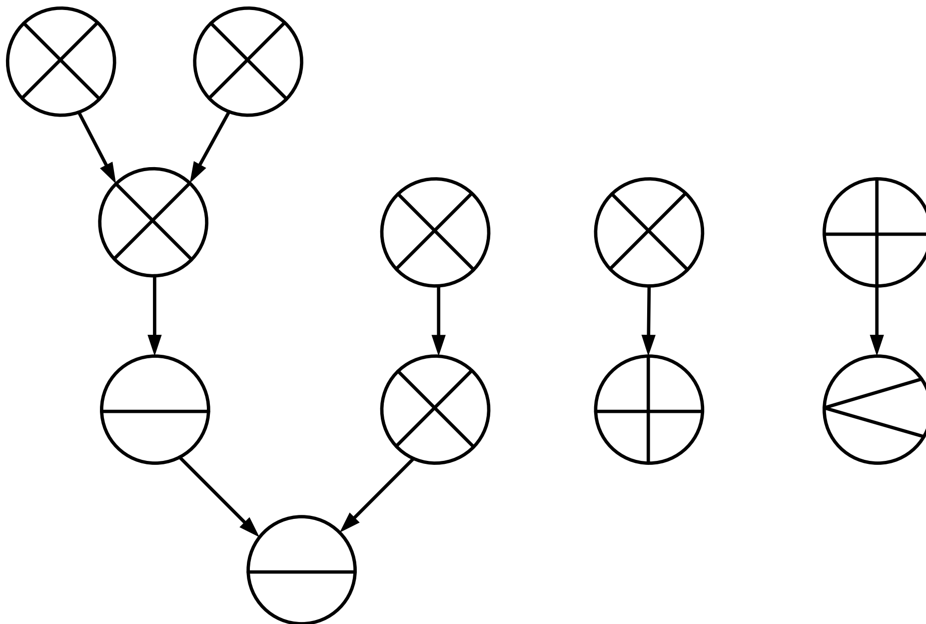


Figure 1.2: Example of task graph for high-level synthesis

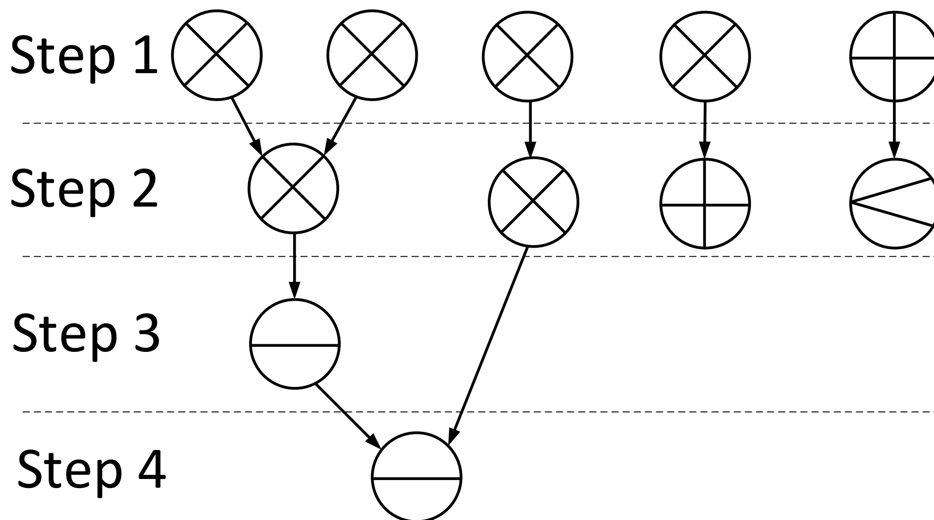


Figure 1.3: The ASAP scheduling result of task graph in figure 1.2

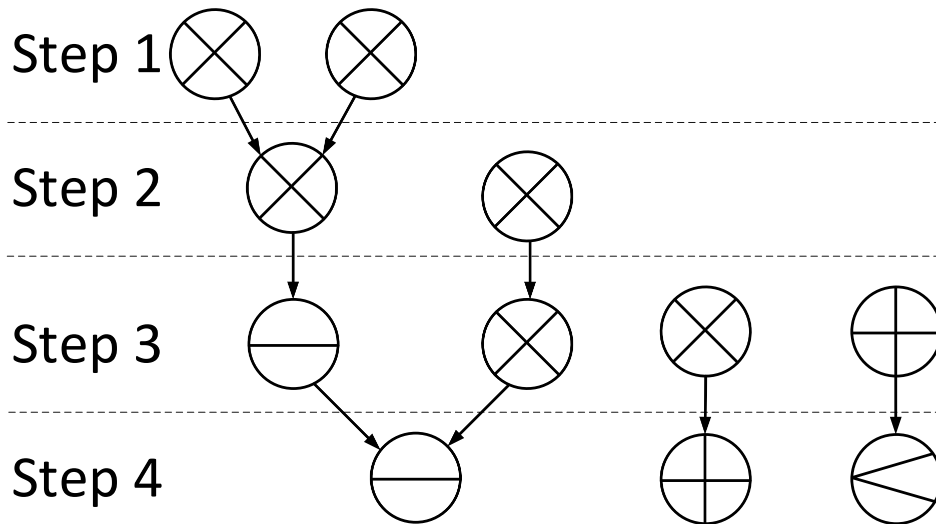


Figure 1.4: The ALAP scheduling result of task graph in figure 1.2

Linear Programming problem with each scheduling option for each task having a corresponding decision variable. Another work in 2006 [14] modeled the scheduling problem as solving a system of difference constraints (SDC), which still requires a linear programming solver to solve.

These works later contributed to more attentions of high-level synthesis, especially for FPGAs. There are some HLS frameworks for research proposed such as SPARK [15] and xPilot [16]. The latter one uses the SDC formulation of scheduling and was incorporated into Vivado Design Suite by Xilinx as Vivado HLS [17]. Open-source HLS frameworks were also proposed such as LegUp [18] and ROCCC [19]. These works all take C/C++ as input high-level language. Except SPARK, xPilot, LegUp and ROCCC 2.0 all use the LLVM compiler infrastructure [20] as front-end. The LLVM compiler framework is able transform the C/C++ inputs to LLVM intermediate representation (LLVM IR). Then, the directed acyclic graph of operation precedence relationships can be extracted from the LLVM IR. The LLVM compiler framework partially standardized the input specification of HLS. After LLVM's release, most newly implemented HLS frameworks use it and transform the high-level languages to LLVM IR first.

1.3.1 HLS for Approximate Circuits

Since the demands for lower-power ICs are increasing rapidly, the approximate circuits were proposed several years ago as a paradigm for improving power-efficiency. Approximate adder and multiplier designs were also proposed such as [3, 21, 22]. Similar to operations with lower precision, these designs are able to reduce the power consumption compared with traditional full-precision circuits, but provide more accuracy than lower-precision designs. Generally, they remove parts of the original full-precision circuits, the result then becomes inaccurate with a possibility of error. With these recent advancements in circuits design, new requirements for high-level synthesis emerge. It is then possible for HLS to reduce the power consumption as much as possible while achieving reasonable precision of the final results.

This research therefore propose an ILP model and a heuristic algorithm to reach this goal [1], by automatically choosing the proper precision option for the implementation of every operations. The ILP model is able to get the optimal HLS results; while the heuristic algorithm delivers suboptimal results with tenth of runtime.

1.3.2 Mapping-Based HLS for Pipelined Circuits

FPGA is known to be more power-efficient than GPU and CPU. For example, before custom ASICs dominated bitcoin mining, FPGAs were used to replace GPUs for a short time, as they typically can achieve slightly worse hash-rates compared with GPUs while only cost no more than half of the power [23]. FPGAs were also used for some machine learning applications [24].

Although FPGAs have these benefits, there are several key challenges for them to be more widely adopted as an option for low-power high-performance applications:

1. The FPGA programming is difficult than the GPU programming. With CUDA (Compute Unified Device Architecture) introduced in 2007 [25] by NVida, the general-purpose programming for GPU is simplified to a degree that no special training in the computer graphics programming or the GPU architecture is required. However, developers still need special knowledge of the hardware design to be able to write the Verilog HDL code or tune the

HLS-generated Verilog code to get better performance. This hinders software developers to program with FPGAs.

2. The HLS softwares are generally orders of magnitude slower than software compilers, which costs much time to do trial and error for debugging and optimization. This partly is because the HLS aims to get optimized design and thus costs much time on optimization with limited knowledge of the lower level hardware requirements. The software compilers, however, is able to perform iterative and local optimizations. For instance, there are three optimization levels for GCC, `O1`, `O2` and `O3`.

3. The existing HLS tools lack support for key features required by some algorithms such as recursion.

4. The existing HLS tools lack support for synthesizing fully pipelined and parallelized circuits, which are the key features that distinguish FPGAs from sequential processing CPUs. Besides, these tools didn't consider specially designed *parallel algorithms*, which have additional design methodologies that are not seen in general algorithm design [26].

Due to these drawbacks of HLS tools, the industry adoptions of FPGAs for high-performance computing acceleration are still in small-scale compared with GPUs. This research proposes a fast mapping-based high-level synthesis technique that targets synthesizing fully pipelined and parallelized circuits with substantially reduced synthesis runtime.

1.4 Adaptive Circuits

Adaptive integrated circuits are another type of technologies that were often used to achieve better power-efficiency. These techniques adjust the operating parameters such as supply voltage and frequency according to the varying conditions the IC chips might encounter, including process variations [27], transistor aging effects [28] and work load variations [29]. Dynamic Voltage Frequency Scaling (DVFS) is one of the most prominent among these adative circuits techniques, which has been deployed in a wide range of CPUs for both computers and mobile devices.

The adaptive circuits often contain some type of sensors to detect potential operating condition variations of the IC chip. One of the first sensor design was proposed in 2003, namely Razor [30], which uses a shadow latch to detect timing errors. These detection results then can be used to adjust the supply voltage and make more aggressive voltage scaling possible.

Another major question for adaptive circuits is how to tune the voltage or frequency of the circuits. The voltage of the circuit can often be tune through body biasing [31] and voltage interpolation [5]. The voltage interpolation provides several supply voltage options and divide the entire circuits into several blocks. Different blocks then are able to choose between these different voltage options, according to the sensor outputs. Recent works also proposed methods for Statistical Static Timing Analysis (SSTA) [32] and gate clustering [33] for adaptive circuits.

1.4.1 Control Circuits Synthesis for Adaptive Supply Voltage

Previous control logic for adaptive voltage are simple. The voltage of entire circuits are adjusted simultaneously. Although this coarse-grained tuning can reduce some unnecessary power consumption, fine-grained tuning, which divide the entire circuits into different clusters and tuning individual cluster independently, provide even more power reduction.

In this research, we use the voltage interpolation to tune the supply voltage, and propose a automatic synthesis method for the fine-graind adaptive supply voltage (ASV) control circuits [2].

1.5 Summary of Contributions

In this research, we explored several synthesis methods and techniques especially the high-level synthesis techniques, which improves the power-efficiency of modern integrated circuits.

1. We propose a error model that can be integrated into the integer linear programming (ILP) model for high-level synthesis. Based on the error model, we also propose two HLS flow for joint high-level synthesis and precision optimization. Experiments are done on task graphs, which can be modeled as directed acyclic graph. We also manually designed several approximate adder and multiplier design to test our HLS flow.

2. In addition to the HLS for approximate circuits, we exploit the parallelism in hardware such

as FPGAs and propose a mapping-based high-level synthesis technique that transforms the static-single assignment (SSA)-form intermediate representation (IR) to Verilog HDL design of fully pipelined circuits. An optional phase of resource optimization is also proposed. It is able to alter the fully pipelined circuit to a partially pipelined circuit or nonpipelined circuit. Pipeline interlocking to address the hazards due to resource conflicts is provided, which achieves better power-efficiency and allows more flexibility of input patterns.

3. We propose a control synthesis algorithm that can automatically generate control circuit block used to adjust the supply voltages of different power domains. An adaptive design flow is also developed to validate our control circuit synthesis algorithm.

2. HIGH-LEVEL SYNTHESIS FOR APPROXIMATE COMPUTING[1]

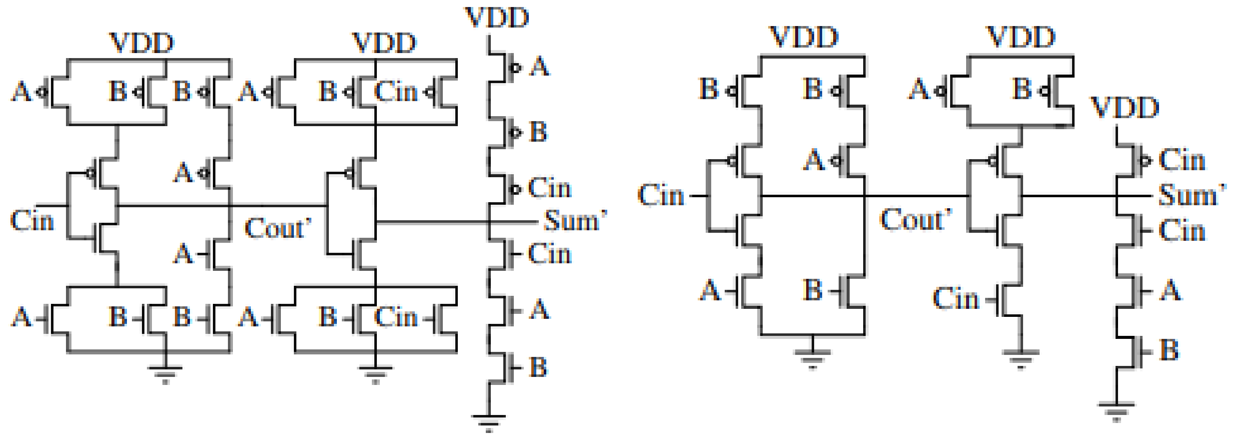
2.1 Introduction

Approximate computing is an emerging research topic for improving power-efficiency [3, 21, 34, 35, 36]. In conventional designs, datapath computations are precise for its bitwidth, i.e., the error is restricted to those less than the least significant bit weight. We term them as *per-bitwidth precise computing*. Approximate computing exploits the intrinsic error tolerance in certain applications such as JPEG compression in digital image processing [37] and weights updating in machine learning [38], and allows occasional small errors beyond the quantization error caused by limited bitwidth. This makes it possible to achieve further power reduction compared with per-bitwidth precise circuits. There has been research works done for the analysis and modeling of approximate computing [39, 40, 41, 36] and the approximate circuits design [3, 21, 42]. These works help implementing power-efficient circuits with approximate computing. One example of approximate adder is shown in figure 2.1, compared with the accurate adder design. The transistor-level Verilog HDL code is also attached in the appendix.

The analysis and modeling of approximate computing are mainly in the logic synthesis level. The increasing complexity of integrated circuits prompts us to explore automated system-level design and high-level synthesis for approximate computing. High level decisions usually generate large impact to the overall system performance and power. Given a rich body of low/circuit level approximation techniques, how to efficiently utilize them at system level is of great importance but not well studied yet. In [39], an automated design space exploration technique is introduced to choose between approximate and per-bitwidth precise implementations for each operation. However, it pays little attention to scheduling and resource allocation/binding algorithms, which are the core techniques in HLS.

This research attempts to find techniques that make efficient use of approximate circuits at

[1]Reprint with permission from "Joint Precision Optimization and High-Level Synthesis for Approximate Computing" by Chaofan Li, Wei Luo, Sachin S. Sapatnekar and Jiang Hu, 2015. *Proceedings of the 52nd Annual Design Automation Conference*, Article 104, Copyright 2015 by the authors.



(a) One accurate adder design

(b) One approximate adder design

Figure 2.1: The comparison of the accurate and the approximate adder [3]

system/block level by considering them in high-level synthesis. While conventional HLS mostly emphasizes performance, power and area, the notion of approximate computing requires error control especially, since the error incurred by approximation cannot be too large. Existing error models for approximate computing [40, 43, 41] are mostly targeted to post-design analysis and very difficult to be used within optimizations. In this research, we first study optimization-friendly error models that can be integrated into the HLS flow. Then, we propose and investigate two HLS approaches that cover scheduling, Functional Unit (FU) allocation/binding together with approximation assignment. We focus on data-intensive applications as opposed to control-intensive applications since approximate computing mostly occurs at datapaths. The two HLS approaches are as follows:

1. An ILP (Integer Linear Programming) formulation for simultaneous precision optimization and high-level synthesis including scheduling and FU allocation and binding.

2. A multiple choice multiple dimension Knapsack formulation is introduced for precision optimization that concurrently considers approximation selection and bitwidth optimization, followed by an iterative list scheduling heuristic is developed to perform simultaneous scheduling, FU allocation and binding with consideration of approximation.

2.1.1 Error Control

The error control in approximate computing is similar to the bitwidth optimization. The difference is that for bitwidth optimization, the inaccuracy or error incurred by truncation or rounding is expected and certain, while the error caused by approximate circuits is unexpected. For a certain computation, the result might be accurate or inaccurate. Research for bitwidth optimization has been performed for at least a decade [44, 45], where bitwidth options of each operation is selected to minimize implementation cost subject to precision constraints. Bitwidth optimization has also been studied together with HLS [46, 47]. The HLS and bitwidth optimization interact with each other but are considered separatedly. In per-bitwidth precise designs, the quantization error analysis must be accurate and thus is too complex to be incorporated within HLS. However, the requirement for accuracy is less strict in approximate computing. Thus, we integrate bitwidth optimization and approximation assignment with HLS such that the solution space is searched more thoroughly. The approximate functional units are considered together with functional units with lower precision.

Error Metric	Definition
Error Rate	$ER = \sum_{X s.t. O_{appx}(X) \neq O_{ref}(X)} Pr(X)$
Error Significance	$ES = \mathrm{E}[O_{appx}(X) - O_{ref}(X)]$
Signal-Noise Ratio	$SNR = \mathrm{E}[\frac{O^2_{ref}(X)}{[O_{appx}(X) - O_{ref}(X)]^2}]$
Average Relative Error Significance	$ARES = \mathrm{E}[\frac{O_{appx}(X) - O_{ref}(X)}{O_{ref}(X)}]$
Mean Squared Error	$MSE = \mathrm{E}[\lvert O_{appx(X) - O_{ref}(X)}\rvert^2]$
Maximum Error	$MAXE = \max[\lvert O_{appx}(X) - O_{ref}(X)\rvert]$

Table 2.1: Existing error metrics

In previous research works, many error metrics have been proposed to measure the inaccuracy caused by either bitwidth optimization or approximation. Table 2.1 summarizes these metrics. In this research, we propose a variance-based error model that is very simple to use in optimizations. It is enhanced by error sensitivity to capture structural correlations in error propagation. Its credibility is validated by Verilog-based Monte Carlo simulations. The proposed techniques are evaluated by simulations on benchmark applications including Verilog simulations. The results confirm the effectiveness of our techniques.

2.2 Preliminaries

The input to the our methods is a task graph or dataflow graph $G(V, E)$ where the node set V is composed by disjoint subsets of primary inputs PI, computation operation nodes $\hat{V}$ and primary outputs PO, and the edges E indicate data dependencies. Since we only consider data paths, there are no loops. This graph can be modeled as Directed Acyclic Graph (DAG). Each node $\omega \in V$ has an operation type τ_ω, such as addition and multiplication. One example of such task graph is shown in figure 1.2.

In approximate computing, a key part is precision control. When there are multiple approximation options, e.g., the approximate adder can be implemented with different numbers of imprecise bits, we need to decide how to choose among different approximate implementations. Further, approximate implementation can be considered together with bitwidth optimization, which decides how many bits are utilized in implementing a computation. The conventional bitwidth optimization consists of two parts: one that decides the data range and the other that affects the computing precision. We focus on the precision part as it coheres better with approximate computing.

For each operation type τ, there is a set of implementations $\mathcal{I}_\tau = \{F_\tau^{\phi_1}, F_\tau^{\phi_2}, ...\}$ of different precisions $\phi_1, \phi_2, ...$ and we use $\mathcal{I}_\tau^{\succeq\phi}$ to represent the subset in $\mathcal{I}_\tau$ that has at least precision ϕ. The precision optimization is to find the lowest precision level $\underline{\phi}(\omega)$ for each $\omega \in V$ such that the system precision specification is satisfied.

The HLS in this work considers FU allocation, binding and scheduling. An Functional Units (FU) is an instance of certain operation implementation. Binding is to associate an operation

$\omega \in V$ with an FU instance f of implementation $F \in \mathcal{I}_{\tau_\omega}$ and we also use the notation $F(\omega)$ and $f(\omega)$ to tell the implementation and FU instance for ω, respectively. We allow operation ω to be bound to an FU of precision higher than $\underline{\phi}(\omega)$ in order to encourage FU sharing, i.e., ω can be bound to any instance of $F \in \mathcal{I}_{\tau_\omega}^{\succeq \underline{\phi}(\omega)}$. Given a latency constraint Q, the scheduling is to find start time $0 \leq s(\omega) < Q - \Lambda_{F(\omega)}, \forall \omega \in V$, where $\Lambda_{F(\omega)}$ is the execution latency for the FU of ω. When we try to bind/schedule an operation but there is no corresponding FU available, a new FU is allocated. Thus, the allocation is decided along with binding and scheduling. The overall objective is to minimize total leakage energy consumption while latency and system precision constraints are satisfied. The extension to include dynamic energy is not difficult. Leakage energy is also highly correlated with FU cost, which is a typical objective function in conventional HLS.

The notations can be summarized in table 2.2.

2.3 Analytic Error Models

If not carefully used, approximation may result in errors that are extravagant and beyond acceptable level. Hence, precision control and error models are of critical importance for approximate computing. An error model should quantify the difference between the precision of a system implementation and the precision specification. If a model is to be employed within optimization algorithms, i.e., to guide each solution search step during optimization, it must be simple to compute as it would be called very frequently. Meanwhile, it must be credible and close to accurate analysis.

The work of [40] attempts to find a couple of error models that do not rely on time consuming simulations. One is based on interval arithmetic, which can be overly pessimistic, and the other is based on affine arithmetic, which has poor storage scalability. Moreover, both techniques entail lookup tables in error propagation, which is restrictive to use in optimizations and is hard to integrate into the HLS integer linear programming (ILP) models. Error rate [41], which is the probability that an approximate result is different from its precise counterpart, is simple to use with ILP by taking logarithm since the logarithm of error rate can be summed together to get the error rate at the primary outputs. The ILP needs only to constrain the error rate at the primary out-

$G(V,E)$	The task graph (modeled as a directed acyclic graph) G with operation set V (modeled as the vertex set of the DAG) and precedence relationship set E (modeled as the edge set of the DAG)
$PO \subseteq V$	The set of operations that are also primary output
$\hat{V} \subseteq V$	The set of operations that are not primary output
$\omega \in V$	A certain operation in the operation set V
τ_ω	The operation type of the operation ω. e.g., addition and multiplication
$\mathcal{I}_\tau$	The set of implementations of the type of operation
$F_\tau^{\phi_1} \in \mathcal{I}_\tau$	A certain implementation for operation type τ with precision ϕ_1
$\mathcal{I}_\tau^{\succeq\phi} \subseteq \mathcal{I}_\tau$	Implementation subset for operation type τ with at least precision ϕ
$\underline{\phi}(\omega)$	The lowest precision level for ω such that the system precision specification is satisfied
$F(\omega)$	The implementation of an operation ω
$f(\omega)$	The FU instance of the implementation of operation ω
Q	The latency constraint
$\Lambda_{F(\omega)}$	The execution latency for the FU implementation $F(\omega)$ for an operation ω
$s(\omega)$	The start time of an operation ω
ϵ	Error of a certain operation or value
$\mu(\epsilon)$	Mean of the error
$\nu(\epsilon)$	Variance of the error

Table 2.2: Notations for HLS with approximate computing

puts. However, it only addresses error frequency without attention to error magnitude. A variance based error model is described in [45]. However, it neglects structural correlations among signal propagations, which can be quite remarkable and it doesn't consider the approximate circuits and HLS.

We now discuss how errors are propagated in a couple of typical operations. Consider addition operation $y = a + b$ and let errors of a, b, the addition and s be denoted by ϵ_a, ϵ_b, ϵ_+ and ϵ_y, respectively. Then, the approximate addition can be represented by

$$y + \epsilon_y = (a + \epsilon_a) + (b + \epsilon_b) + \epsilon_+. \tag{2.1}$$

Likewise, the error propagation in multiplication $p = a \times b$ is described by

$$p + \epsilon_p = a \cdot b + a\epsilon_b + b\epsilon_a + \epsilon_\times \cancel{+\epsilon_a\epsilon_b}. \tag{2.2}$$

For the sake of simplicity without significant loss of accuracy, we neglect the second order error $\epsilon_a\epsilon_b$.

We propose an error model where each error ϵ is treated as a random variable. Then, an error can be characterized by its mean $\mu(\epsilon)$ and variance $\nu(\epsilon)$. We argue that the mean error $\mu(\epsilon)$ at a system/block output under the operations of approximate computing is systematic and can be compensated by a constant offset. Then, the overall computation precision is determined by the variance $\nu(\epsilon)$.

Lemma: *If an error ϵ is a random variable, its variance after the constant compensation is equivalent to the Mean Squared Error (MSE).*

Definitions:

$$\mathrm{MSE}(\epsilon) = \mathrm{E}(\epsilon^2)$$

$$\nu(\epsilon) = \mathrm{E}[(\epsilon - \mu(\epsilon))^2]$$

Proof:

$$\begin{aligned}
\nu(\epsilon) &= \mathrm{E}[(\epsilon - \mu(\epsilon))^2] \\
&= \mathrm{E}[\epsilon^2 - 2\mu(\epsilon)\epsilon + \mu^2(\epsilon)] \\
&= \mathrm{E}\epsilon^2 - 2\mu^2(\epsilon) + \mu^2(\epsilon) \\
&= \mathrm{E}\epsilon^2 - \mu^2(\epsilon) \\
&= \mathrm{MSE}(\epsilon) - \mu^2(\epsilon)
\end{aligned}$$

MSE is a common error model in approximate computing [41] and equivalent to Peak Signal Noise Ratio, which is employed in [3, 34]. According to Equations (2.1) and (2.2), the error of system output is a linear combination of the operation errors. The variance of a linear combination of random variables $X_1, X_2, ..., X_n$ is

$$\nu(\sum_{i=1}^{n} k_i X_i) = \sum_{i=1}^{n} k_i^2 \nu(X_i) + \cancel{2\sum_{i=1}^{n}\sum_{j=i+1}^{n} k_i k_j cov(X_i, X_j)}, \tag{2.3}$$

where k_i denotes constant coefficient and $cov(X_i, X_j)$ is the covariance between X_i and X_j.

The overall system error is a composite effect due to error generation, like the ϵ_+ in Equation (2.1) and $\epsilon_\times$ in Equation (2.2), and error propagation like ϵ_a and ϵ_b in Equation (2.1) and (2.2). The error generations among different operations are largely independent of each other, except a rare case where two operations use the same implementation and the same input data. Therefore, we drop the covariance term in our model. This simplification also helps to avoid nonlinear terms of decision variables in optimization.

Error propagations exhibit strong structural correlations that cannot be ignored. For example, an error from the approximate adder in Figure 2.2 is propagated along two paths and it is subtracted in the upper path. When the two paths reconverge, the error from the lower path is canceled by the one propagated along the upper path. We propose the concept of *error sensitivity* (ES) to capture the first order effect of such structural correlation. For a single error ϵ_ω from operation $\omega \in \hat{V}$

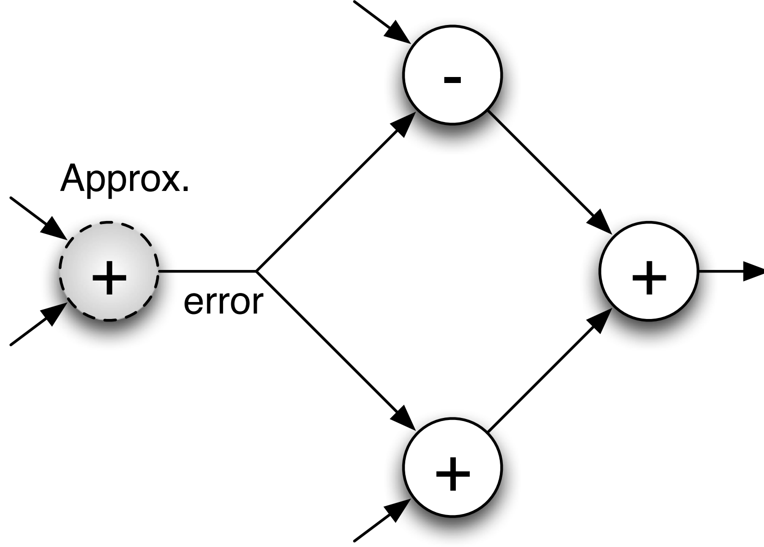


Figure 2.2: An error from an approximate adder may cancel itself along reconvergent paths.

and the error $\epsilon_{\omega,o}$ incurred by ϵ_ω at a primary output $o \in PO$, the error sensitivity is defined as $ES_{\omega,o} = \frac{\epsilon_{\omega,o}}{\epsilon_\omega}$. Then, the error variance of an output $o \in PO$ can be expressed as

$$\nu(\epsilon_o) = \sum_{\forall \omega \in \hat{V}} ES^2_{\omega,o} \cdot \nu(\epsilon_\omega). \tag{2.4}$$

Please note that ES is squared here like the coefficient k in Equation (2.3).

The ES of each node $\omega \in \hat{V}$ can be obtained through an extension to depth first search (DFS) of G. If all operations are addition, $\epsilon_{\omega,o}$ is simply $n \times \epsilon_\omega$, where n is the number of distinct paths from node ω to output o, and thus $ES_{\omega,o}$ is n. If the error ϵ_ω experiences a scaling $\times K$ operation along one of the paths to o, $ES_{\omega,o}$ is $(n-1)+K$. If it is multiplied by another variable, the error propagation is approximated by scaling of an empirical value. Given a task graph, the DFS-based ES estimation is performed once as a pre-processing. The pseudo-code for algorithm is shown in algorithm 1.

```
1  Error_Sensitivity(G(V,E)) begin
2      foreach ω ∈ V, o ∈ PO do
3          ES_{ω,o} ← 0
4      end
5      foreach v ∈ V do
6          DFS(v)
7      end
8  end
9  DFS(v) begin
10     if v is visited then
11         return
12     else if v == o ∈ PO then
13         ES_{v,o} ← 1
14     else
15         foreach edge (v, w) ∈ E do
16             foreach  o ∈ PO do
17                 DFS(w)
18                 if w is scaling ×K then
19                     ES_{v,o} ← ES_{v,o} + ES_{w,o} × K
20                 else if output of v is subtracted in w then
21                     ES_{v,o} ← ES_{v,o} − ES_{w,o}
22                 else if output of v is added in w then
23                     ES_{v,o} ← ES_{v,o} + ES_{w,o}
24                 else
25                     ES_{v,o} ← ES_{w,o}
26                 end
27             end
28         end
29     end
30 end
```

Algorithm 1: Error sensitivity computation.

2.4 Knapsack-Based HLS for Approximate Computing

We describe a sequential heuristic that first decides precisions, considering both approximation selection and bitwidth optimization, and then performs a list scheduling-based HLS algorithm with consideration of approximation.

2.4.1 Knapsack-Based Precision Optimization

In the simplest case, each operation $\omega \in V$ has only one approximate implementation, i.e., $|\mathcal{I}_{\tau_\omega}| = 1$. A binary decision variable $x_\omega \in \{0, 1\}$ tells if to choose approximation for ω or not. If the energy saving for using approximation at ω is Ψ_ω, we wish to maximize the total energy savings subject to error variance constraint at the output. If there is a single output at G, the formulation is

$$\text{maximize} \sum_{\omega \in \hat{V}} \Psi_\omega \cdot x_\omega \tag{2.5}$$

$$\text{s.t.} \sum_{\omega \in V} ES_\omega^2 \cdot \nu(\epsilon_\omega) \cdot x_\omega \leq \nu_B \tag{2.6}$$

where ν_B is the error variance bound at the output. This formulation is the well-known 0-1 knapsack problem. Note that inequality (2.6) has a linear form as a result of removing the covariance term in Equation (2.3).

If there are more than one primary output, for example, k primary outputs, the problem becomes a k-dimensional knapsack problem:

$$\begin{aligned}
\text{maximize} \sum_{\omega \in V} \Psi_\omega \cdot x_\omega & \\
\text{s.t.} \sum_{\omega \in V} ES_{\omega,o}^2 \cdot \nu(\epsilon_\omega) \cdot x_\omega \leq \nu_{B_o} & \qquad \forall o \in PO \\
x_\omega \in \{0, 1\} & \qquad \forall \omega \in V
\end{aligned}$$

When we consider multiple approximation implementations, simultaneous bitwidth optimizations and multiple output nodes, the above formulations can be extended as a Multiple choice Multiple

dimension Knapsack Problem (MMKP). In MMKP, a set of items are partitioned into n classes and k dimensions. One needs to choose exactly one item from each class such that the overall benefit is maximized while the capacity constraint of each dimension is satisfied. To our case, each operation $\omega \in \hat{V}$ is a class, which corresponds to a set $\mathcal{I}_{\tau_\omega}$ of implementations, and each dimension is for one output $o \in PO$ with error variance bound ν_{B_o}. The decision variable becomes $x_{\omega,F} \in \{0,1\}$, which tells if to choose $F \in \mathcal{I}_{\tau_\omega}$ for operation $\omega \in \hat{V}$. The formulation is:

$$\begin{aligned} \text{maximize} \sum_{\omega \in V} \sum_{F \in \mathcal{I}_{\tau_\omega}} & \Psi_{\omega,F} \cdot x_{\omega,F} \\ \text{s.t.} \sum_{\omega \in V} \sum_{F \in \mathcal{I}_{\tau_\omega}} & ES^2_{\omega,o} \cdot \nu(\epsilon_{\omega,F}) \cdot x_{\omega,F} \le \nu_{B_o}, \ \ \forall o \in PO \\ \sum_{F \in \mathcal{I}_{\tau_\omega}} & x_{\omega,F} = 1, \ \ \forall \omega \in V \end{aligned}$$

Please note the set $\mathcal{I}_{\tau_\omega}$ includes implementations of different bitwidths, different approximation and their combinations for operation type τ_ω. As such, bitwidth optimization and approximation selection are carried out in an integrated manner. We solve this MMKP problem using an ILP solver. The result tells the *lowest precision* $\underline{\phi}(\omega)$ for each node $\omega \in V$ such that the overall system precision specification is satisfied. Please note the $\underline{\phi}(\omega)$ is not a commitment but a guidance to the subsequent HLS.

2.4.2 Approximation-Aware HLS

2.4.2.1 Conventional List Scheduling

One popular heuristic for HLS is the list scheduling [46, 9], which has two variants: one is to minimize resource cost subject to latency constraint and the other is to minimize latency subject to resource constraint. We take the former variant as a basis due to its similarity to our problem formulation, and make a remarkable extension to take approximation/precision into account. The list scheduling handles FU allocation/binding as well.

The core part of conventional list scheduling is preceded by ASAP (As Soon As Possible) and ALAP (As Late As Possible) schedulings. For each node $\omega \in \hat{V}$, the lower (upper) bound of its

start time $\underline{t}_\omega$ ($\bar{t}_\omega$) is its ASAP (ALAP) schedule. Then, the range of its schedule is initially $[\underline{t}_\omega, \bar{t}_\omega]$. Next, the core algorithm is a one-pass topological order traversal of the task graph G. During the traversal, a ready list maintains the nodes whose precedent nodes are either in PI or have already been scheduled. Among all nodes in the ready list, the one $\omega \in \hat{V}$ with the minimum $\bar{t}_\omega$ is selected to be scheduled. If an FU f that implements ω has already been allocated and is available in $[\underline{t}_\omega, \bar{t}_\omega]$, then ω is bound to f and scheduled to the earliest available time of f in $[\underline{t}_\omega, \bar{t}_\omega]$. If no such FU can be found, a new FU is allocated and bound to ω. This procedure is repeated till all nodes are scheduled.

When approximation/precision is considered, a binding has multiple implementation options of asymmetric compatibility. That is, a low precision operation can be bound to a high precision FU of the same type, but not vice versa. This creates a subtlety that makes the conventional list scheduling inefficient. Consider the example in Figure 2.3 where an adder delay is 1 and a multiplier delay is 2. In conventional list scheduling, the approximate multiplication is first bound to an approximate multiplier and later a precise multiplier must be allocated and bound to the precise multiplication, as shown in Figure 2.3 (b). However, the two multiplications can share a single precise multiplier as in Figure 2.3 (c).

Even if approximation is not considered, the conventional list scheduling may be inefficient due to its myopic nature. This is illustrated by the example in Figure 2.4, where the latency deadline is 3 and an adder delay is 1. In the conventional scheduling, when node B is considered for scheduling, it can be bound to $Adder1$ without violating latency constraint. Then, node C can no longer use $Adder1$ due to the latency constraint and $Adder2$ is allocated. In the last time step, all of nodes D, E and F must be scheduled to satisfy the latency constraint. Overall, three adders are allocated. However, one can see that actually only two adders are necessary, as shown in Figure 2.4 (c).

2.4.2.2 Iterative List Scheduling

In order to solve the aforementioned problems, we propose two significant changes to the list scheduling and show the overall pseudo code in Algorithm 2. We first change the algorithm to be iterative instead of one-pass graph traversal. The iterations are shown as the **while** loop in step 7,

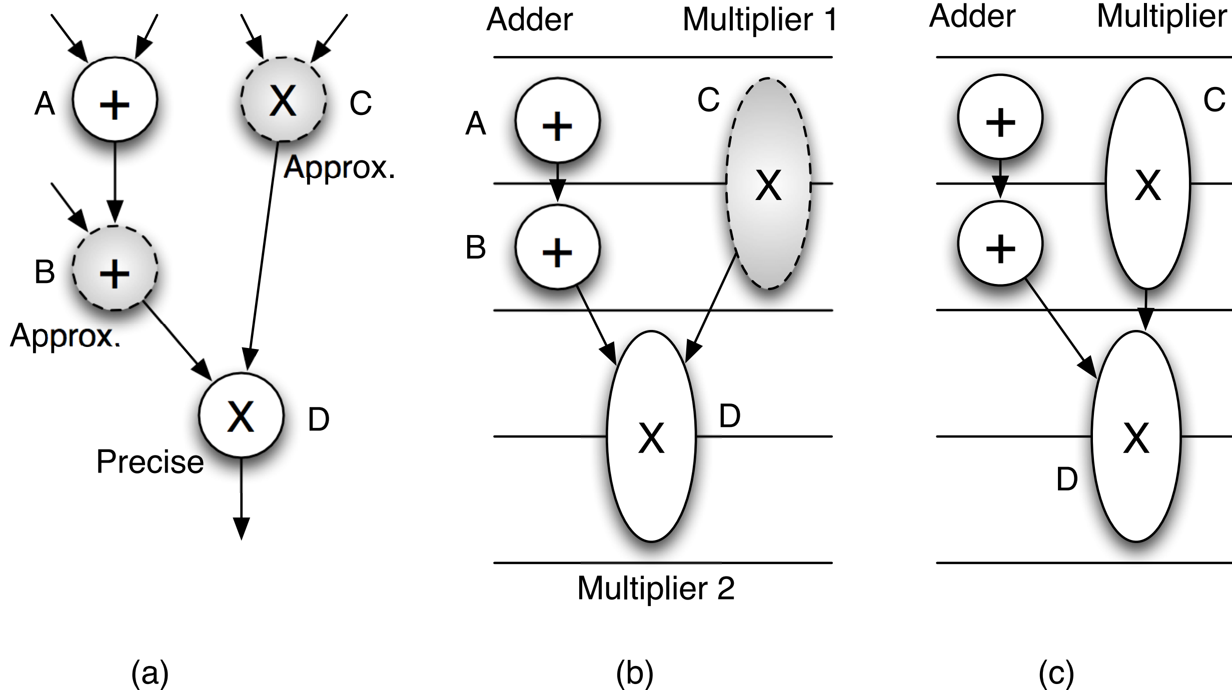


Figure 2.3: (a)Task graph; (b) Conventional list scheduling results in two multipliers; (c) The two multiplications can share one multiplier.

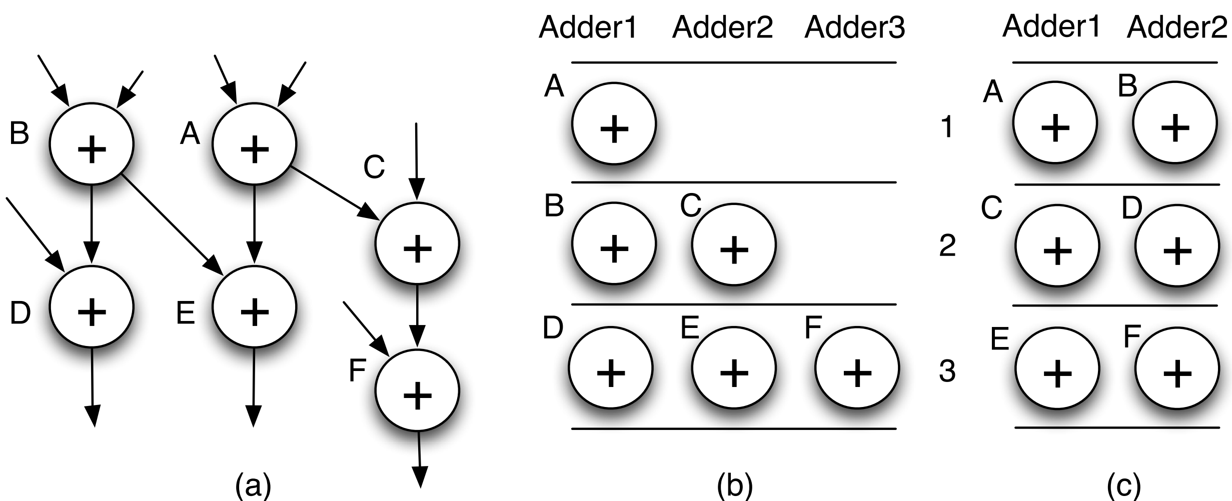


Figure 2.4: (a)Task graph; (b) Conventional list scheduling results in 3 adders for latency constraint of 3; (c) Two adders are sufficient.

where $\mathcal{F}^i$ indicates the set of FUs allocated in iteration i. The first iteration generates the initial FU allocation and the subsequent iterations try to reduce the FU allocation.

1 Approximation_Aware_HLS(G(V,E)) **begin**
2   **foreach** $\omega \in \hat{V}$ **do**
3     $\underline{t}_\omega \leftarrow ASAP_schedule, \bar{t}_\omega \leftarrow ALAP_schedule$
4     $\mathcal{F}^0_{\tau_\omega} \leftarrow \emptyset$ // Initialize allocated FUs for τ_ω
5   **end**
6   $i = 0$
7   **while** $i \leq 1$ *or* $|\mathcal{F}^i| < |\mathcal{F}^{i-1}|$ **do**
8     $i++$
9   **foreach** $\omega \in \hat{V}$ **do**
10       $\mathcal{F}^i_{\tau_\omega} \leftarrow \emptyset$
11   **end**
12   Initialize $Ready_List$ from PI
13   **while** $Ready_List \neq \emptyset$ **do**
14       $\omega \leftarrow$ node with min $\bar{t}$ in $Ready_List$
15       $Candidates^{i-1} \leftarrow f \in \mathcal{F}^{\succeq underline\phi(\omega),i-1}_{\tau_\omega}$ & free in $[\underline{t}_\omega, \bar{t}_\omega]$
16       $Candidates^i \leftarrow f \in \mathcal{F}^{\succeq \underline{\phi}(\omega),i}_{\tau_\omega} And$ free in $[\underline{t}_\omega, \bar{t}_\omega]$
17   **if** $Candidates^{i-1} \neq \emptyset$ *or* $Candidates^i \neq \emptyset$ **then**
18   **if** $\Upsilon^{Candidates^{i-1}}_{\prec\omega} < \gamma \cdot Upsilon^{Candidates^{i-1}}$ **then**
19   Find $\hat{f} \in Candidates^{i-1}$ with min start time
20   break tie with max Υ
21   **else**
22   Find $\hat{f} \in Candidates^i$ with max Υ
23   break tie with min start time
24   **end**
25   **else**
26   Allocate $\hat{f}$ of $F^{\phi(\omega)}_{\tau_\omega}$
27   **end**
28       $\mathcal{F}^i_{\tau_\omega} \leftarrow \mathcal{F}^i_{\tau_\omega} \cup \{\hat{f}\}, \quad f(\omega) \leftarrow \hat{f}$ // binding
29       $s(\omega) \leftarrow$ earliest available time for $\hat{f}$ in $[\underline{t}_\omega, \bar{t}_\omega]$
30   Update $Ready_List$, update $[\underline{t}_\omega, \bar{t}_\omega], \forall \omega \in \hat{V}$
31   **end**
32   **end**
33 **end**

Algorithm 2: Algorithm for approximation aware HLS.

The second and more important change is step 15-24. In step 15 and 16, a set of candidate FUs

are identified from the FUs $\mathcal{F}_{\tau_\omega}^{\succeq\phi(\omega)}$ that are allocated in current and the previous iterations. The "free in $[\underline{t}_\omega, \bar{t}_\omega]$" means that the FU is available from a time in $[\underline{t}_\omega, \bar{t}_\omega]$ to an extent long enough to accommodate one complete operation on the FU. The selection of FUs among the candidates is based on two factors. (i) Early start time: if an operation is scheduled to start early, greater slack (or mobility) is left to subsequent nodes, which consequently have greater chance to share FUs and reduce the number of FUs. (ii) Utilization: if we move operations from FUs with low utilization to those with high utilization, there would be greater chance to empty some FUs.

We define *utilization* $\Upsilon^{\mathcal{F}}$ of a set $\mathcal{F}$ of FUs as the ratio of the total time the FUs are used versus the total latency multiplied by the number FUs in the set. For example, the utilization of $Adder3$ in Figure 2.4 (b) is $\frac{1}{3}$ and the utilization for all three adders is $\frac{6}{3\times3}$. We also define *ancestor utilization* $\Upsilon_{\prec\omega}^{\mathcal{F}}$ with respect to operation $\omega \in \hat{V}$ as the ratio of the total time the FUs in $\mathcal{F}$ being used by ancestor nodes of ω versus the wall-to-wall time of these uses multiplied by the number of FUs in $\mathcal{F}$.

Ideally, we wish all FUs are fully and equally loaded by operations. In other words, at any specific time step, the FU utilization is preferred to be equal to the overall utilization among all FUs. In step 18, we choose the utilization for all candidates $\Upsilon^{Candidates^{i-1}}$ scaled by a correction factor γ as the target. If the utilization of candidate FUs by ancestor nodes of ω is less than the target, the utilization so far is relatively low and we prefer to schedule ω as early as possible. Otherwise (step 22), we only use FUs allocated in current iteration and prefers to squeeze the operation into the FU with high utilization. Steps 20 and 23 tell how break tie.

Now let us see how our algorithm solves the case of Figure 2.3. After the first iteration, as in Figure 2.3 (b), one approximate multiplier and one precise multiplier are allocated. In the second iteration, when node C is considered, both multipliers are candidates and the corresponding $\Upsilon^{Candidates^1}$ is $\frac{1}{2}$. As there is no node preceding C, we go to branch of step 19. Since both multipliers can be scheduled to time step 1, we break tie according to utilization. Excluding operation C, the utilization of the approximate multiplier is 0 while the utilization of the precise multiplier is $\frac{1}{2}$. Thus, node C is bound to the precise multiplier and the approximate multiplier is no longer

used in the second iteration.

For the example in Figure 2.4, we reach (b) after the first iteration. When we try to schedule node B in the second iteration, the $\Upsilon_{\prec B}^{Candidates^1}$ means the utilization of $Adder1$ in time step 1, which is $\frac{1}{3}$. This is less than the overall utilization of $\frac{2}{3}$ and therefore we go to step 19 to bind B with $Adder2$ and start B at time step 1. Eventually, the second iteration would reach a result like Figure 2.4 (c).

2.5 ILP-Based HLS for Approximate Computing

Integer Linear Programming is an early high-level synthesis formulation proposed in [8]. To integrate the precision optimization with the ILP model for high-level synthesis, we simply add error constraints for every primary outputs. Since the error sensitivities can be accumulated together at the primary outpus, there is no need to constrain every internal operations. Then, The precision optimization, scheduling, FU allocation and binding can be performed simultaneously through ILP:

$$\text{Min} \quad \sum_F L_F \cdot u_F \cdot Q \tag{2.7}$$

$$\text{s.t.} \quad \sum_{0 \le t < Q} \sum_F x_{\omega,t,F} = 1,\ \forall \omega \in V \tag{2.8}$$

$$\sum_t \sum_F x_{\omega,t,F} \cdot t - s(\omega) = 0,\ \forall \omega \in V \tag{2.9}$$

$$s(v) + \sum_t \sum_F x_{v,t,F} \cdot \Lambda_F \le s(\omega),\ (v,\omega) \in E \tag{2.10}$$

$$s(o) \le Q,\ \forall o \in PO \tag{2.11}$$

$$\sum_{t | t \le \hat{t} < t + \Lambda_F} \sum_{\omega \in V} x_{\omega,t,F} \le u_F \le U_F, 0 \le \hat{t} < Q, \forall F \tag{2.12}$$

$$\sum_{\omega \in V} \sum_t \sum_F ES^2_{\omega,o} \nu(\epsilon_F) x_{\omega,t,F} \le \nu_{B_o}, \forall o \in PO \tag{2.13}$$

The latency constraint is denoted as Q and thus all operations must be executed in the time range $0 \le t < Q$. Decision variable $x_{\omega,t,F} \in \{0,1\}$ tells if operation $\omega \in \hat{V}$ is scheduled to start at time t and bound to an FU of implementation F. The leakage power and latency of implementation F are represented by L_F and Λ_F, respectively. The number of FUs of implementation F being allocated

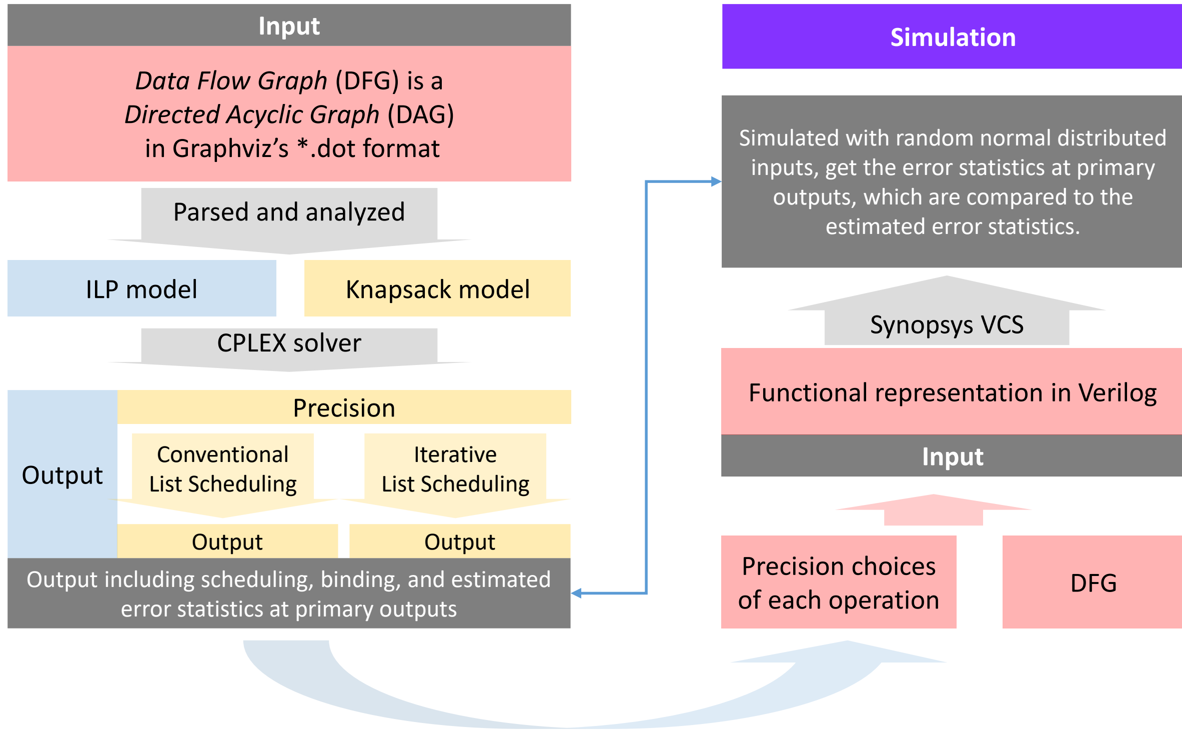


Figure 2.5: Overview of the experiment flow

is u_F and U_F is a constant upper bound for u_F. The start time of operation ω is denoted by $s(\omega)$.

The objective here is to minimize the total leakage energy. Constraint (2.8) ensures that each operation is scheduled to only one time and bound to only one FU. Inequality (2.10) is the precedence constraint and inequality (2.11) is the latency constraints at PO. The FU allocation is realized through inequality (2.12). The last constraint is to bound variance at each PO node.

2.6 Experiment

We implemented gate level designs of approximate adder [3] and approximate multiplier [21], along with 24-bit, 28-bit and 32-bit precise adder/multipliers using $15nm$ technology [48]. The full adder Verilog HDL design of approximate adder in [3] is included in appendix 1. Energy and latency are characterized through SPICE simulations while error variance is obtained through Verilog-based Monte Carlo simulations using Synopsys VCS [49]. The experiments are performed on a set of MediaBench applications [50] and some other common applications like FIR, IIR and

ARF, each of which has about 10 to 100 nodes. The ILP and Knapsack problems are solved by the CPLEX Optimizer [51]. Since there is no previous work on scheduling/binding considering approximate computing, we compare the following methods:

- All-Prcs: all FUs take the most precise implementation upon the conventional list scheduling result.
- All-Apprx: all FUs choose an approximate implementation upon the conventional list scheduling result.
- K-LS: our Knapsack based precision optimization followed by conventional list scheduling.
- KILS: our Knapsack based precision optimization followed by our iterative list scheduling.
- ILP: our integer linear programming approach.

An overview of the experiment setup can be seen in figure 2.5.

The main results are shown in figure 2.6 and figure 4.10b, where 2.6 is for energy and 4.10b is for error standard deviation, which is equivalent to variance. Unlike variance that is in the dimension of square of data, standard deviation has the same dimension as data and provides more intuitive sense. The rightmost clusters of bars are the average results. All results satisfy latency constraints. The energy savings from K-LS, which is the conventional list scheduling, is about 11% while our ILP can achieve average energy reduction of 40%. Our KILS heuristic also outperforms the conventional approach by reducing 16% energy. ILP often results in less energy than All-Apprx as it uses less number of FUs. Figure 4.10b shows that all methods except All-Apprx satisfy the error constraint. Without error control, the All-Apprx method causes standard deviation about $5\times$ of the constraints, although it provides 20% energy savings.

In order to confirm the credibility of our variance-based model, we implement the ILP results in Verilog for two cases - ARF (Auto Regression Filter) and FIR filter. We run Verilog-based 20K-run Monte Caro simulations to obtain MSE at the outputs, which are compensated by constant offsets to nullify the mean errors. We observe that there is about 10% difference between our variance

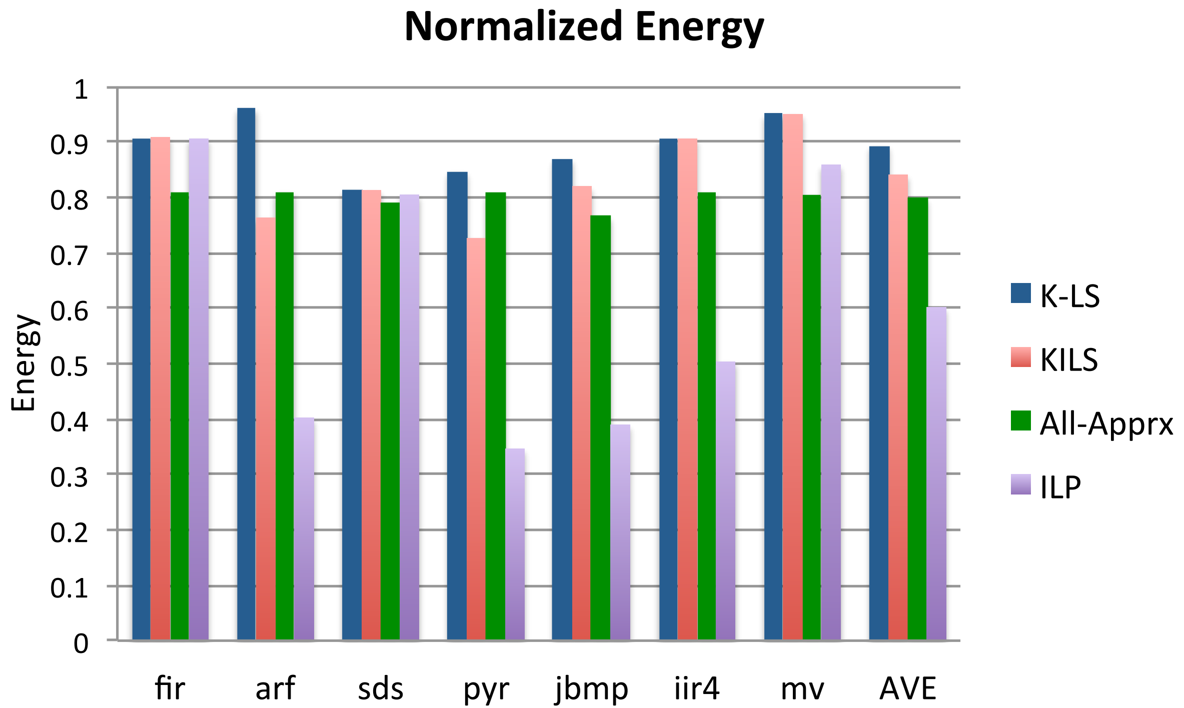


Figure 2.6: Energy normalized with respect to All-Prcs results.

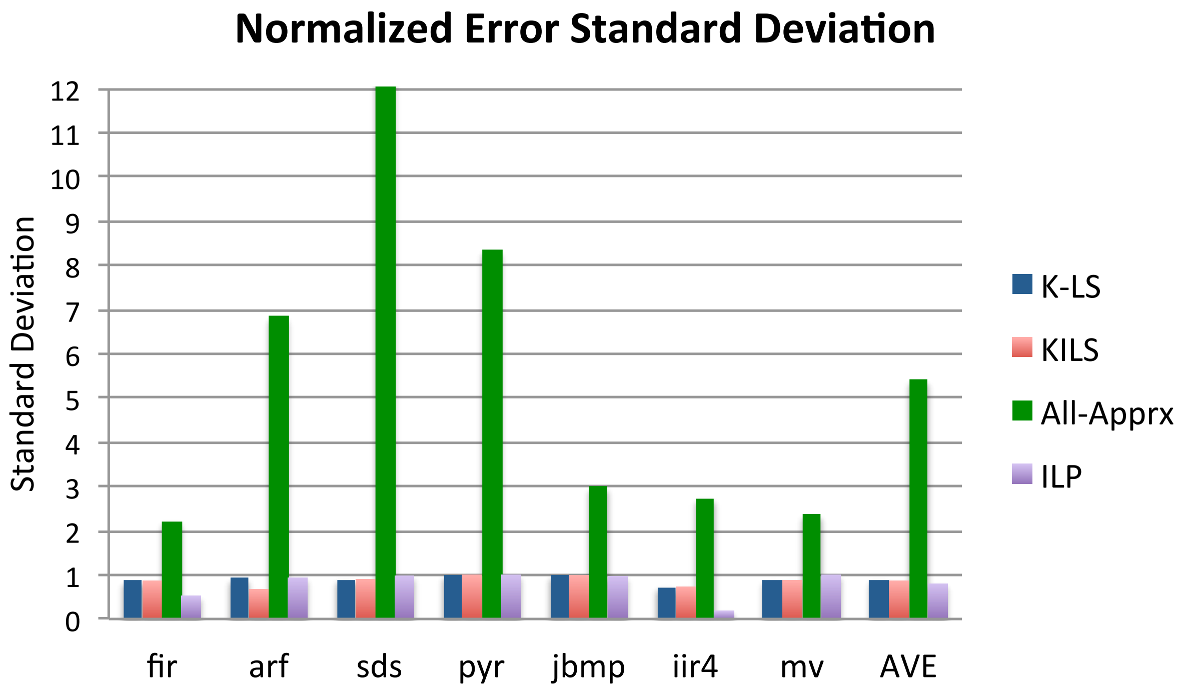


Figure 2.7: Error standard deviation normalized with respect to standard deviation constraints.

and the simulated MSE. However, the *correlation coeffcient* between them is 0.94, which means they are highly correlated. We further plot the energy-error tradeoff curves using the two different error models in Figure 2.8. For each of FIR and AFR cases, the curves from the two models are almost identical. This reminds the Elmore delay model, which is inaccurate but has high fidelity and provides reliable guidance in optimizations. Figure 2.8 also confirms that our approach can realize different energy-error tradeoffs.

Testcases	K-LS			KILS			ILP		
	Estm ν	Siml ν	Diff	Estm ν	Siml ν	Diff	Estm ν	Siml ν	Diff
FIR	28792	24826	16%	28792	24826	16%	28792	23285	24%
ARF	24717	24344	1.5%	1438	1270	13%	29153	21011	39%

Table 2.3: Verilog simulation results.

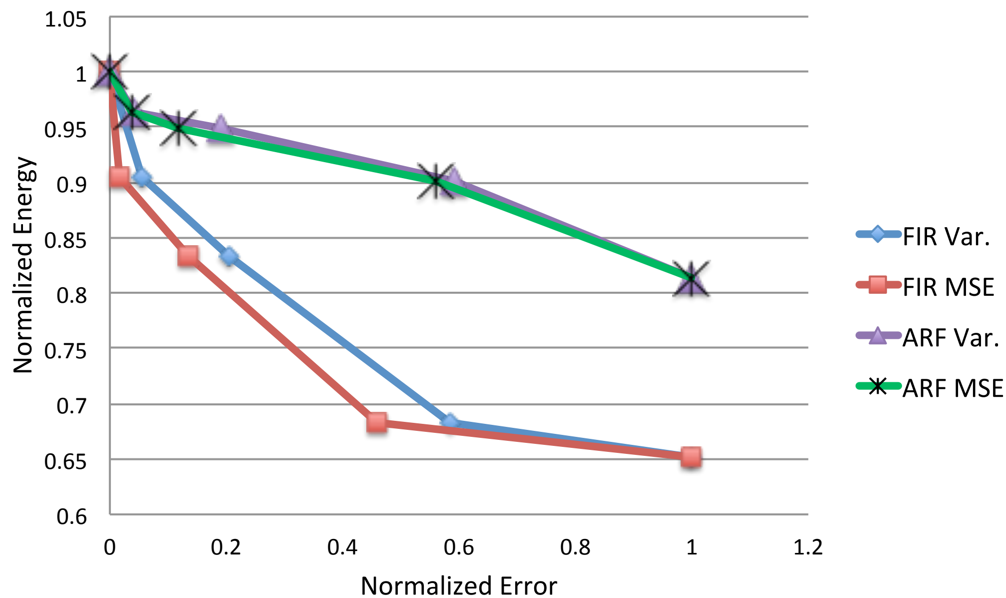


Figure 2.8: Energy-error tradeoff from ILP result. The MSE results are from Verilog simulations.

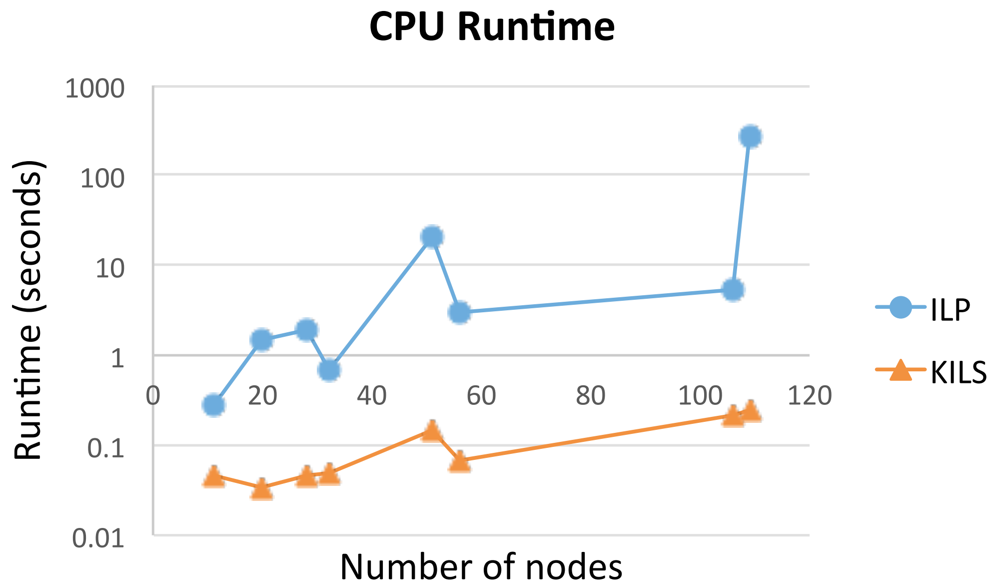


Figure 2.9: Runtime (in log scale) comparison.

We compare runtime of KILS and ILP in Figure 2.9. One can see that KILS is usually one order of magnitude faster than ILP. Moreover, KILS has better scalability. KILS has an advantage in handling large cases, although its solutions are not as good as those from ILP.

2.7 Conclusion

In this work, we propose a simple yet credible error model. We develop a sequential heuristic and an ILP based exact approach for joint precision optimization and approximation-aware HLS. The approaches provide energy vs. error tradeoff at system level.

3. MAPPING-BASED HIGH-LEVEL SYNTHESIS FOR PIPELINED CIRCUITS

3.1 Introduction

High-level synthesis (HLS), which automatically synthesizes designs from high-level languages to implementations in register-transfer level (RTL), has been increasingly adopted by designers, especially for FPGA synthesis. Since the FPGA has become a popular platform for high performance computing, such as that in FPGA-based accelerators [52], HLS plays a central role in bridging the gap between algorithms written in high-level languages, such as C/C++ [18, 53, 54, 15] and Haskell [12, 55, 56], and RTL designs specified in hardware description languages (e.g. Verilog HDL and VHDL). The C/C++-based HLS usually spends a lot of effort on resource-constrained optimizations and loop transformations. The loop transformation may include both simple loop unrolling, also generally supported by software compilers, and complex polyhedral analysis [57]. The HLS with Haskell, quite different from the C/C++-based HLS, pays little attention to loops due to the functional programming nature of Haskell. Instead, recursion is often preferred [12]. The functional programming patterns are also exploited to better support parallel computing [58]. The C/C++-based HLS has more commercial frameworks, such as Vivado HLS [53], LegUp [18] (Open-source till version 4.0), Synopsys Synphony [59] and Cadence Stratus [60]. While, the only commercial HLS tool with Haskell, to the best of our knowledge, is Bluespec [55], based on an extension of Haskell called Bluespec SystemVerilog (BSV). Our work is C-based HLS and thus it is mostly compared with C/C++-based previous works. Some features in Haskell-based HLS are also inspiring such as focusing on the controllability [61] of HLS and support for recursion [12].

Despite the tremendous progress on HLS technologies, C/C++-based HLS still requires complex configuration of constraints and pragma/directive insertions in the high-level language source code. For designers, it is very hard to control the architecture and predict the performance and cost of synthesized RTL designs according to these parameters of constraints and pragmas before a complete run of HLS. In practice, it is very rare to start with a single run of HLS and then continue

to logic and physical synthesis. Instead, HLS is more often employed as a solution evaluation kernel frequently called in design space exploration (DSE), which searches for parameters that lead to the optimized designs [62, 63, 64, 65, 66, 67]. Although HLS is much faster than logic and physical synthesis, it is called so often such that its runtime is still a major bottleneck for DSE. By attempting many possible combinations of constraints and pragma parameters, a DSE can easily run for hours to days. The nontrivial HLS runtime is mainly caused by the optimization process in most C/C++-based HLS frameworks, which usually calls a mathematical optimization solver to solve the models including one or more of integer linear programming (ILP) [8] (which we also used in chapter 2), linear programming (LP) [14] and boolean satisfiability problem (SAT) [68]. These solvers are also hard to control the results and to perform incremental optimizations, since the detailed process of optimization is hidden in the solving process.

In this work, we propose a fast mapping-based HLS technique that is friendly to local incremental design and with particular support to pipelined circuit synthesis and parallel processing. Instead of constrained optimization via optimization engine such as ILP and LP, our approach first directly maps a static-single assignment (SSA)-form intermediate representation (IR) onto a fully pipelined circuit with temporary relaxation of resource constraints. To facilitate the fast mapping, we propose a new datapath control synthesis leveraging the Φ function used in SSA IR instead of the conventional finite-state machine based approach. If there is resource constraint violation after the mapping, resource optimizations are performed to achieve resource sharing in an iterative manner. The iterations can proceed till the circuit is only partially pipelined or even without pipelining, depending on design needs. The resource relaxation and Φ function based data path control help fast HLS, while the iterative resource optimization allows local incremental modifications.

Resource sharing in a pipelined circuit may lead to structural hazards. In conventional HLS, this problem is solved by regulating the input data patterns according to interval pragmas in source code and different pragmas may result in quite different levels of resource sharing. We solve this problem through a new approach of automatic interlock synthesis, which can relax the requirement of regulated input patterns and fits well with our iterative resource optimization procedure. As such,

we can achieve explicit control on resource sharing as opposed to the conventional trial pragma approach. This controllability can further reduce the trial HLS runs in design space exploration. To the best of our knowledge, this is the first work on automatic pipeline interlock synthesis together with HLS. Perhaps the only remotely related work is [69], where local clock gating for interlocked pipelines is studied and the large wire delay of stall signals is addressed by two-phase transparent latches or master-slave flip-flops.

The contributions of this work are summarized as follows.

(1) We propose a fast mapping-based high-level synthesis technique, which is an order of magnitude faster than a state-of-the-art commercial HLS tool.

(2) A new dataflow control synthesis approach based on the Φ function is developed. It plays an important role in the mapping-based HLS.

(3) Hardware oriented array SSA exploitation is first studied in this work, to the best of our knowledge. It helps increase the throughput of synthesized circuits.

(4) An iterative resource optimization method is developed. It supports interlocked pipeline synthesis, which does not require data input regulation and helps reduce trial runs of HLS.

(5) The proposed HLS can handle structural recursion, which is a feature not well studied before.

3.2 Backgrounds

In this section, we first introduce two important techniques that are important to the mapping-based HLS: distributed memories in hardware such as FPGAs and the static-single assignment form. The distributed memories in FPGAs make it more efficient to implement pipelined circuits first and alter it to partially pipelined or nonpipelined circuits later; the SSA-form provides a good form of description of the RTL designs to synthesize with our mapping-based HLS.

3.2.1 Distributed Memories

Compared with traditional architectures used in microprocessors, hardware such as FPGAs provide much more distributed memory resources. There are mainly two types of distributed

memories in FPGAs: the LUT RAMs and registers, both of which are located within the logic blocks.

Look-up tables (LUT) are the main programmable logic resources on FPGAs. Since LUTs are programmable, they can also be configured as memories. For instance, on Xilinx's FPGAs the LUTs can be configured as ROM, RAM or even RAM-based shift registers, which is commonly called distributed RAM [70]. This LUT memory implemented as shift registers is especially useful to support pipelining delay compensations as shown in figure 3.1 and [4].

The logic block, which consists of several slices or cells, provides the main logic resources on FPGAs. For FPGAs these logic resources also need to be reconfigurable, and thus they must have some memories to store the logic information, which is just the look-up tables (LUTs). Therefore, the main logic resources on FPGAs are intrinsically programmable memories. Naturally, the LUTs might be able to perform as memories.

Also, the logic blocks often contains many flip-flops, which can be used to implement discrete registers, and thus can be seen as another type of distributed memories. As an example, table 3.1 summarizes the characteristics of the major three types of memories on Xilinx Virtex 7 series FPGAs.

	Block RAM	Distributed RAM	Flip-flops
Size	28.8–64.8 Mb	6.912–21.016 Mb	0.408–2.4 Mb
Width	57.6–129.6 Kb	162–516 Kb	0.408–2.4 Mb
Depth	512	36	1

Table 3.1: Storage options on Xilinx Virtex 7 series FPGAs: depth and width is when the RAM is configured with maximal bandwidth.

Although the total capacity of distributed memories are not negligible, recent memory allocation schemes for high-level synthesis often focus on the on-chip block RAM such as the memory partitioning [71, 72, 73]. and multi-ported RAM [74]. Due to the limited reading/writing ports in block RAMs of FPGAs, memory partitioning attempts to partition the data into several parts

and map them to different block RAM banks. Since each bank has its independent reading/writing ports, these additional ports help minimize the memory access conflicts if the memory can be carefully partitioned in accordance with the memory access patterns. Another work about multi-ported RAM also focused on block RAMs [74].

More comprehensive memory management methods with a compiler approach were explored in [75], which considers both the block RAMs and discrete registers together with compiler approaches of *data distribution*, *data replication* and *scalar replacement*. Data distribution is similar to memory partitioning, which partitions an array's data into disjoint data sets, and maps them to different single-ported memories; data replication creates copies of the data mapped to distinct memory blocks, which [74] also considers to support multi-read RAM; Scalar replacement or register promotion maps array references to discrete registers that can be accessed concurrently, as if the array elements are scalar variables.

In our HLS, we promotes the memory references to registers whenever possible. The register promotion for scalar variables stored in the stack is well supported by existing compiler frameworks, and we also implement the register promotion for array references to better support parallel access of array data.

3.2.2 SSA Form

SSA (Static Single Assignment) [76] is a form of intermediate representation (IR) used in compilers for helping code optimizations. A variable in SSA is assigned value exactly once. If a variable needs to be assigned value more than once in a program, a new renamed variable is created corresponding to each value assignment. In SSA form, expressions like $i = i + 1$ would virtually become $i2 = i1 + 1$. Then, variable i might cause the use of two different registers corresponding to $i1$ and $i2$, respectively. As a small yet complete example, code 3.1 can be transformed to LLVM IR, a SSA-form-based IR used in the LLVM compiler framework [77], in code 3.2.

```
1 int br(int a, int b){
2   a = a + b ;
3   if( a > 0 ) b = b + 1 ;
```

```
4   else b = b - 1 ;
5   return b ;
6 }
```

Listing 3.1: Simple C program with a branch.

The labels such as "if.then:" divide a function in the IR into several **basic blocks**. All instructions in LLVM IR lie in a certain basic block. The label marks the starting point of a basic block, which is terminated by terminating instructions such as br and ret. For example, lines 9-11 in code 3.2 form a basic block. Each function has an entry basic block, which has no preceding basic blocks. Other basic blocks always have at least one preceding basic block. The br instruction has labels as arguments, which determine the succeeding basic blocks to enter. All these basic blocks within a function and their preceding/succeeding relationships constitute a directed graph.

Since each original variable has different names for every different values it may have during its lifetime, it is mapped to different registers for newly assigned values. These registers can be naturally used as pipeline registers that divide a computing flow into multiple pipeline stages.

One key problem for the SSA transformation is that when two branches merge afterwards, the program may need to select which renamed variable (or register) from the two branches to use, since they may belong to the same variable in the original program. The function that makes the selection is traditionally called phi or Φ function in literatures related to SSA. The work of [78] proposed an efficient algorithm to decide where to insert Φ functions.

```
1 define i32 @br(i32 %a, i32 %b) #0 {
2 entry:
3  %add = add nsw i32 %a, %b
4  %cmp = icmp sgt i32 %add, 0
5  br i1 %cmp, label %if.then,label %if.else
6 if.then:
7  %add1 = add nsw i32 %b, 1
8  br label %if.end
```

```
9  if.else:
10   %sub = sub nsw i32 %b, 1
11   br label %if.end
12 if.end:
13   %b.addr.0 = phi i32 [ %add1, %if.then ],
14                       [ %sub, %if.else ]
15   ret i32 %b.addr.0
16 }
```

Listing 3.2: Code 3.1 compiled to LLVM IR by `clang -emit-llvm` and optimized by LLVM optimizer `opt` with `-mem2reg`, memory to register promotion which promotes scalar variables from the stack in memory to registers, removes LLVM instructions such as `load`, `store`, `alloca`, and adds appropriate `phi` (Φ) functions.

The SSA form was originally proposed for scalar variables. In [79], software compiling techniques were introduced for handling array SSA, which is much more complicated than scalar SSA. Although SSA form is often employed in C/C++-based HLS, the existing approaches are mostly for software level code optimizations with scalar SSA [80]. To the best of our knowledge, there has been no study on hardware oriented use of array SSA yet.

3.3 Phase I: Mapping

Before HLS, the input C language code is first fed to the "clang" compiler that performs optimizations, such as register promotion, loop unrolling and dead code elimination, and generates LLVM IR. Then, our HLS starts with the mapping phase that conducts scheduling, resource binding, and datapath control generation in one pass scan of the SSA form based LLVM IR. The IR is directly mapped to a fully pipelined circuit with high throughput or input data rates. Resource constraint is temporarily relaxed in this phase such that the mapping can be finished in linear time.

3.3.1 Scheduling

Scheduling determines the relative start time for operations or computing tasks obeying data and control dependencies. Since resource optimizations are deferred to next phase, we choose

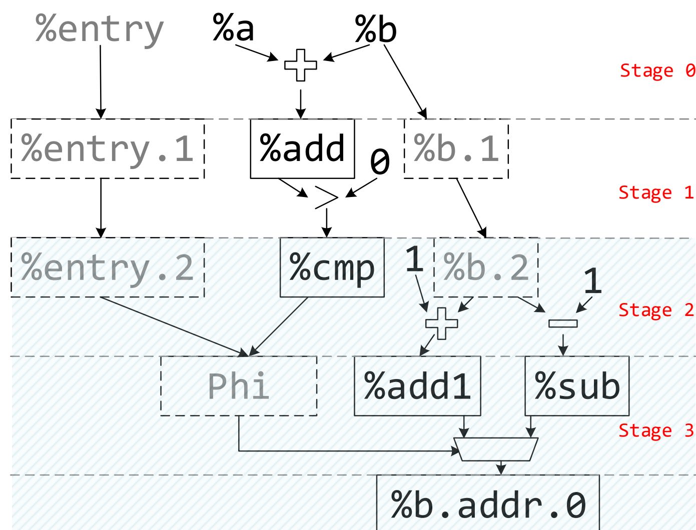


Figure 3.1: The storage binding for code 3.2. The circuit is divided into pipeline stages indicated by dashed horizontal lines. Each black font name corresponds to one register in code 3.2 and each gray font name indicates a newly added register for pipeline synchronization or control. Every variable has one pipeline register storing its value. The registers in dashed rectangles, except `Phi`, are for data synchronization in the pipeline. On FPGA, these registers can be implemented by configuring look-up tables (LUT) to shift registers and thus have low cost [4].

the as-soon-as-possible (ASAP) scheduling, which has linear time complexity and can be easily integrated in the one pass scan of the LLVM IR.

Before scheduling, one needs to estimate the delay or number of clock cycles for each operation. If the operations that require multiple clock cycles to finish are not pipelined, they would cause additional computing latency in the synthesized circuits. To simplify the synthesis, all the LLVM IR instructions performing actual computing tasks such as `add` and `sub` are assumed to take one clock cycle. Some trivial operations such as `sext`, the sign extension operation, and `trunc`, the truncation operation, are set to zero clock cycle, since they do not form a pipeline stage by themselves and can be inserted into the datapath with negligible latency overhead.

After the computing delay estimation, a control data flow graph (CDFG) is obtained to capture operation dependencies. The SSA form is originally used for control data flow analysis in compilers and thus the SSA form based LLVM framework already builds a dependence graph of

instructions in the IR, which can be reused for CDFG. Therefore, the ASAP scheduling is performed on the instruction dependence graph of the parsed LLVM IR. Each instruction is assigned to the first available step or pipeline stage. No two sequentially dependent LLVM instructions are assigned to the same stage (e.g. the addition preceding `%add` and the comparison before `%cmp` in Figure 3.1 are sequentially dependent), otherwise clock frequency must be reduced. On the other hand, a low frequency design with less registers can be obtained by merging some pipeline stages by simply removing the pipeline registers. For instructions with two or more inputs, often one input has data available several cycles earlier than the other inputs. Then, the input receiving data early requires additional pipeline stages using shift registers like variable `%b` in Figure 3.1. Every basic block has a starting pipeline stage to synchronize the data from its precedent basic blocks.

3.3.2 Storage Binding for Scalar Variables

In this mapping phase, the storage binding for scalar variations is straightforward. Each renamed scalar variable in the SSA form is bound to one register. If the value of a variable is propagated to later pipeline stages without computing operations, like variable `%b` in Figure 3.1, it is bound to a shift register. Otherwise the variable is bound to a discrete register. On FPGA, shift registers can be realized by configuring look-up tables (LUTs) such that a low cost is enjoyed [4]. For example, the pipeline registers of value `%b` in Figure 3.1 cost $32 \times 2 = 64$ flip-flops, which would occupy 32 slices but only use $32/2 = 16$ slices if implemented in LUTs (assuming there are 2 flip-flops and 2 LUTs in each slice of Xilinx FPGAs).

In Figure 3.1, each solid rectangle represents a discrete register, and each instruction that assigns a value in the LLVM IR has a distinct register allocated to store the value. Therefore, registers are never shared across different variables, or different values of the same variable in the original C program during its lifetime. Without register sharing, pipelining is well supported. At each moment, registers at different levels can store values of different program runs with different inputs. For instance, in Figure 3.1, `%b.addr.0` can be storing the final result of the first set of `%a` and `%b`. At the same time, `%add1` and `%sub` are storing the intermediate results of the second set of inputs, `%cmp` and `%b.2` are storing the intermediate results of the third set of inputs, and `%add`

and %b.1 are storing the intermediate results of the fourth set of inputs.

3.3.3 Datapath Control

Datapath synthesis involves assigning multiplexers to select appropriate inputs for some operations. Existing works such as [81, 15] often use a finite-state machine (FSM) to control such multiplexing. Pipeline stages are recorded as the FSM states. If there is no br instruction or other branch instructions (e.g. switch in LLVM IR), the next states of the FSM are statically determined without inputs to the FSM. If there are branches, the next states are decided dynamically according to the branch variables. In a fully pipelined circuit like our case, there are many pipeline stages that make the FSM design rather complex.

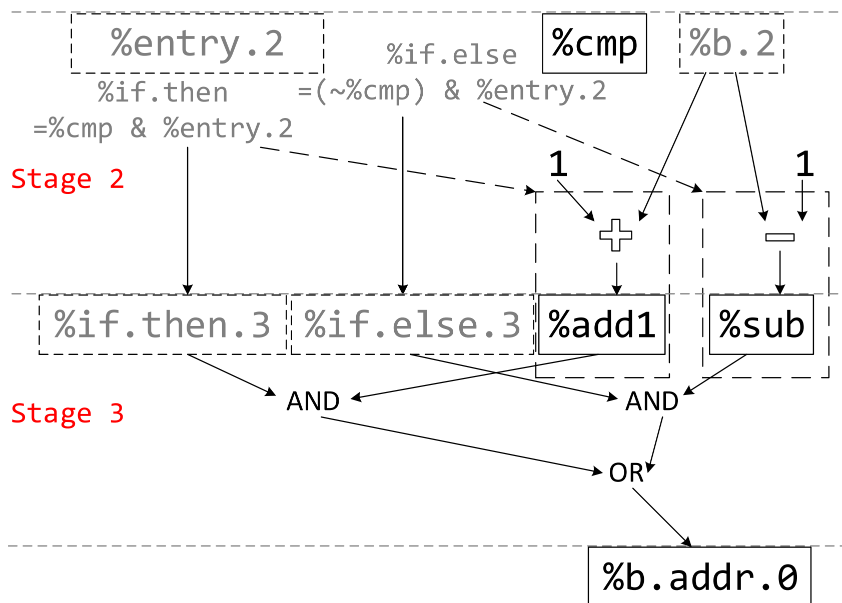


Figure 3.2: Details of the shaded region in Figure 3.1 (stage 2 and 3). The AND operations require that the enable signals such as %if.then.3 are replicated to have the same bit-width as the arithmetic operations. The dashed arrows and boxes indicate an alternative implementation of the Φ function where an enable signal resets the register in a dashed box (with synchronous active-low RST) such that the value is zero and the corresponding AND operation can be avoided.

We propose a new flow control method that dovetails with pipelined circuits as well as our

mapping procedure. It directly maps the Φ functions in SSA-form IR to control circuits. An example of implementing the Φ function in Figure 3.1 is shown in Figure 3.2. The variable `%cmp` is extracted to enable signals `%if.then` and `%if.else`, which are propagated to the next pipeline stage as `%if.then.3` and `%if.else.3`, respectively. The `AND` and `OR` operations with `%add1` and `%sub` basically select the result from the two arithmetic operations. Alternatively, one can reset one operation register, e.g. `%add1`, if corresponding enable signal is false. In this alternative approach, registers `%if.then.3` and `%if.else.3` can be omitted.

We describe the Φ function based flow control for a `C` language function using code 3.2 as an example. If the enable signal is 1 for the entry basic block, which starts with `entry:` in code 3.2, the inputs to this `C` function is available. If the circuit is fully pipelined, the input can be fed to the function continuously and the enable signal is 1 continuously as well. For non-entry basic blocks, their enable signals are determined by the operands of branch instructions like `br`. For example, one `br` may have one variable operand and two label operands. It selects the basic block marked by the first label if the variable has value 1, and otherwise selects the basic block marked by the second label. If `br` has only one label operand, it always selects the labeled basic block, and behaves like a jump instruction in assembly language.

The enable signal for each basic block is propagated through the entire block. For example, there is an enable signal (`%entry`) propagated through the basic block of lines 2-5 in code 3.2. The enable signal `%if.then` for basic block of lines 6-8 is 1 when the enable signal `%entry.2` is 1 and the `br` in line 5 selects basic block with label `%if.then:`.

3.3.4 Synthesis of Array Datapaths

Although scalar SSA has been widely used in modern compilers, array SSA [79] has not been supported very well. For example, the LLVM IR treats arrays in a similar way as C language, where arrays can only be accessed through a pointer. The instructions related to arrays are quite different from scalar instructions. To access an element in an array, `load` and `store` instructions are used. Before loading or storing value to a location in an array, an address or pointer of the element must be acquired using instructions such as `getelementptr`. All these intermediate

instructions are based on an assumption that there is a single large continuous memory address space, which microprocessors often have. However, hardware such as FPGAs have much more flexible and distributed memories with no such continuous memory address space. Moreover, these memory reference instructions are rarely compatible with parallelization, while hardware such as FPGAs are intrinsically parallel. Thus, these intermediate instructions used in LLVM IR do not fit hardware synthesis.

Array SSA has been studied for software compilers [79]. Such array SSA treats the elements in an array in a similar way to scalar variables, instead. It creates a new array every time a new value is assigned to the original array, and chooses the appropriate one to use later. The main difference from scalar SSA is that array SSA extends the usage of Φ functions to also merge two partially different arrays (i.e. some elements of the two arrays are different). With the array SSA extension, the generalized Φ function supports not only selection of different versions of variables, including both scalar variables and arrays, from different basic blocks, but also merging of the partially modified array and the original array, which is called *define-ϕ* in [79]. One example is shown in code 3.3.

```
X0[0:7] = {1,2,3,4,5,6,7,8}
X1[i] = 0;
X2 = phi(X0,X1)
```

Listing 3.3: An example of define-Φ: elements in `X1` other than `X1[i]` are undefined; elements in `X2` are selected either from `X0` or `X1`, according to whether the index equals i. `X0`, `X1` and `X2` all correspond to the same array in the original source code.

In [79], array SSA is used to do element-level data flow analysis and loop parallelization. However, it is for software compiling on microprocessors while the actual storage and hardware implementation are not specified for hardware. For HLS, we introduce a new technique of implementing array SSA transformations along with the register promotion of array elements in order to support element-level pipelining and parallelization for computations involving arrays, required

by some parallel algorithms involving arrays such that prefix sum [6]. The register promotion transforms the memory reference instructions such as `load`, `store` and `getelementptr` to register accesses. After the register promotion, array elements are stored individually similar to scalar variables. The key problem is how to construct new versions of the array across pipeline stages for the define-Φ functions.

In this work, two different methods are devised to synthesize the define-Φ for merging two versions of an array.

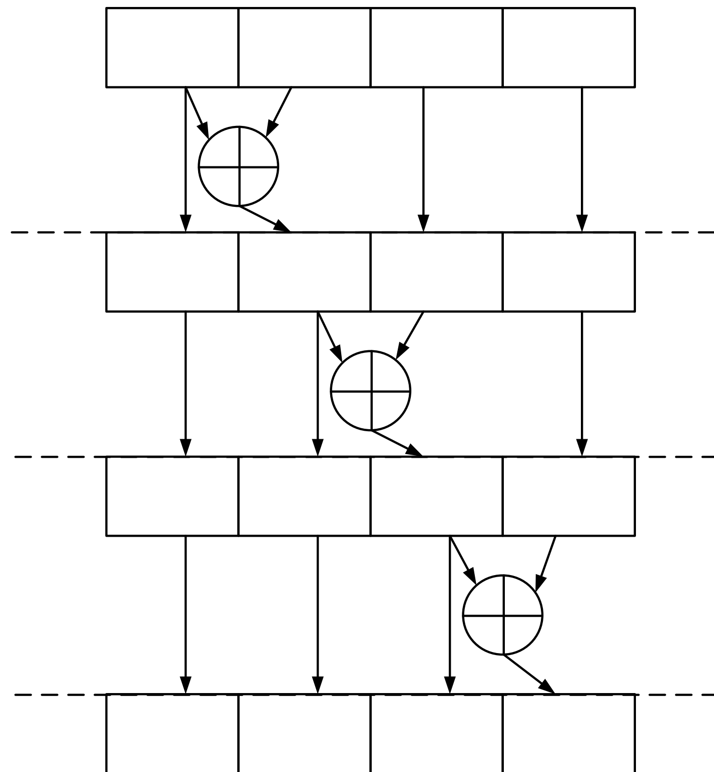


Figure 3.3: Example of array SSA: arrays are propagated to different pipeline stages.

Merging on Writing After each writing to the array, a new array is constructed. All the array elements then can be treated in the same way as scalar variables.

Merging on Reading The modified array element and the corresponding array index are stored elsewhere as a patch element and are not merged with the original array, until an element in the array is accessed.

Depending on how many elements in the original version of array are not constants, the two methods have different resource cost.If there are more constants in the array, the merging on reading method is better.One example of array merging on writing is shown in figure 3.3.

3.3.5 Loops

In this phase, no loop unrolling is performed, since it can be done more easily by the compiler front-end during the IR generation. For loops not unrolled, dynamically or at runtime the variables within loop bodies in the SSA-form IR might still be assigned different values multiple times. The name of "**Static** Single Assignment" just means the variables are assigned exactly once only statically at compile-time or literally in the SSA-form IR.

The loops, that result in cycles in the CDFG of the IR, are mapped directly to Verilog HDL from the SSA form, implying sharing both the registers and operators within the loop bodies by executing them multiple times (e.g., Figure 3.4c is equivalent to a loop adding 1 for five times), which inevitably causes pipeline hazards. How to avoid these hazards incurred by resource sharing is provided in Section 3.4.

3.4 Phase II: Resource Optimization

After the mapping phase, if any resource constraint is violated, our HLS proceeds with the resource optimization phase. Other than explicit resource sharing via a loop, operations of the same type in different pipeline stages can share the same operator or functional unit. In this optimization phase, resources are incrementally shared in an iterative manner so that the resource usage is lowered in sacrifice of input data rates. When resources are shared in pipelined circuits, the conflict of operations in different pipeline stages trying to use the same operator inevitably causes structural hazards. The hazards also exist in pipelined microprocessors such as MIPS (**M**icroprocessor without **I**nterlocked **P**ipe **S**tages), which are usually solved by interlocked pipelines. A simpler option

used by MIPS is inserting `NOP` instructions, which create bubbles in the pipelined microprocessors, by compilers or manually in the assembly code. Then, pipeline interlocking is not needed.

In this phase, both corresponding methods are supported: the manual insertion (or by something like a FIFO) of bubbles in the input stream, similar to the `NOP` insertions, is trivial; Other than that, automatic synthesis of the stall and bubble signals for pipeline interlocking is also provided.

3.4.1 Iterative Resource Sharing

The number of possible combinations of sharing is very large and with the number of operations increasing, grows even faster than exponentially. To implement m distinct operations with n identical operators, where $m \geqslant n > 0$, the number of possible sharing combinations is equal to the number of ways to partition a set of size m into n non-empty subsets, known in combinatorics as *Stirling numbers of the second kind* or $\{{m \atop n}\}$. In particular,

$$\{{m \atop 1}\} = \{{m \atop m}\} = 1$$

which means there is only one way for m operations to share a single operator, and one way to implement m operations with m operators (assuming no wasted operators), i.e., no sharing at all. Then, the total number of possible combinations of sharing for m distinct operations with any number of identical operators is

$$B_m = \{{m \atop 0}\} + \{{m \atop 1}\} + \{{m \atop 2}\} + \cdots + \{{m \atop m}\} = \sum_{k=0}^{m} \{{m \atop k}\}$$

where B_m is the mth *Bell number*, huge even for a small m. For example, $B_{19} = 5,832,742,205,057$.

Although the number of possible combinations of sharing is very large, lots of them are illegal (e.g., operations in the same pipeline stage) or impossible to implement with interlocked pipelines. In this work, we choose a representative subset of all possible combinations of sharing, where operations sharing the same operator can be iteratively added and the pipeline interlocks can be added accordingly without causing deadlock. The operations need to be evenly distributed in

different pipeline stages. Formally, if the pipeline stages are numbered starting from 0, according to the order of input to output like that in Figure 3.1, 3.4 and 3.5, then we can define that a list $OP = \{op_0, op_1, \ldots, op_{N-1}\}$ of N operations is a *periodic sharing list*, if and only if

$$\forall m, n \in \{0, 1, 2, \ldots, N-1\} \; \exists K \in \mathbb{N}^* : \\ stage(op_m) - stage(op_n) = K \cdot (m - n) \tag{3.1}$$

where K is a positive constant integer indicating the *period* of the periodic sharing list, and $stage(op_n)$ represents the number of the pipeline stage of op_n. Equation (3.1) implies

$$stage(op_m) - stage(op_n) = 0 \Leftrightarrow m = n \tag{3.2}$$

$$stage(op_m) - stage(op_n) \equiv 0 \mod K \tag{3.3}$$

$$stage(op_m) - stage(op_n) = stage(op_p) - stage(op_q) \Leftrightarrow m - n = p - q \tag{3.4}$$

We define that an empty list and a list with only one operation are also periodic sharing lists. If the operations in a periodic sharing list all share the same operator, the maximal asymptotic data rate without causing structural hazards will be f/N, where f is the clock frequency. Note that N is not the same as the initiation interval, which is the number of clock cycles between the start times of two consecutive valid inputs. Only when $K = 1$, they are equal. For example, If $K = 3$ and $N = 3$, an input stream of the enable signals without structural hazards will be $1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, \ldots\ldots$. The pattern cannot even be specified by the function initiation interval pragma (`#pragma HLS PIPELINE II=X`).

Figure 3.4 shows a special example of iterative resource sharing, where the period K is 1. The operations are iteratively added to a periodic sharing list with period 1. Figure 3.4b shows the circuit when two operations are in the sharing list; Figure 3.4c shows the circuit when all operations are added to the sharing list. Note that the multiplexers in Figure 3.4c can be simplified to only one multiplexer, like the implementation of a loop, for this special periodic sharing list where the

Data:
$OP \leftarrow \{\}, N \leftarrow 0$
$K \leftarrow$ the period of the periodic sharing list OP
$IT \leftarrow 1$ i.e., VDD
$\{IN_i\} \leftarrow$ undefined wire signals
$STALL \leftarrow$ an undefined wire signal
$BUBBLE \leftarrow$ an undefine wire signal
Input:
op, the operation to share the same operator with operations in OP
Results: Updated $STALL, BUBBLE, OP, N, IT, IN_i$

```
1  if N = 0 then
2  |   OP ← {op_0 = op}
3  |   N ← N + 1
4  |   foreach IN_i do
5  |   |   IN_i ← IN_i(op)
6  |   end
7  else if stage(op) - stage(op_{N-1}) = K then
8  |   OP ← {op_0, op_1, ..., op_{N-1}, op_N = op}
9  |   N ← N + 1
10 |   foreach IN_i do
11 |   |   IN_i ← MUX{en[stage(op) - 1], IN_i, IN_i(op)}
12 |   end
13 |   IT ← OR{IT, en[stage(op) - 1]}
14 else if stage(op) - stage(op_{N-1}) ≠ K then
15 |   Illegeal op
16 end
17 if op is legal and N ≥ 2 then
18 |   STALL ← AND{IT, en(stage(op_0) - 1)}
19 |   BUBBLE ← AND{NOT(IT), en[stage(op_0) - 1]}
20 end
```

Algorithm 3: Update the stall, the bubble and the input signals of the operator to be shared, after adding an operation to a periodic sharing list sharing the same operator.

output of op_n is the input of op_{n+1}.

3.4.2 Pipeline Interlock Synthesis

Resource sharing may hinder pipelining and lower the throughput. If the input data rate is already lowered with bubble insertion in the input stream, no pipeline hazards would occur. Otherwise, the circuit cannot work correctly without proper pipeline interlocks.The even distribution

condition of the periodic sharing list defined in (3.1) guarantees that if there is only one periodic sharing list sharing the same operator, the resource conflicts could only happen between the op_0 and one of the remaining operations in the list at a certain clock cycle and no deadlock would happen. This property can proved as follows:

Proof. (1) For a periodic sharing list $\{op_0, op_1\}$, the resource conflicts can only happen between op_0 and op_1.

(2) i Assume for a periodic sharing list $\{op_0, op_1, \ldots, op_{N-1}\}$, the resource conflicts can only happen between op_0 and op_n where $0 < n < N$. Then, if we add one operation in the list, the list becomes $\{op_0, op_1, \ldots, op_{N-1}, op_N\}$. Since the resource conflicts between op_0 and op_{N-1} will be resolved, no resource conflicts will happen between op_1 and op_N (By equation (3.4), $stage(op_N) - stage(op_1) = stage(op_{N-1}) - stage(op_0)$).

ii According to the assumption, there are no resource conflicts between op_n and op_{N-1}, where $0 < n < N - 1$. So, no resource conflicts between op_{n+1} and op_N (By equation (3.4), $stage(op_N) - stage(op_{n+1}) = stage(op_{N-1}) - stage(op_n)$).

iii According to i and ii, no resource conflicts will happen between op_m and op_N , where $0 < m < N$, i.e., for periodic sharing list $\{op_0, op_1, \ldots, op_{N-1}, op_N\}$, resource conflicts can only happen between op_0 and op_n, where $0 < n < N \Rightarrow$ recource conflicts can only happen between op_0 and op_m, where $0 < m < N + 1$.

(3) By induction, for every periodic sharing list $\{op_0, op_1, \ldots, op_{N-1}\}$ sharing one operator with interlocked pipelines, resource conflicts can only happen between op_0 and op_n, where $0 < n < N$.

□

This property substantially reduces the complexity of the pipeline interlocking. The stall signal and the bubble signal are only needed for the operation op_0. However, if there are multiple periodic sharing lists, each sharing an operator respectively, they have the above property when at least one of the two following conditions is satisfied:

1. All the periodic sharing lists have the same period K.

2. Any two of the periodic sharing lists don't overlap, i.e., for every two periodic sharing lists OP^0 and OP^1,

$$\text{either, } stage[\min(OP^0)] > stage[\max(OP^1)]$$
$$\text{or, } stage[\min(OP^1)] > stage[\max(OP^0)]$$

   where, for a list $OP = \{op_0, op_1, op_2, \ldots, op_{N-1}\}$,

$$\min(OP) = op_0,\ \max(OP) = op_{N-1}$$

Then, for a periodic sharing list, the corresponding stall, bubble and input signals can be synthesized iteratively with Algorithm 3. The periodic sharing list OP is first initialized to an empty list. The IT is an internal variable representing a wire to generate the $BUBBLE$ and the $STALL$. The IN_i is the ith input to the shared operator. Then, the algorithm is iterated and the operations is added one by one. The IN_i and IT are also updated iteratively. The OR, AND,MUX and NOT mean generating corresponding gates. Particularly, the first input of MUX is the selection signal sel. The second input is selected when the sel is low; the third input of MUX is selected when sel is high. Figure 3.5 shows how they are synthesized for the same example in Figure 3.4.

3.4.3 Sharing for Loops

```
define i32 @man(i32 %a, i32 %c, i32 %e) #0 {
  br label %1
; <label>:1
  %.0 = phi i32 [ %e, %0 ], [ %4, %5 ]
  %i.0 = phi i32 [ 0, %0 ], [ %6, %5 ]
  %2 = icmp slt i32 %i.0, %a
  br i1 %2, label %3, label %7
; <label>:3
```

```
  %4 = add nsw i32 %.0, %c
  br label %5
; <label>:5
  %6 = add nsw i32 %i.0, 1
  br label %1
; <label>:7
  ret i32 %.0
}
```

Listing 3.4: Example LLVM IR for program with a loop. The basic block 5 has basic block 1 as its succeeding basic block, and the control flow through basic block 1 is possible to continue to go through basic block 5, which constitutes a loop.

As shown in code 3.4, the basic blocks %1, %3, and %5 are part of the loop as shown in figure 3.6. There are two preceding basic blocks for basic block %1, and one of them is part of a loop. We can not use the same techniques as in figure 3.1 and figure 3.2, since the starting level of basic block %1 would be infinitely large. However, the techniques for resource optimization can be used again. As mentioned before, each basic block has an enable signal. Each `br` instruction also generates one or two control signals. In the case in figure 3.6, the basic block %5's terminating `br` instruction has only one control signal, which together with the control signal from basic block %0 controls the enable signal of basic block %1. Similar to the resource optimization case in figure 3.4, if both the control signals are high, we know there is a conflict and pipeline stall is need to resolve the conflict. In the above case in figure 3.6, the loop control signal would have higher priority. If not, the intermediate results stored in the pipeline registers in the loop would be destroyed by new inputs from basic block %0, which are the inputs of this function.

3.5 Support for parallelization

Hardware such as FPGAs has intrinsic parallelism, which is worth to be well exploited in HLS, especially for throughput driven applications. However, previous works on C/C++-based HLS mostly rely on loop transformations and code optimizations for parallelization instead of making good use of this intrinsic parallelism. Specially designed parallel algorithms [82] also

lack adequate support from HLS. This section shows how our work achieves better parallelization support.

3.5.1 Array SSA and Parallelization

Parallel algorithms often require parallel access to arrays, which is enabled by the register promotion in this work. For parallelization of pipelined circuits, our array SSA implementation further provides an intermediate representation form that helps build additional pipelining storage for arrays and the datapath for different versions of the array across multiple pipeline stages.

The array SSA requires substantially higher register usage, as shown in figure 3.3, to achieve pipelining and parallelization. However, this can be alleviated by register sharing using Algorithm 3.

3.5.2 Structural Recursion

Recursive function calls are usually not supported by HLS frameworks. Generally, recursion requires additional memory space for the call stack and a controller such as that in [83]. The state machine controller and memory stack make pipelining and parallelizing even more difficult. We notice that one special recursive function call, the structural recursion, can be integrated into our HLS flow for pipelined and parallelized circuits. Structural recursion decomposes the inputs (e.g. an array) recursively till the inputs cannot be decomposed. In contrast, generative recursion generates new inputs for the recursive function calls.

```
1  generate
2  /* Recursively decomposing the input a */
3  if(SIZE == 1) begin
4    assign out0[31:0] = a[31:0];
5    assign out1[31:0] = 32'b0;
6  end else if(SIZE == 2) begin
7    assign out0[31:0] = a[31:0];
8    assign out1[31:0] = a[63:32];
9  end else begin
10   recursion #(.SIZE( SIZE / 2) )  re0(
```

```
11      .a(a[32*(SIZE / 2) - 1:0]),
12      .clk( clk),
13      .out(out0));
14    recursion #(.SIZE( SIZE - SIZE / 2) ) re1 (
15      .a(a[32*SIZE - 1:32*(SIZE / 2)]) ,
16      .clk( clk),
17      .out(out1) );
18  end
19  endgenerate
```

Listing 3.5: The recursive `generate` struct in Verilog that can be used to implement structure recursion. This example shows the part for the structure of figure 3.8 with 32 inputs.

Some parallel algorithms can be naturally expressed as structural recursion such as part of the structure in parallel prefixsum [6, 84] as shown in figure 3.8, and the bitonic sorter [7], which is shown in figure 3.7. We implemented the structural recursion and synthesized the parameterized Verilog module with `generate` construct, which supports the recursive module instantiation. One example of the `generate` construct is shown in code 3.5.

3.6 Experimental Results

We implemented our high-level synthesis system with LLVM compiler [77] as the front-end, similar to existing tools such as LegUp [18] and Altera FPGA SDK [85]. The C code is first transformed to the LLVM IR using LLVM-based C compiler "clang". The proposed mapping-based HLS, performed with and without the the resource optimization phase, transforms the LLVM IR to Verilog HDL code. We also generated Verilog HDL code using a state-of-the-art commercial HLS tool. We tried several different settings of the commercial tool for each case and selected the best results. After HLS, all the Verilog codes are fed into Xilinx Vivado for logic synthesis, place and route with targeted device of Zynq-7000 family FPGAs. The experiments are conducted on a PC with 2.0GHz processor and 8GB memory.

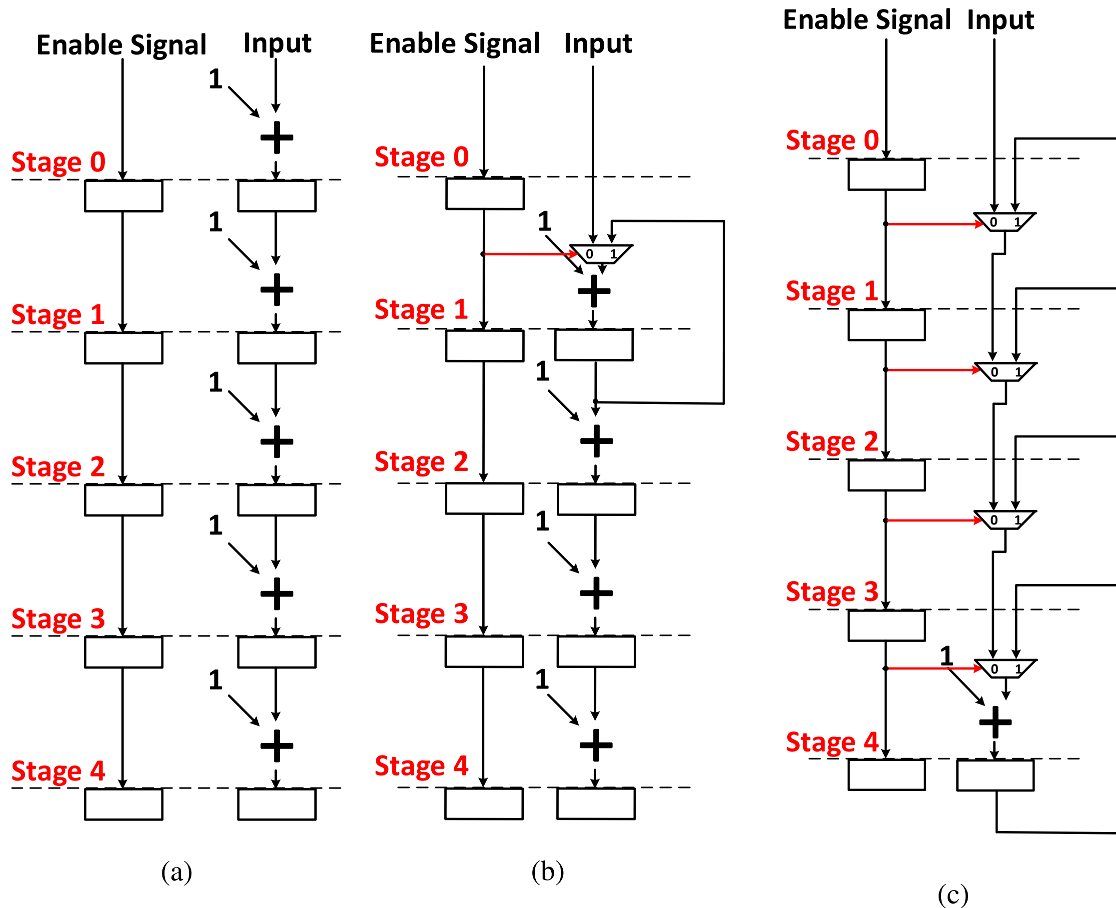


Figure 3.4: Example of resource sharing: the enable signals, similar to the `%entry` in Figure 3.1, are initialized to 0. (a) The original circuit with no sharing. (b) Two operations sharing the same one adder. This shared adder selects inputs according to the enable signal. The enable signal asserts when the corresponding input is valid. If the enable signals do not assert consecutively, there is no structural hazard. A bubble needs to be inserted between two consecutive valid inputs to avoid hazards. (c) All operations sharing the same operator, equivalent to a loop adding 1 for five times.

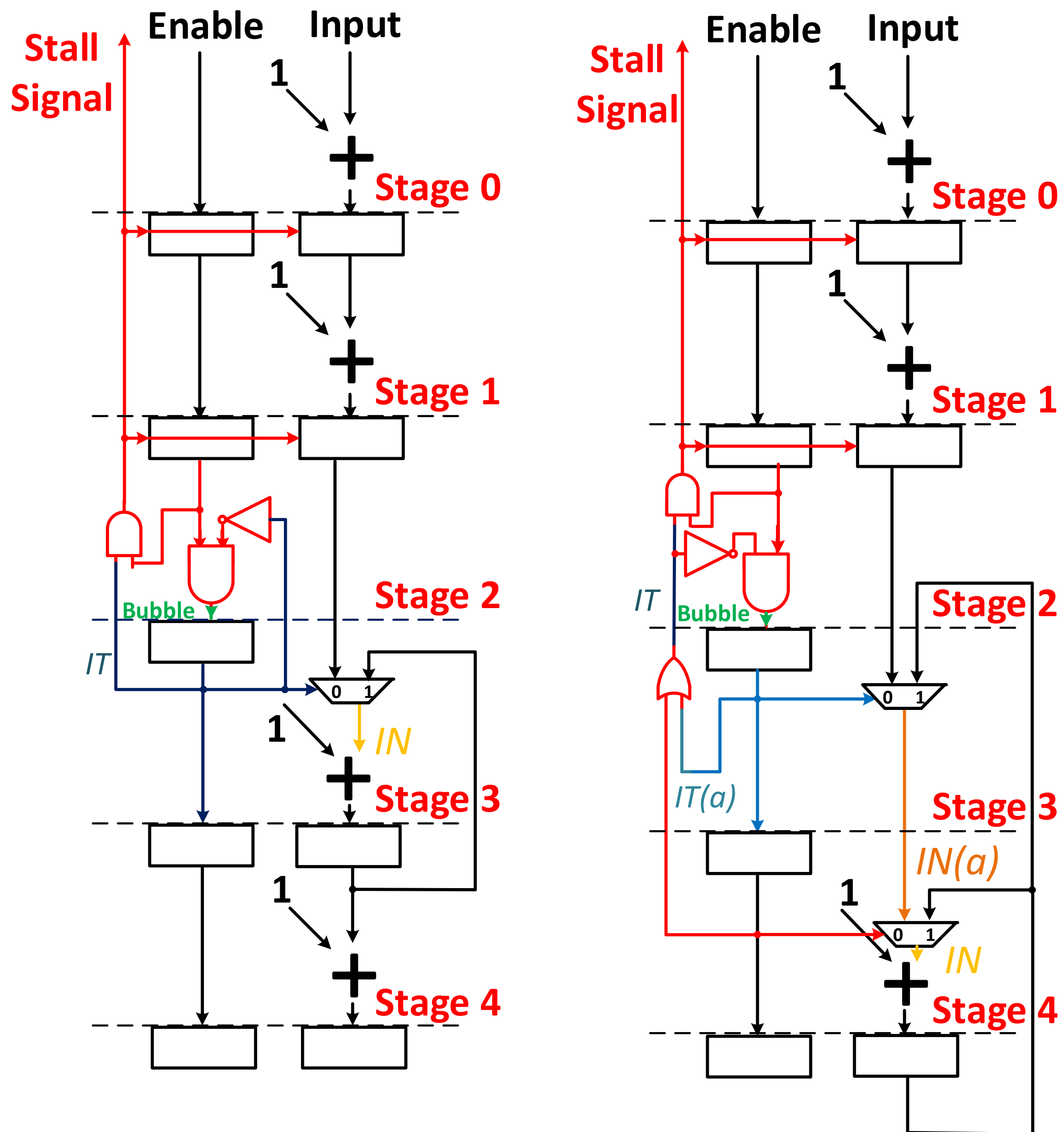


Figure 3.5: Example of interlocked pipelines for the same example in figure 3.4: (a) two operations sharing one operator. (b) another operation added to the periodic sharing list sharing the same operator with the two operations in (a). The $IN(a)$ and $IT(a)$ are the same as the IT and IN in (a) and are updated with Algorithm 3.

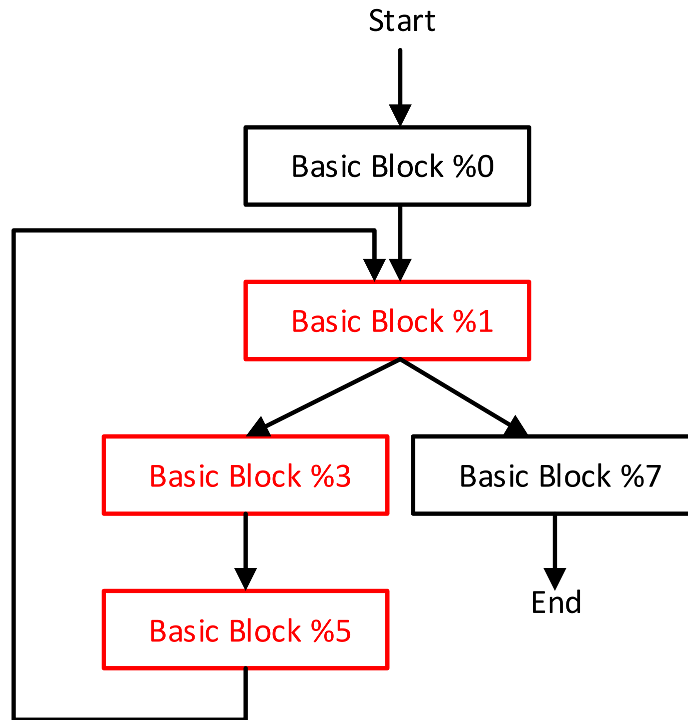


Figure 3.6: Diagram of the basic blocks in code 3.4: the basic blocks in red are part of the loop, which can be shared among different loop cycles. The back edges would also cause pipeline stall, similar to the back edges in figure 3.4.

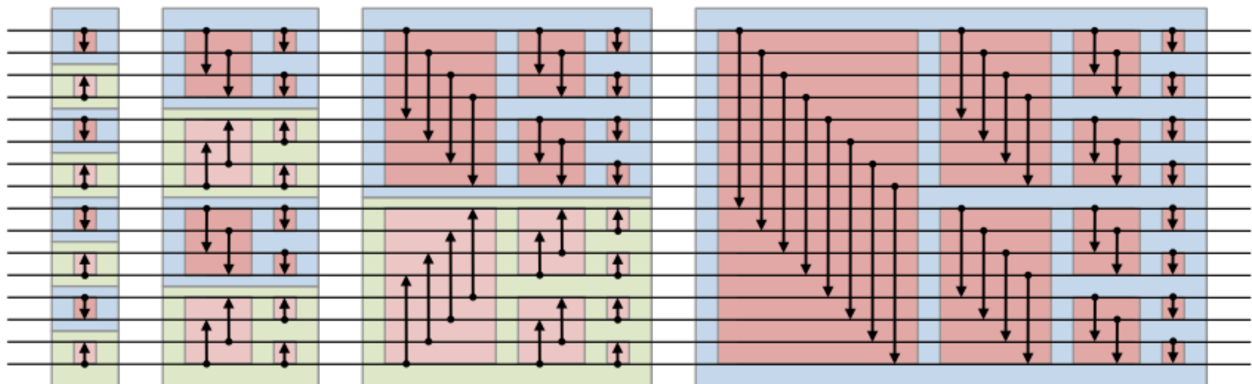


Figure 3.7: Bitonic sorter: another example of structural recursion. Each line represents a number. The arrows mean swapping the two numbers. The bitonic sorter is a common structure for sorting in parallel. Figure courtesy of Magnus Manske.

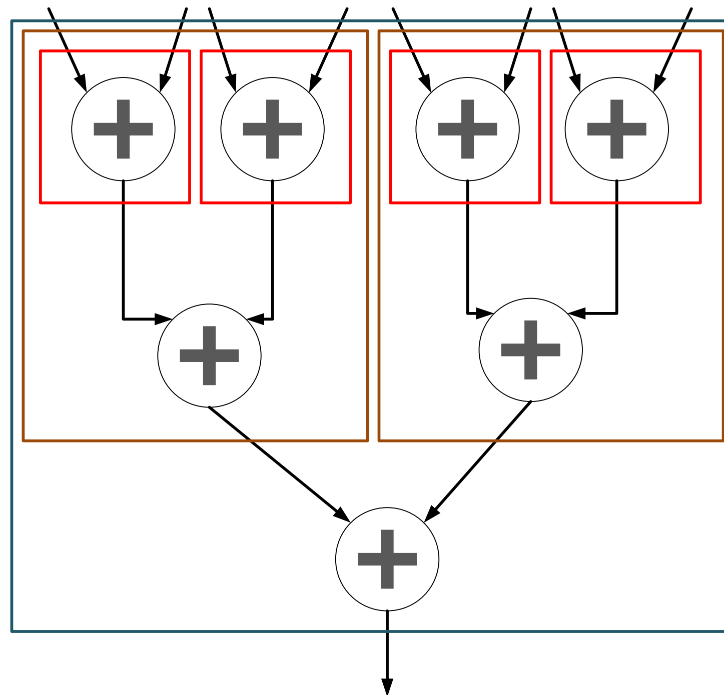


Figure 3.8: Example of structural recursion. The inputs are recursively decomposed. Each box represents a recursive structure except the upper four terminating box.

		#LUT	#Register	#DSP	Frequency (MHz)	Latency (ns)	Power (mW)	Data Rate (MHz)	HLS Runtime (s)
Add	Mapping	18	35	0	276.47	10.85	1	276.47	0.056
	Commercial	19	28	0	190.84	10.48	1	190.84	3.440
Mul	Mapping	78	133	6	120.05	24.99	12	120.05	0.057
	Commercial	80	169	6	128.04	46.86	11	128.04	3.180
FIR8*	Mapping	367	553	0	233.10	34.32	17	233.10	0.064
	Commercial	252	227	0	137.93	50.75	9	17.24	3.480
Sscan*	Mapping	352	896	0	129.03	54.25	12	129.03	0.062
	Commercial	376	263	0	147.06	47.60	8	18.38	2.890
Pscan*	Mapping	432	960	0	118.20	33.84	13	118.20	0.065
	Commercial	359	263	0	104.06	67.27	7	13.01	3.400
MM*	Mapping	2776	3648	192	62.66	63.84	178	62.66	0.078
	Commercial	1305	867	96	88.14	170.18	124	5.51	18.370
Bsort	Mapping	4544	4607	0	202.72	49.33	334	202.72	0.060
	Commercial	5390	4969	0	151.72	65.91	305	151.72	6.210
SHA256	Mapping	32895	41859	0	198.65	956.46	2232	198.65	0.228
	Commercial	31092	45976	0	199.72	1266.77	2200	199.72	121.960
Average	Mapping	119.33%	219.69%	150.00%	114.14%	72.13%	136.03%	571.46%	1.36% (73.5x)

Table 3.2: The comparison of post-layout results obtained from our mapping phase alone without resource optimization and a state-of-the-art commercial HLS tool. The Sscan and Pscan are designs for sequential scan and parallel scan as described in [6]. Bsort is a Bitonic sorter of 16 32-bit numbers [7]. SHA256 is 256-bit secure hash algorithm fully unrolled except the outer-most loop. Designs with arrays are marked with *. The average percentages are our mapping results compared with the commercial tool.

.

		#LUT		#Register		#DSP	Frequency (MHz)		Latency (ns)		Power (mW)		Data Rate (MHz)		Time
Add	-	10	56%	35	100%	0	278.32	101%	10.78	99%	1	100.00%	139.16	50%	104%
	+	12	67%	35	100%	0	266.95	97%	11.24	104%	1	100.00%	133.48	48%	107%
Mul	-	95	122%	116	87%	3	127.36	106%	23.56	94%	9	75.00%	63.68	53%	109%
	+	97	124%	116	87%	3	127.62	106%	23.51	94%	9	75.00%	63.81	53%	112%
FIR8	-	288	78%	525	95%	0	232.88	100%	34.35	100%	17	100.00%	116.44	50%	100%
	+	260	71%	1284	232%	0	220.26	94%	36.32	106%	13	76.47%	110.13	47%	100%
Sscan	-	257	73%	836	93%	0	119.22	92%	67.10	124%	11	91.67%	59.61	46%	105%
	+	233	66%	1479	165%	0	131.37	102%	60.90	112%	9	75.00%	65.69	51%	110%
Pscan	-	353	82%	899	94%	0	117.69	100%	33.99	100%	14	107.69%	58.84	50%	105%
	+	359	83%	1028	107%	0	115.55	98%	34.62	102%	9	69.23%	57.78	49%	105%
MM	-	2377	86%	3507	96%	192	64.36	103%	62.15	97%	174	97.75%	32.18	51%	100%
	+	2621	94%	4132	113%	192	69.92	112%	57.21	90%	187	105.06%	34.96	56%	104%
AVG	-		83%		94%	75%		100%		103%		95.35%		50%	104%
	+		84%		134%	75%		101%		101%		83.46%		51%	106%

Table 3.3: The post-route results obtained from our HLS after the resource optimization phase, with two variants: pipelined circuits without interlock (marked with "-") and interlocked pipelined circuits (marked with "+"). The percentages are compared with the mapping results in Table 3.2. The last column is the total HLS runtime (Phase I + Phase II) compared with the mapping results (Phase I).

A comparison of the post-layout results from our mapping phase alone and the commercial tool is provided in Table 3.2. On average, our mapping-based HLS achieves about $74\times$ speedup with same or better performance. The direct mapping leads to average data rate that is over $5\times$ faster than the commercial tool. The price paid here is more resource utilization, especially registers, as resource constraints are relaxed in this phase. The results in Figure 3.9 is to show the importance of our array SSA implementation. Without the array SSA, the remaining part of our HLS cannot reach high throughput or input data rates.

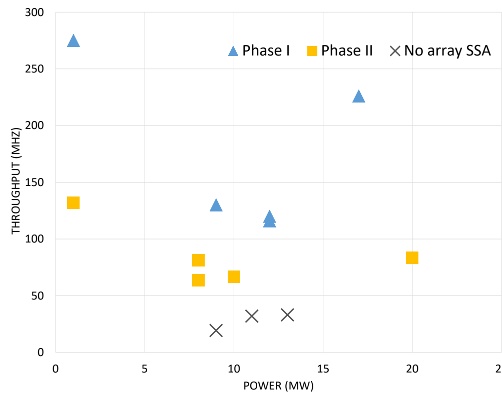


Figure 3.9: The throughput-power comparison among different methods. Each point represents a case. The no-array-SSA results are from modifying our HLS by removing the use of array SSA.

We also conducted experiments with both the mapping and resource optimization phases of our HLS. In the experiment, the number of shared operations is limited to no more than four. The results of both variants, pipeline with and without interlock, are obtained and shown in Table 3.3. The LUT usage is lower with both variants. However, the interlocked pipelined circuits consume about 42% more registers while achieves about 13% power saving on average. The power saving is from the local clock gating, which lowers the switching activities, but the clock-gated registers cannot be mapped to LUT shift registers. Thus, the register usage is significantly higher than those without interlocking.

Our techniques have some advantages that are difficult to be evaluated in a quantitative manner.

For example, the support to structural recursion is an yes/no feature and difficult to be quantified. Our automatic pipeline interlock synthesis would help reduce trial HLS runs in design space exploration. However, its evaluation requires to expand the research scope to not only HLS but also DSE.

3.7 Conclusions and Future Works

We develop a fast mapping-based high level synthesis technique, which leads to $74\times$ speedup over a commercial tool. Such fast speed will facilitate extensive solution search in design space exploration. Although our technique is described and validated on FPGA, it can be extended to ASIC HLS as well. One important feature of our technique is the support to pipeline synthesis, especially the synthesis of pipelined circuit with interlock, which is the first such work, to the best of our knowledge. The local incremental resource optimization helps reduce the number of HLS trials over different pragmas in source code. Moreover, our HLS can handle structural recursion, which is not well addressed in previous works. Our HLS often leads to circuits with higher data rates than the commercial tool, but at the expense of resource utilization increase. In future research, we will refine the resource optimization and validate the proposed techniques in design space exploration.

Although for many cases, the phase of resource optimization is not needed, it can be extended and further divided into more optimization passes similar to those in LLVM compiler framework. Since the LLVM IR is originally designed for microprocessors with a large continuous-address main memory, which is not true in most hardware other than microprocessor-centric systems, a newly designed IR might be a better choice than LLVM IR for the HLS phases and possibly fine-grained HLS passes.

Some functional programming languages prefer use of parallel patterns such as `map`, `filter` and `reduce` [86] and recursions rather than loops. Recent work has applied these functional languages patterns for generating hardware [58, 12]. These functional programming languages might be better fitted for high-level synthesis for hardware to support parallel computing. Theoretically, this mapping-based HLS technique should be able to work with any programming languages hav-

ing SSA-form based compiler front-end. Other programming languages with support of functional parallel patterns might be a better choice than C as the front-end language.

4. CONTROL CIRCUIT SYNTHESIS FOR ADAPTIVE SUPPLY VOLTAGE DESIGNS[1]

4.1 Introduction

Despite the slowing down of VLSI technology scaling, market needs continuously drive for higher performance, more functionalities and better energy-efficiency in chip products. To achieve these goals under the challenges of power and variability, new designs include more and more self management features [87], which make a circuit adapt its operations and resource allocation to circuit state and environment. One such feature is adaptive supply voltage (ASV) [88, 89, 5, 90, 91], which adjusts circuit supply voltage according to its own variations. On-chip delay sensors have been developed [92, 93, 94, 90] to detect the variations and enable truly autonomous ASV adjustment. For example, the supply voltages of a chip can be tuned right after fabrication to compensate its process variations, and later periodically tuned to mitigate circuit aging effects. Through the dynamic power adjustment, there is no need to constantly maintain large timing margin, which accounts for a considerable portion of overall chip power. By using per-core ASV, IBM Power7 processor [90] achieves about 24% chip power reduction without performance loss.

ASV granularity makes difference on the efficiency of resilience against process variations and circuit aging. Coarse-granularity means a small number of large adaptivity blocks [88, 90], wherein all cells in a block share the same tuning actions. Less blocks implies less tuning knobs and relatively simple control. This explains why coarse-grained ASV is quickly adopted in industrial products [90]. However, some variation effects are intrinsically fine-grained, e.g., transistor doping fluctuations. Fine-grained ASV allows more precise control of power allocation and conceptually offers more power saving, which is confirmed by recent studies [5, 91]. Nevertheless, its control can be complicated. For instance, in Figure 4.1, a delay sensor (canary flip-flop [92, 93]) may involve paths passing through multiple adaptivity blocks, and the supply voltage of one block may affect multiple sensors. This is in contrast to coarse-grained ASV, where each sensor is associated

[1]Reprint with permission from "Control Synthesis and Delay Sensor Deployment for Efficient ASV Designs" by Chaofan Li, Sachin S. Sapatnekar and Jiang hu, 2016. *Proceedings of the 35th International Conference on Computer-Aided Design*, Article 64, Copyright 2016 by the authors.

with only one block. Moreover, circuit aging can be partially recovered if the circuit is powered off or gated for a long time. As such, the supply voltage tuning is not always monotone and thus the control can be even more complicated. The few existing methods on fine-grained ASV are for regular datapath [5] or small circuits [91], and none of them elaborates the control design. To the best of our knowledge, there is no previous work for control design of general fine-grained ASV.

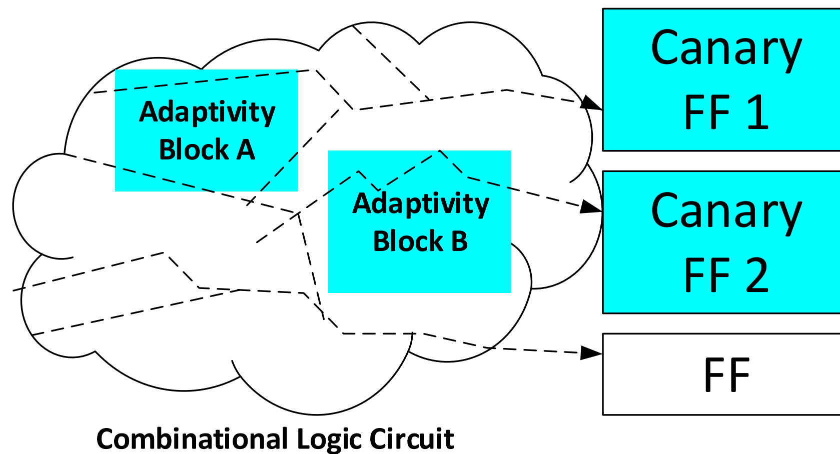


Figure 4.1: Fine grained ASV using canary flip-flop as delay sensors for detecting process variations and circuit aging. Dashed lines indicate timing paths.

Previous studies such as [5, 91] have shown the need for fine-grained ASV. In [5], a fine-grained voltage tuning technique – voltage interpolation, is proposed for processor designs and shown to be effective in addressing delay variations. In [91], a dual-level ASV system with fine-grained voltage regulators is reported. It shows superior power-delay trade-off curve compared to conventional coarse-grained ASV system.

One question is whether or not the need for fine-grained ASV is well justified, especially considering the increased design complexity. The answer lies in the history of VLSI technology advancement, e.g., the changes from constant supply voltage to dynamic voltage scaling, from single threshold voltage (Vt) to multi-Vt design, from single voltage/clock domain to multi-voltage/clock

domains. These once forefront techniques are now ordinary and prevalent, despite that they increase design complexity. The technology trend foresees that the power challenge will continue to worsen. Hence, one can expect that the additional power savings from fine-grained ASV will become a necessity.

We introduce two design techniques for the control of general fine-grained ASV. One is a rule-based control that is derived according to min-cost network flow model. The other is Finite State Machine (FSM) control, which is obtained through a new design methodology. Moreover, new delay sensor deployment techniques are proposed. All the control design and sensor deployment techniques can be fully automated and incorporated in EDA software. Experiments are performed on ICCAD 2014 Incremental Timing Driven Placement Contest benchmark circuits, which include cases of near one million gates. The results show that our techniques achieve 20% leakage power reduction compared to coarse-grained ASV, while maintain about the same timing yield in presence of process variations and aging effects. The proposed delay sensor deployment techniques also outperform the only previous work.

4.2 Backgrounds on ASV

ASV is an effective approach to power-efficient resilience against process variations and circuit aging in nanometer VLSI designs. An ASV design usually consists of three main components: (1) delay sensors that detect circuit delay variations, (2) voltage tuning knobs that can change circuit supply voltage, (3) control circuit that takes sensor results as inputs and decides the tuning knob actions.

Broadly speaking, there are two types of delay sensors – critical path replica [94, 90] and canary flip-flop [92, 27]. A critical path replica is separated from functional circuits and relies on probabilistic correlations to sense delay variations of actual circuits. The separation allows a relatively easy integration with conventional design flows. A canary flip-flop extends conventional flip-flop by adding a redundant sampling to additionally delayed logic signals [92] or logic switching detection near clock edge [27]. It is placed within the actual circuits, and therefore its sensing is more accurate than critical path replica. In addition, it has relatively small size and is friendly to

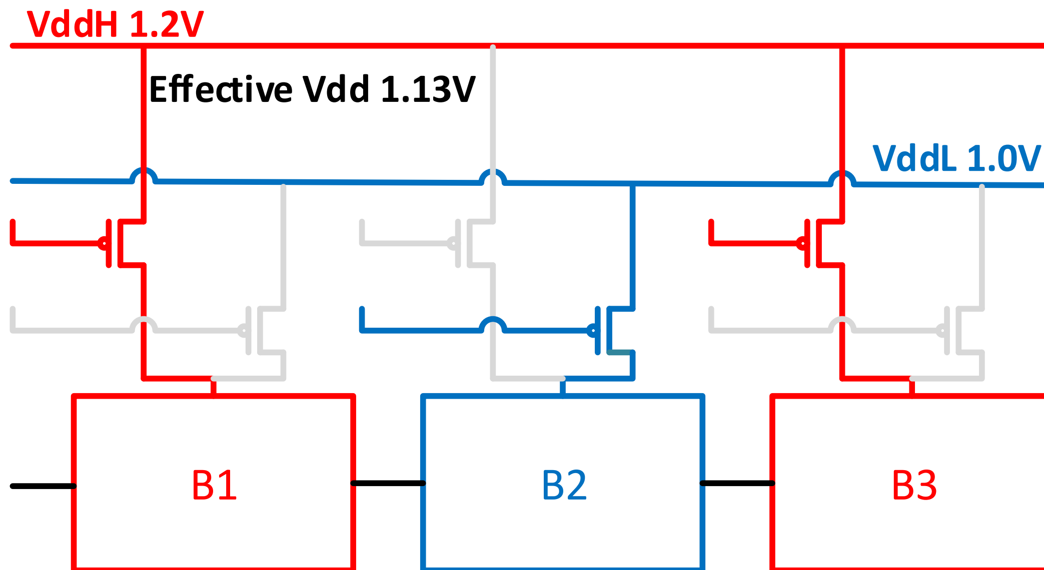


Figure 4.2: Schematic of voltage interpolation [5].

fine-grained ASV design.

Supply voltage tuning can also be carried out in two different ways. In the first approach, the supply voltage of a circuit block is tuned by a dedicated voltage regulator [90, 91]. This approach provides relatively large tuning flexibility. However, regulators usually have large area overhead and their use is mostly restricted to coarse-grained ASV. The other approach is voltage interpolation [5], which divides combinational circuits into different blocks, and supplies two VDD lines of different voltages to multiple blocks. Each block can choose higher or lower VDD between the two lines. A schematic of voltage interpolation is provided in Figure 4.2. In this way, the timing path passing through different blocks is operating at an "effective" supply voltage, which is determined by the supply voltage combination of these blocks. Since it needs only two regulators regardless the number of blocks, this approach is flexible in accommodating different adaptivity granularities. Experimental results in [5] also shows that if the difference between the two supply voltages is small, level shifters are not needed.

The control design for coarse-grained ASV is straightforward and similar to the control for dynamic voltage frequency scaling (DVFS). In DVFS, the voltage and frequency are usually adjusted according to workload and controlled by runtime software [95]. In ASV designs, the voltage tuning is mostly for compensating process variations and aging effects. In coarse-grained ASV, the voltage tuning of an adaptivity block solely depends on the block's own sensors and there is almost no interaction among different blocks. For fine-grained ASV, as illustrated in Figure 4.1, the control is a MIMO (multi-input and multi-output) system and its design is no longer straightforward. For example, if the fanin cone of FF2 has more overlapping cells with block B than with block A, and there is a warning signal from FF2, it is better to tune the voltage of block B. Even though it is possible that the aging in block A causes the warning, tuning block B to high voltage is still able to compensate the variations warned by FF2. However, if there are warning signals from both FF1 and FF2, we might want to tune block A as it involves timing paths to both FF1 and FF2.

4.3 Problem Formulation

Without loss of generality, we consider using canary flip-flop as delay sensors and voltage interpolation as voltage tuning knobs. Each canary flip-flop has a binary output. We overload notation s as both the object of a canary flip-flop and its output variable. If a canary flip-flop $s_i = 0$, the slack at s_i is not too small and no tuning action is needed. Output $s_i = 1$ means a warning signal, which tells that the slack is small and near violation of timing constraint. In voltage interpolation, there are two VDD options for each block and the default is low VDD. We also overload notation b as the object of a block and the binary control variable to its supply voltage, with $b_i = 0$ for low VDD and $b_i = 1$ for high VDD.

The control system takes sensor output vector $\mathbf{s} = [s_k \ \ s_{k-1} \ \ ... \ \ s_1]^T$ as input and generates the control vector $\mathbf{b} = [b_m \ \ b_{m-1} \ \ ... \ \ b_1]^T$ as output. By turning some blocks from $b = 0$ to $b = 1$, the delay for corresponding blocks should be reduced and consequently related sensor outputs may change from $s = 1$ to $s = 0$. The control is to find value for $\mathbf{b}$ such that all sensor warning signals are eliminated in presence of variations. Switching the voltage of a block from low to high causes more power dissipation. When there are multiple $\mathbf{b}$ values that can make $\mathbf{s} = \mathbf{0}$, we need to choose

the one with the minimum power dissipation.

4.4 Rule-Based Control

One approach for ASV control is combinational logic circuit with input $\mathbf{s}$ and output $\mathbf{b}$. A brute force method for designing such circuit is to build its truth table. In order to do so, one needs to find a specific value of $\mathbf{b}$ for each certain value of $\mathbf{s}$ such that the power overhead is minimized while all warning signals are eliminated. The best value of $\mathbf{b}$ can be found by examination (e.g., timing analysis) of all its 2^m combinations in the worst case. This search is repeated for all 2^k sensor output combinations. Such brute force method requires $2^m \cdot 2^k$ runs of timing analysis and is evidently not scalable. For example, a circuit of $m = 10$ and $k = 10$ would need over one million runs of timing analysis.

We suggest a rule-based heuristic where a block is switched to high VDD by the warning from one sensor. In other words, one sensor is assigned to one block. This is a matching problem that can be formulated and solved by network flow model.

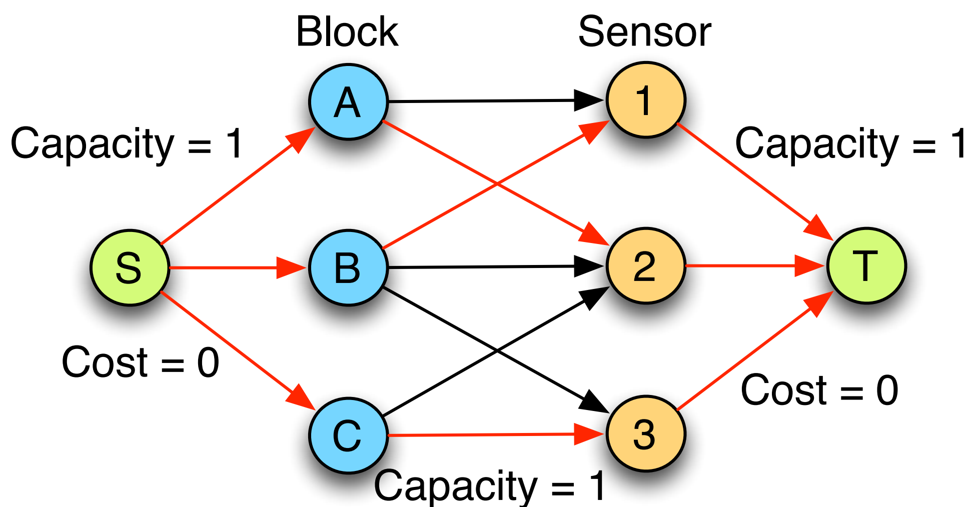


Figure 4.3: Network flow model for assigning sensors to adaptivity blocks. The edge cost between block and sensor vertex is inversely proportional to the overlap between the sensor fanin cone and the block. The red edges indicate a flow solution.

The network is a graph $G = (V, E)$ (see Figure 4.3). The vertices V is composed by block vertices V_b, sensor vertices V_s, the source vertex S and the target vertex T. The set E includes edges E_{Sb} from the source vertex to each block vertex, E_{bs} for edges between V_b and V_s, and E_{sT} for edges from each sensor vertex to the target vertex. Every edge has capacity of one. The cost for each edge in E_{Sb} and E_{st} is zero. The cost for edge $(b, s) \in E_{bs}$ is the inverse of the number of cells that are in both block b and the fanin cone of sensor s, i.e. overlapping cells. If the overlap between b and fanin cone of s is empty, no edge between them exists in the network. Then, we run off-the-shelf min-cost flow algorithm on this network with flow constraint equal to the total capacity of E_{Sb}. The flow solution would match a sensor to a block with large overlap to its fanin cone.

Sometimes the number of blocks $|V_b|$ is different from the number of sensors $|V_s|$. If $|V_b| > |V_s|$, we choose the sensors (canary flip-flops) with the minimum nominal slack and duplicate their vertices. Such sensors usually involve long timing paths, which may need more cells with high VDD. When $|V_b| < |V_s|$, we select some sensors and merge their vertices. For each sensor, one can find the block that has the largest overlap with its fanin cone. Based on the pigeonhole principle, there are at least two sensors s_i and s_j that have the largest overlap with the same block. We take s_i OR s_j as the warning signal in place of the separated signals from s_i and s_j. Then, one can use the merged vertex $s_{i,j} \in V_s$ in the network.

4.5 Finite State Machine Control

The ASV control may have different states, which are represented by different values of $\mathbf{b} = [b_m \ b_{m-1} \ ... \ b_1]^T$. A Finite State Machine (FSM) generates new output $\mathbf{b}_{new}$ according to not only $\mathbf{s}$ but also current output or state $\mathbf{b}_{current}$. As such, FSM can account for the feedback from the effect of current block voltages, and iteratively improve the control result. Obviously, this is a more powerful technique than the rule-based method introduced in section 4.4, which considers only $\mathbf{s}$. The number of states is 2^m, which can be huge for an FSM design. Therefore, we propose a new methodology that is different from conventional FSM designs. It has two phases. Phase I makes the minimum effort in response to warning signals at the initial state, which is $\mathbf{b} = [0 \ 0 \ ... \ 0]^T$.

Phase II makes incremental changes in an effort for fixing all timing warnings.

4.5.1 Phase I: Initial Response

We define that an adaptivity block b_i *flags* a sensor s_j, if the block has large overlap with the fanin cone of s_j, i.e., the value of b_i has significant impact to the output of s_j. For example, in Figure 4.1, block A flags canary FF1. If a block control variable b_i changes from 0 to 1 due to warning signal at s_j, block b_i is said to *respond* to s_j. Setting $b_i = 1$ means the supply voltage of block b_i is the high VDD, which costs more power dissipation than the default low VDD setting.

Problem formulation. *The design of the initial response is to ensure that every sensor with warning signal is responded by at least one of its flagging blocks and the total power overhead due to the responses is minimized.*

We define flag matrix C with rows corresponding to adaptivity blocks and columns corresponding to sensors. A matrix element $c_{ij} = 1$ if block b_i flags sensor s_j. One example of flag matrix is given below.

$$C = \begin{array}{c} \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{array} \begin{array}{c} \begin{array}{cccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{4.1}$$

In this example, block 2 flags sensor 1, 2 and 3. For $s_6 = 1$, either b_1 or b_5 can respond. To reduce power dissipation, the one with less power between b_1 and b_5 is set to high VDD. When $s_4 = 1$, the initial response needs to make $b_1 = 1$. The initial control vector $\mathbf{b}$ generation procedure uses the following two concepts.

Definition 1 *Maximum Single-Response Scenario (MaxSRS)*: A maximum single-response sce-

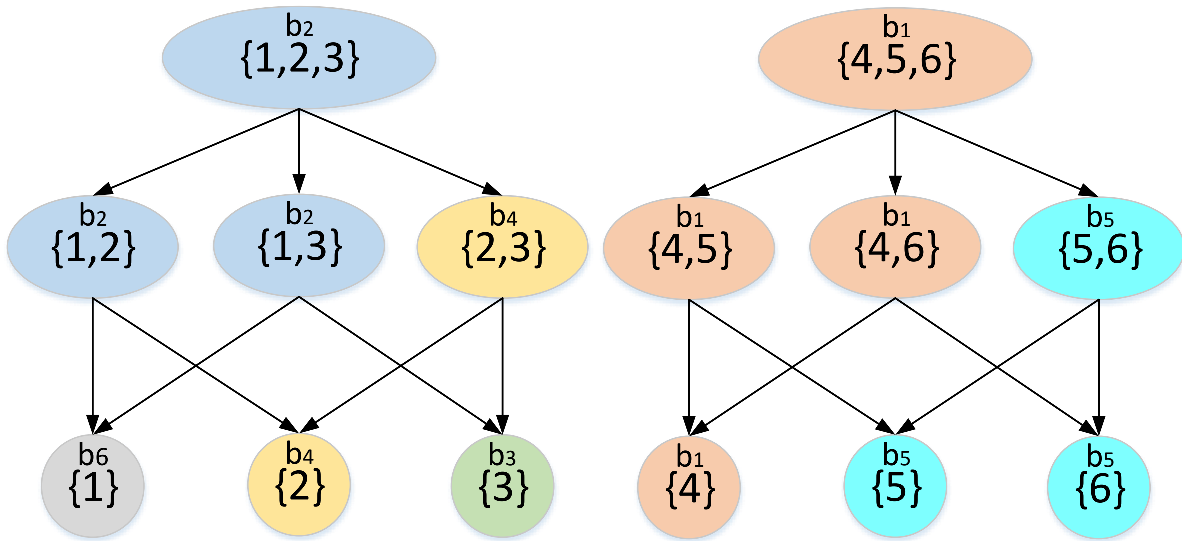
nario Ψ is the maximum set of sensors that can be flagged by only one block. For instance, there are two MaxSRS $\{s_3, s_2, s_1\}$ and $\{s_6, s_5, s_4\}$ in the example of flag matrix (4.1).

Definition 2 *Single-Response Scenario (SRS)*: A single-response scenario ψ is a set of sensors that is an element in the power set $\mathcal{P}(\Psi)$ of a MaxSRS Ψ. Additionally, all sensors in ψ have output 1 and all sensors in $\Psi - \psi$ have output 0. For example, the sensor scenarios for the MaxSRS $\{s_3, s_2, s_1\}$ include $\{s_3, s_2, s_1\}$, $\{s_2, s_1\}$, $\{s_3, s_2\}$, $\{s_3, s_1\}$, $\{s_1\}$, $\{s_2\}$ and $\{s_3\}$. Please note $\{s_3, s_1\}$ implies $s_3 = s_1 = 1$ and $s_2 = 0$.

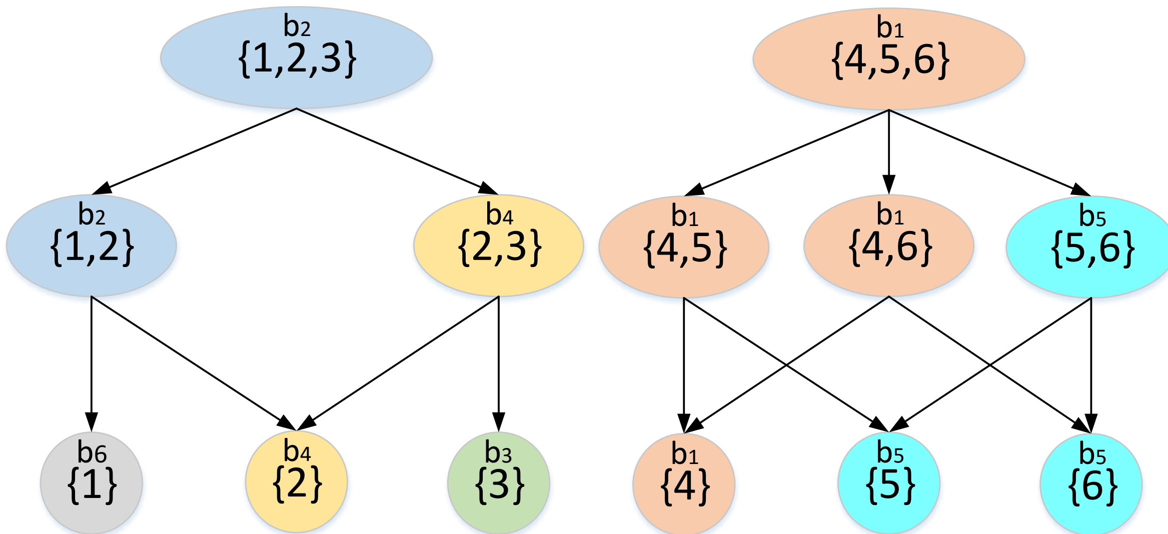
For each SRS ψ_i, we can always find at least one block, called *response block*, to respond to all sensors in ψ_i. If there are multiple such blocks, we choose the one with the minimum power overhead as the response block. If b_i is the response block for SRS ψ, then ψ is called the *stimulus SRS* to b_i. The initial response of $\mathbf{b}$ is largely decided by the response block and stimulus SRS relation.

Single response control generation. Adaptivity block $b_i = 1$ if any of its stimulus SRS asserts. Hence, the value of b_i equals logic OR among all of its stimulus SRS, i.e., sum of minterms that correspond to stimulus SRS. For the example in matrix (4.1), $b_4 = \bar{s}_3 s_2 \bar{s}_1 + s_3 s_2 \bar{s}_1$, if power overhead of b_4 is less than b_2. A minterm or an SRS $\psi \in \mathcal{P}(\Psi)$ is the logic AND operation among the output or output complement of all sensors in Ψ. For example, SRS $\{s_3, s_1\}$ means $s_3 \bar{s}_2 s_1$. Usually, the sum of minterm logic contains redundancy. For example, $b_4 = \bar{s}_3 s_2 \bar{s}_1 + s_3 s_2 \bar{s}_1$ can be simplified to $b_4 = s_2 \bar{s}_1$.

Multi-response control generation. *Multi-response* means that a sensor scenario is responded by multiple blocks. For example, a sensor scenario $\{s_3, s_1\}$ for matrix (4.1) can be responded by single block b_2, or by two blocks $\{b_6, b_3\}$, and it is likely that the later case incurs less power overhead. We develop a graph based algorithm to handle both the multi-response control generation and the logic simplification of control circuits. First, we construct a dominance graph, where each vertex corresponds to one SRS. There is an edge from SRS ψ_i to ψ_j if $\psi_j \subset \psi_i$, i.e., ψ_i dominates ψ_j. The dominance graph for matrix (4.1) is shown in Figure 4.4(a). The response block for each SRS is also labeled for each vertex. The vertices are colored according to their response blocks. The dom-



(a) The dominance graph for control matrix (4.1). Each vertex is labeled with its sensor scenario and response block.



(b) The dominance graph after pruning.

Figure 4.4: Dominance graph for matrix (4.1) and its pruning, on the assumption that the power overhead of $b3$ and $b6$ combined is less than that of $b2$.

inance graph is pruned to facilitate the optimal response with the minimum power overhead. The pruning is a depth-first search of the dominance graph as outlined in Algorithm 4. The key steps are from line 8 to line 17, where a vertex and its associated edges are removed if its corresponding

power overhead is greater than that of its successor vertices. An example of such pruning is shown in Figure 4.4(b).

```
1  PruneGraph(G(V,E))
   /* G is the dominance graph                                          */
2  foreach v ∈ V without predecessor do
3  |   Prune(v)
4  end

1  Prune(v)
2  if v has no successor vertex then
3  |   return
4  end
5  foreach (v, w) ∈ E do
6  |   Prune(w)
7  end
8  if P({v_succ}) < P(v) then
   |   /* P({v_succ}) is the total power overhead of all the unique
   |      blocks associated with successor vertices of v, P(v) is
   |      the power overhead of the block corresponding to
   |      vertex v                                                      */
9  |   foreach (v, w_i) ∈ E do
10 |   |   foreach (p_j, v) ∈ E do
11 |   |   |   if (p_j, v, w_i) is the only path from p_j to w_i then
12 |   |   |   |   E = E ∪ {(p_j, w_i)}
13 |   |   |   end
14 |   |   end
15 |   end
16 |   Remove vertex v, edge (p_j, v) and edge (v, w_i), ∀i, j /* Remove v and its
   |      incident edges                                                */
17 end
```

Algorithm 4: Graph Pruning Algorithm

The optimal multi-response control vector can be directly derived from the pruned dominance graph. Denote $\mathcal{B}_i$ as the set of sensor scenarios that block b_i should respond to, i.e., the vertices in the dominance graph labeled with b_i. For each set of sensor scenarios $\mathcal{B}_i$, there is a unique sensor scenario with the maximum number of sensors, which is defined as the max-scenario for b_i and is

denoted $\mathbf{B}_i^{max}$. Also, $\mathcal{B}_i$ contains a set of scenarios with the minimum number of sensors, which is defined as the min-scenario set of b_i and denoted as $\mathcal{B}_i^{min} \subseteq \mathcal{B}_i$. For the flag matrix (4.1), the min-scenario sets and the max scenario for b_5 and b_4 are:

$$\mathcal{B}_5 = \{\{s_6, s_5\}, \{s_6\}, \{s_5\}\} \qquad \mathcal{B}_4 = \{\{s_2\}, \{s_3, s_2\}\}$$

$$\mathbf{B}_5^{max} = \{s_6, s_5\} \qquad \mathbf{B}_4^{max} = \{s_3, s_2\}$$

$$\mathcal{B}_5^{min} = \{\{s_6\}, \{s_5\}\} \qquad \mathcal{B}_4^{min} = \{\{s_2\}\}$$

The value for b_i can be decided by considering only $\mathcal{B}_i^{min}$ and $\mathbf{B}_i^{max}$. Please note $\mathbf{B}_i^{max}$ must correspond to a vertex in the dominance graph. More precisely speaking, we need to consider the complement of the sensors in the predecessor of $\mathbf{B}_i^{max}$ but not in $\mathbf{B}_i^{max}$ itself. For example, $\mathbf{B}_5^{max} = \{s_6, s_5\}$ and its predecessor vertex corresponds to sensor scenario $\{s_6, s_5, s_4\}$. Then, we include $\bar{s}_4$ in constructing the circuit for b_5. The algorithm for generating multi-response control with logic simplification is provided in Algorithm 5.

1 `GenerateTerm(`*G(V,E)*, b_i`)`
2 Find $\mathbf{B}_i^{max}$ and $\mathcal{B}_i^{min}$
3 $b_i = 0$
4 **foreach** $\mathbf{B}_{ij}^{min} \in \mathcal{B}_i^{min}$ **do**
5 $\quad$ $b_i = b_i +$ m$(\mathbf{B}_{ij}^{min})$
$\quad$ `/* m generates product term for a sensor scenario, e.g.,` `m(`$\{s_2, s_1\}$`) = `$s_2 s_1$ `*/`
6 **end**
7 **foreach** $(p_j, v_i^{max}) \in E$ **do**
$\quad$ `/*` v_i^{max} `corresponds to` $\mathbf{B}_i^{max}$ `*/`
8 $\quad$ $b_i = b_i \cdot \overline{\text{m}(\mathbf{p}_j - \mathbf{B}_i^{max})}$
$\quad$ `/*` $\mathbf{p}_j$ `is the sensor scenario for` p_j`, e.g.,` $\overline{\text{m}(\{s_3, s_2, s_1\} - \{s_3\})} = \overline{s_2 s_1} = \bar{s}_2 + \bar{s}_1$ `*/`
9 **end**
10 **return** b_i

Algorithm 5: Control Generation Algorithm

Using the proposed algorithms, we can generate the control vectors for flag matrix (4.1):

$$b_1 = s_4 \qquad b_2 = s_1 s_2$$

$$b_3 = s_3 \bar{s}_2 \qquad b_4 = s_2 \bar{s}_1$$

$$b_5 = (s_5 + s_6)\bar{s}_4 = s_5\bar{s}_4 + s_6\bar{s}_4 \qquad b_6 = s_1 \bar{s}_2$$

Even for a complex dominance graph, our algorithms can generate simplified logic circuit for the control signal vector $\mathbf{b}$. The control vectors are guaranteed to be optimal in terms of power overhead, with an assumption that for every two max scenarios $\mathbf{B}_i^{max}$ and $\mathbf{B}_j^{max}$, either $\mathbf{B}_i^{max} \subseteq \mathbf{B}_j^{max}$ or $\mathbf{B}_j^{max} \subseteq \mathbf{B}_i^{max}$, i.e., either $\mathbf{B}_j^{max}$ dominates $\mathbf{B}_i^{max}$ or $\mathbf{B}_i^{max}$ dominates $\mathbf{B}_j^{max}$.

4.5.2 Phase II: Incremental Responses for FSM

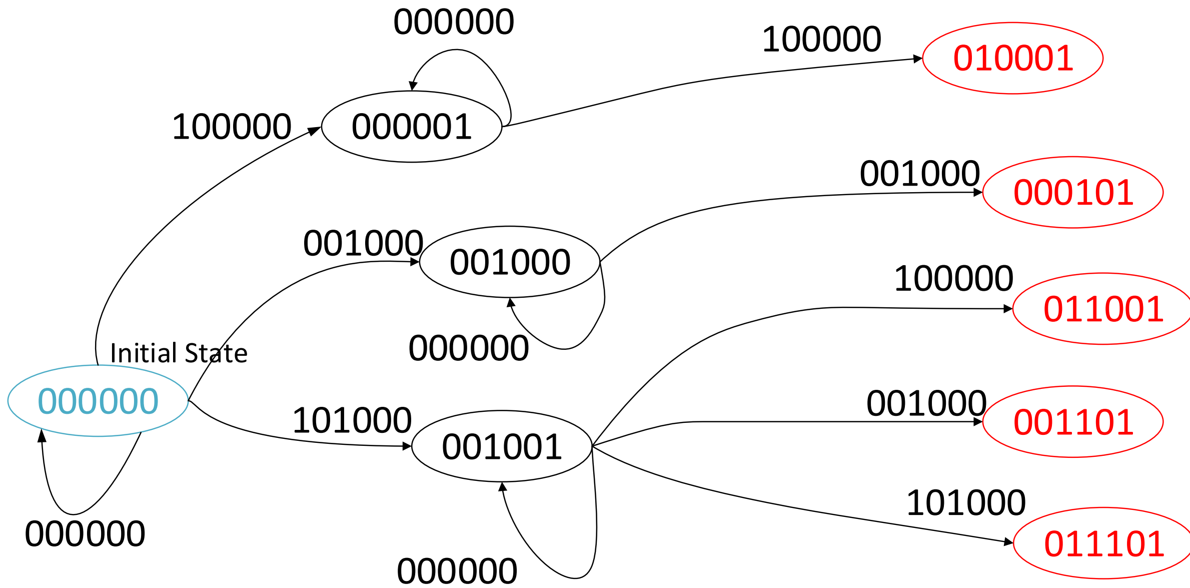


Figure 4.5: Partial FSM transition diagram for control matrix (4.1).

After the initial response, there still might be sensors with warning signals and we need to

switch more blocks to high VDD. The initial response leads the circuit into different states according to different sensor outputs. Ideally, we like to repeat phase I for each of these states. However, such approach would result in overly complex control. Therefore, we suggest greedy incremental changes for the states after the initial response. For each sensor $s_j = 1$, we pick one of its flagging blocks b_i that has value 0 after the initial response and the minimum power overhead, and set $b_i = 1$. This procedure is repeated till all sensor outputs become 0 or all control signals become 1. A part of the FSM design corresponding matrix (4.1) is shown in Figure 4.5, where question marks means either 0 or 1. The complete design is too complicated to be presented in a figure.

4.6 Delay Sensor Deployment

The role of a delay sensor is to monitor timing paths and produces warning signal when a path delay is near timing violation. In general, there are two competing design goals for delay sensor deployment. First, sensors should monitor all paths whose delay variations pose a risk on timing violation, i.e., the monitoring coverage needs to be sufficiently high. Second, the number of sensors needs to be restricted or minimized, as they cause area and power overhead.

We consider fine-grained ASV using canary flip-flops as delay sensors. The work of [96], perhaps the only previous work on canary FF deployment, simply selects flip-flops with slack below certain threshold and converts them into canary FFs. One observation is that there may be large overlap among timing paths to different flip-flops. For instance, in Figure 4.6, the timing paths to FF2 are largely covered by paths to FF1 and FF3. Then, FF2 does not need to be a sensor if both FF1 and FF3 become canary FFs. By incorporating this observation, we propose two problem formulations and corresponding algorithms for the delay sensor (canary FF) deployment.

Before presenting the formulations and algorithms, we first define some terms. A *timing critical path* is a path that has the maximum delay to a flip-flop and its slack is below a threshold τ. This definition is slightly different from the conventional definition for the convenience of presenting the algorithms. A logic gate or a part of a path is called *covered* if it is within the fanin cone of any canary FF.

Delay Sensor Deployment Formulation 1: *Given a circuit, find a subset of flip-flops to be con-*

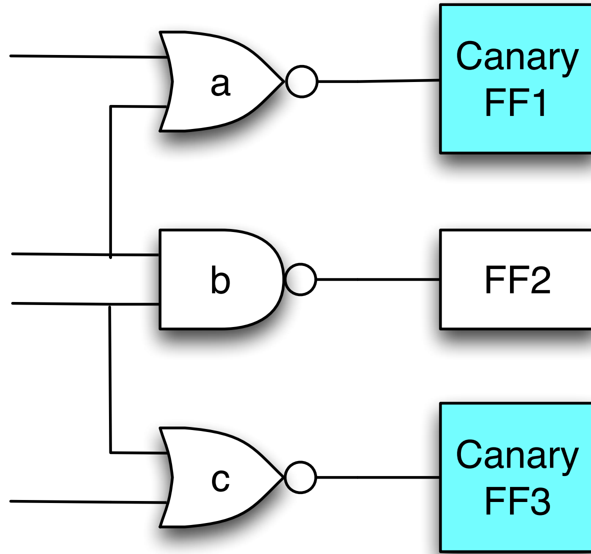


Figure 4.6: The paths to FF2 are largely covered by paths to FF1 and FF3.

verted to canary FFs, such that the coverage to the timing critical paths is maximized while the number of canary FFs is no greater than a given budget.

The algorithm for solving this formulation is an iterative greedy heuristic. In each iteration, one flip-flop is selected to be canary FF according to the following preference metric.

$$p_i = (d_i - \omega) \cdot \frac{\bar{d}_i}{d_i} \tag{4.2}$$

where d_i is the longest path delay to flip-flop i, $\bar{d}_i$ is the delay of uncovered portion of this path, and ω is a weighting factor. The first term indicates the timing criticality for a flip-flop and the second term favors a flip-flop whose critical path has not been covered much. The weighting factor ω provides a tradeoff between the two terms. A greater value of ω emphasizes more on the timing criticality than the coverage. If $\omega = 0$, the preference is solely based on coverage. The tradeoff is illustrated by an example in Figure 4.7. This example has three flip-flops, with delay vector $(2.9, 3, 3.1)$ and uncovered delay vector $(1.8, 1.5, 1.2)$. The first flip-flop has the smallest delay but the largest uncovered delay. By increasing ω from 2.4 to 2.7, the preference changes from flip-flop

1 to flip-flop 3. The algorithm iteratively chooses the flip-flop with the largest value of p to be canary FF till the constraint on the number of canary FFs is reached.

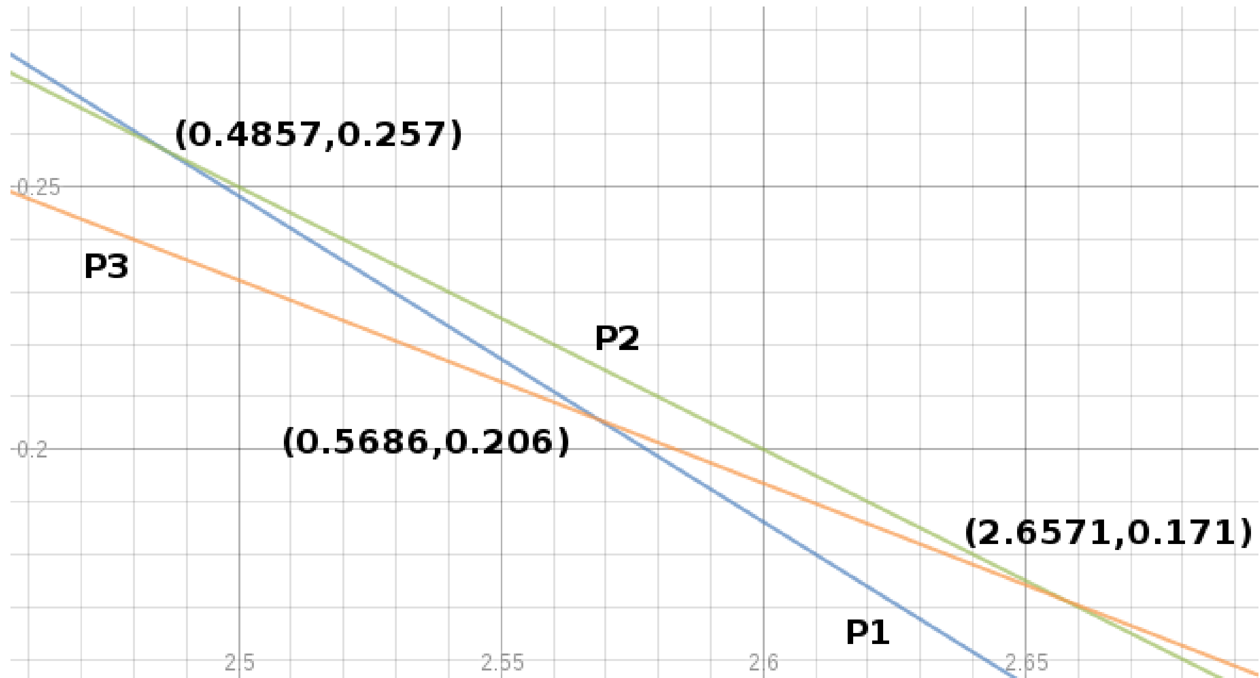


Figure 4.7: Impact of ω on the preference metric with horizontal axis for ω and vertical axis for p. If $\omega > 2.6571$, $p_3 > p_2 > p_1$; if $\omega < 2.4857$, $p_1 > p_2 > p_3$.

Delay Sensor Deployment Formulation 2: *Given a circuit, find a subset of flip-flops to be converted to canary FFs, such that the number of canary FFs is minimized while the coverage to all timing critical paths is no less than a given constraint.*

The second formulation is very similar as the set cover problem. Given a universe $\mathcal{U} = \{1, 2, 3, ..., m\}$ with m elements, a family of element sets $\mathcal{S} = \{S_1, S_2, ...\}$, where $S_i \subset \mathcal{U}$, the set cover problem is to find a subset $\mathcal{C} \subseteq \mathcal{S}$ such that all elements in $\mathcal{U}$ are contained in $\mathcal{C}$ and the cardinality $|\mathcal{C}|$ is minimized. By treating timing critical paths as the elements and flip-flops as $\mathcal{S}$, the canary FF deployment problem can be mapped to the set cover problem with a minor change.

We define that a path i is *covered* if the ratio $\bar{d}_i/d_i$ is less than a parameter ρ. This definition is in accordance with the observation in Figure 4.6 and problem formulation 2. The set cover problem

can be solved by integer linear programming, Boolean satisfiability or greedy heuristic. We adopt a greedy heuristic where a flip-flop with the largest uncovered critical path is iteratively selected till all timing critical paths are covered.

4.7 Experiments

4.7.1 Adaptive Design Flow

The experiments are performed on ICCAD 2014 Incremental Timing Driven Placement Contest benchmark circuits, whose gate counts are provided in Table 4.2. The flow of experiments is shown in figure 4.8. We use Cadence SoC Encounter to insert buffers to the circuit layout. According to the placement results, we cluster cells with similar timing criticalities and spatially close to each other, to form cell blocks using the method proposed in [33]. We also used the method in [97] to get the timing analysis results for clustering. The timing critical blocks are designated as adaptivity blocks. Then, another phase of more accurate timing analysis for delay sensor deployment is conducted using Synopsys PrimeTime. The cell timing and power information are obtained from the Nangate 45nm Open Cell Library [98]. The delay sensor design by [27] is adopted and the sensor deployment is conducted using our method of formulation 1 in Section 4.6. Then, the updated design is placed and routed by Cadence SoC Encounter again. One example of the layout is shown in figure 4.9. Like in [5], the high VDD and low VDD in the voltage interpolation are set to 1.25V and 1.1V, respectively. The process variation and aging variation models are the same as [91]. We consider gate length variation with σ being 5% of nominal value, and threshold voltage variation with σ being 7% of nominal value. The spatial correlation among gate length variations is included. The aging effect is captured as additional threshold voltage variation, with mean of 10% and σ of 3% of nominal value. Based on these models and SPICE simulations, cell delay and power variations are obtained. Timing yield of a circuit is estimated through 5K-run Monte Carlo simulation of timing analysis.

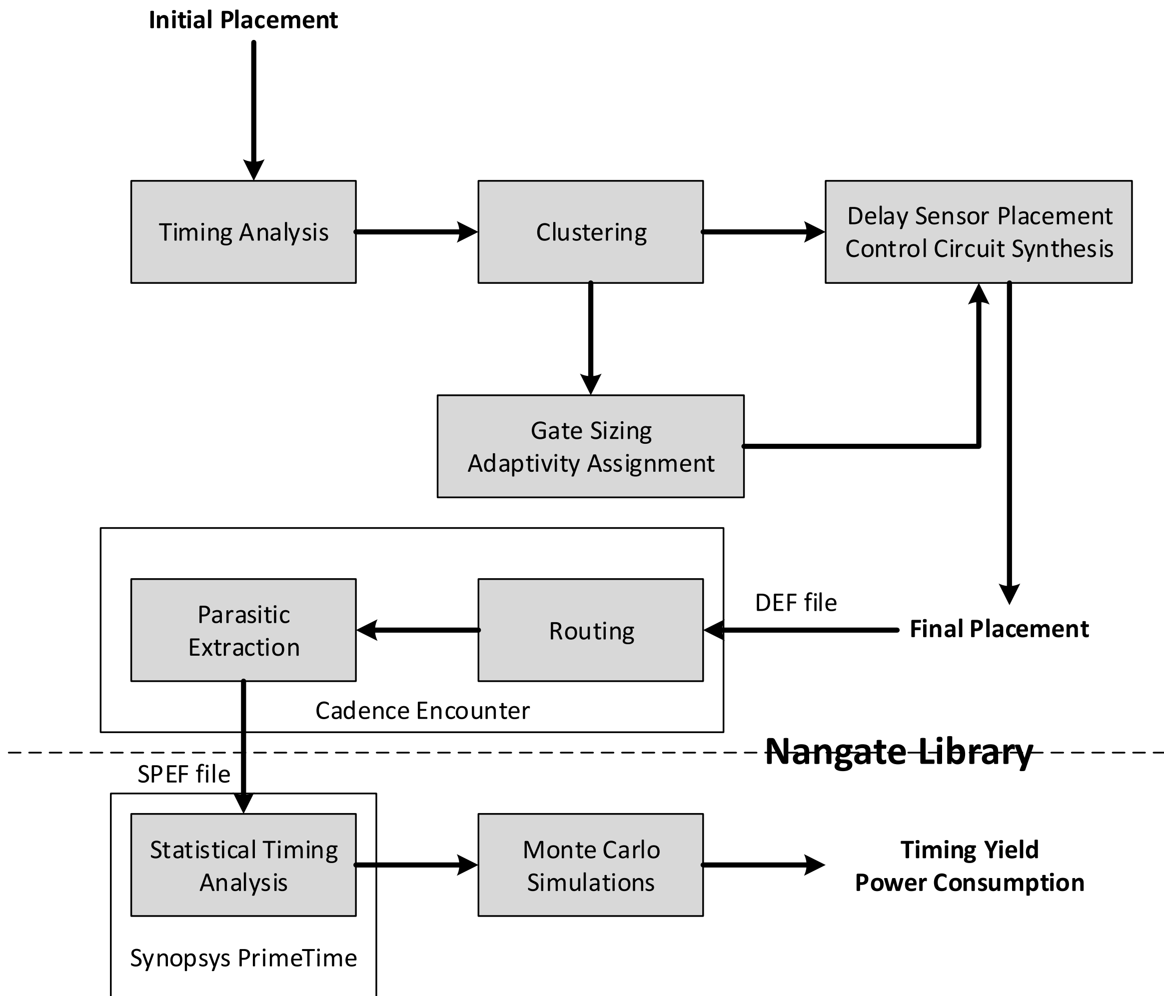
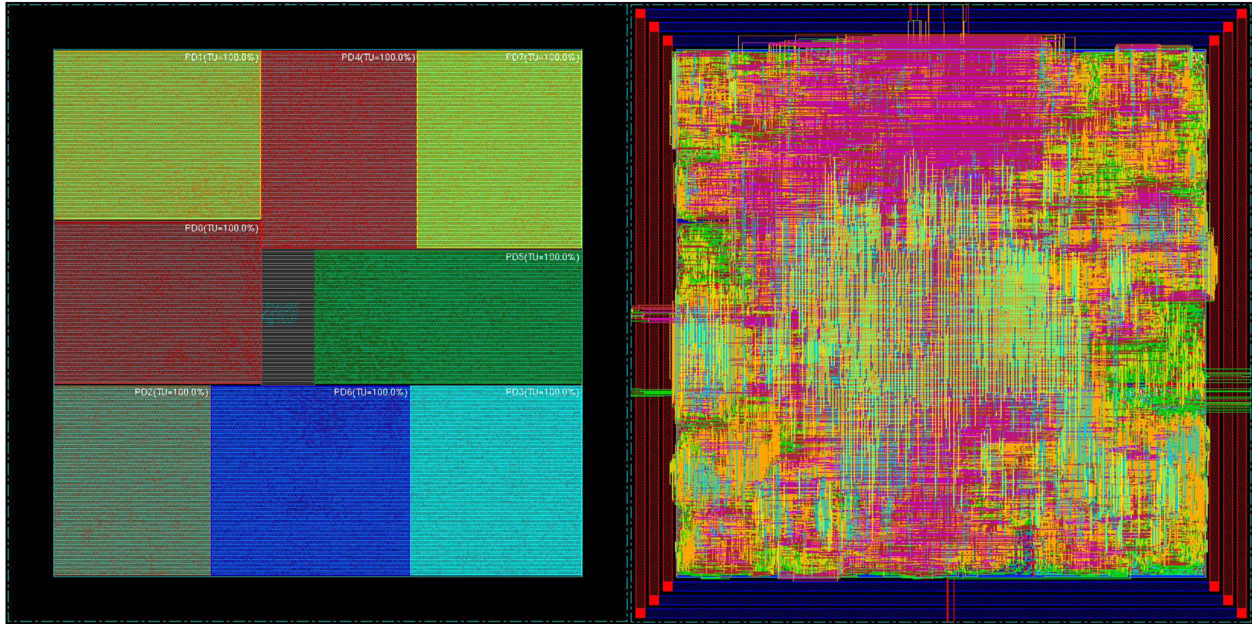


Figure 4.8: The flow of experiments

		Core area(μm^2)	Normalized	Wire length (μm)	Normalized
b19	Adaptive design	342912	1.0306	2624987	1.1727
	Conventional design	332733	1	2238409	1
mgc_edit_dist	Adaptive design	756900	1.0476	4785085	0.8858
	Conventional design	722500	1	5401957	1

Table 4.1: Area and wire length comparison of two cases for conventional design and adaptive design



(a) The layout after placement with power domains (b) The layout of the same design after routing

Figure 4.9: The layout of testcase b19: The power domains are divided according to the gate clustering results

4.7.2 Experimental Results

As there is no previous work on fine-grained ASV control, to the best of our knowledge, we compare the following:

Baseline: The entire circuit has static high VDD.

Coarse: All adaptivity cells are merged into one block, which can be set to high VDD by warning from any sensor.

Rule: Our rule-based control.

FSM: Our finite state machine control.

All-low: The entire circuit has static low VDD.

The main results are shown in Figure 4.10 and Table 4.2. Figure 4.10(a) compares the timing yields from different methods. The timing yields from the **baseline** (all-static-high VDD) are very

Testcases	#gates	Adp blk		Rule		FSM	
		#	%	%A	#H	%A	#H
edit_dist	130661	13	94	0.07	3.36	0.25	1.67
matrix_mult	155325	6	61	0.07	2.02	1.28	1.22
vga_lcd	164891	12	69	0.04	1.23	0.17	0.72
b19	219263	10	77	0.04	0.94	0.15	0.77
leon3mp	649191	9	41	0.01	1.77	0.15	1.01
leon2	794286	16	52	0.01	2.27	0.12	1.17
netcard	958780	17	69	0.01	2.41	0.16	1.26
Average				0.04	2.00	0.33	1.12

Table 4.2: Testcases; total number of gates; number of adaptivity blocks; percentage of gates in adaptivity blocks; area overhead due to adaptivity (%A) and average number of blocks at high VDD (#H) for rule-based and FSM control.

near to 1 for all cases. The average timing yields for **coarse**, **rule** and **FSM** are 99.5%, 98.9% and 98.7%, respectively. The timing yield from **all-low** is much lower. The leakage power data in Figure 4.10(b) are normalized with respect to the results from the **baseline**. Only leakage power is investigated because it is quite significant in nanometer technology, correlates with dynamic power and does not rely on switching activities. Both our **rule** and **FSM** methods cause significantly less leakage than **coarse** with very similar timing yields. Our **FSM** is slightly better than **rule** on leakage and reaches about the same yield. Compared to **coarse**, the **FSM** causes 20% less leakage power on average. Overall, **FSM** achieves timing yield near **baseline** and leakage power not far from **all-low**.

Table 4.2 lists circuit statistics and additional results. The 4th column displays the percentage of gates in adaptivity blocks. Columns 5 and 7 show the percentage adaptivity area overhead mainly including the area increase from canary flip-flops and the control circuits. Columns 6 and 8 are the average number of blocks being switched to high VDD obtained from the Monte Carlo simulation.

Figure 4.11 are the timing yield vs. leakage plots for circuit b19 with various timing constraints. **Coarse** tends to reach high yield solutions with relatively large leakage power. **FSM** is superior to **rule** in maintaining high yield for different timing constraints. Overall, **FSM** has the high timing

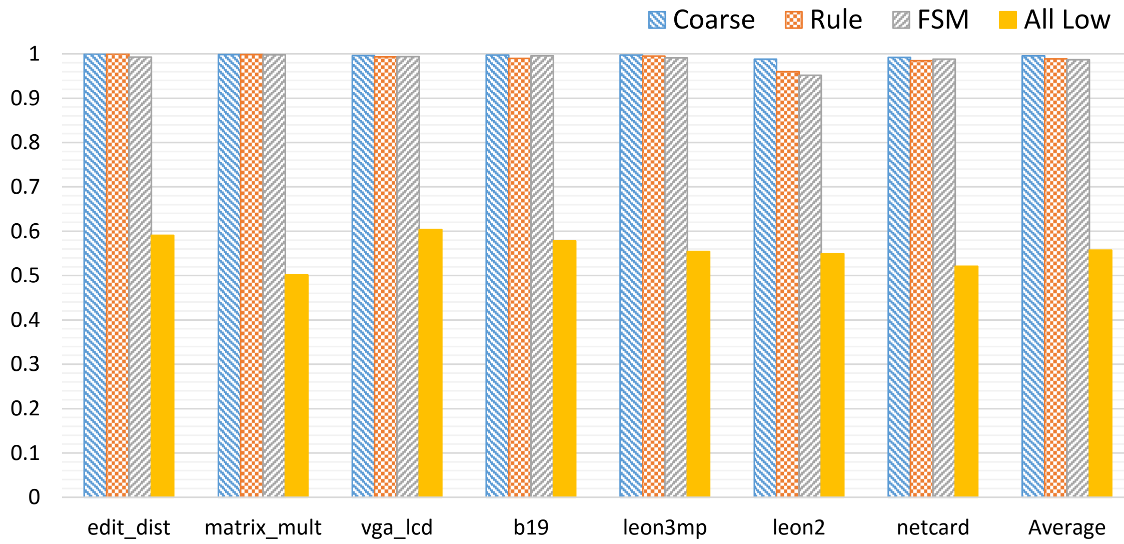
yield as **coarse** and the low leakage power as **rule**.

The impact of the number of sensors is investigated for circuit b19 and the results are shown in Figure 4.13a. For **coarse**, both timing yield and leakage power increase with sensor count at the beginning and then get saturated. The leakage power from **rule** is not sensitive to the number of sensors. The yield of **rule** peaks when the number of sensors is close to the number of adaptivity blocks, which is 10. In other words, the network flow method does not work well when there is significant mismatch between the numbers of sensors and blocks. **FSM** has the yield close to **coarse** and leakage power almost the same as **rule** over different number of sensors. Thus, **FSM** is the best among the three methods.

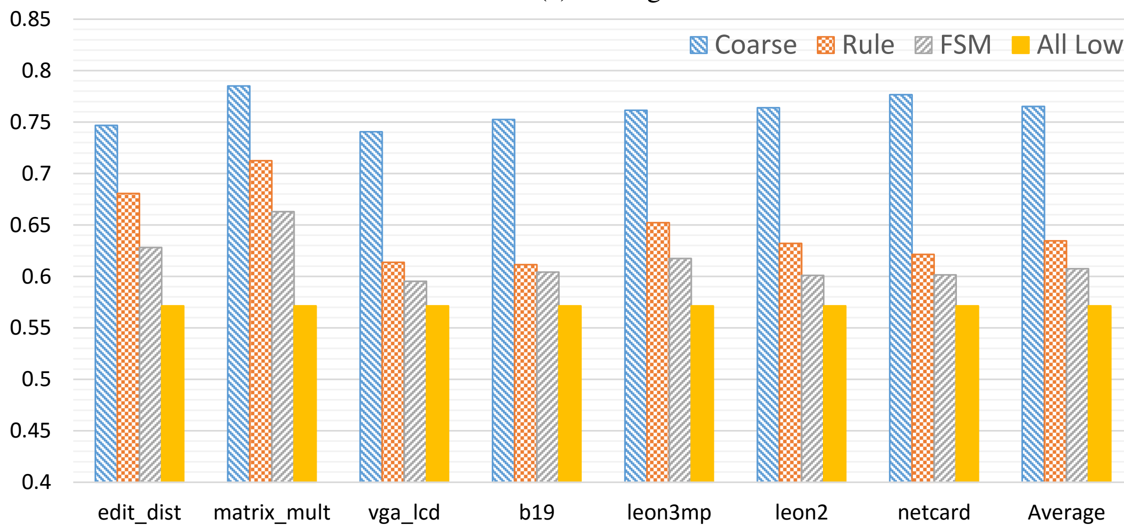
In Figure 4.12, we compare our sensor deployment methods (formulation 1 and 2 in Section 4.6) with the previous work [96]. A good method should provide high coverage on critical paths with small number of sensors. Both our methods are superior to [96] in the coverage vs. sensor count tradeoff. Our formulation 1 is the best, and is utilized in all the other experiments.

4.8 Conclusions

ASV is an effective approach to power-efficient resilience against process variations and circuit aging in nanometer VLSI designs. We propose two techniques for fine-grained ASV control, which has not been well studied before. Our FSM control can achieve about the same timing yield as coarse-grained ASV, but costs 20% less leakage power. In fact, the leakage from our FSM control is not far from using static low VDD for the entire circuit. We also show new techniques of delay sensor deployment, which are demonstrated to be superior to the only previous work. All these proposed techniques can be fully automated.



(a) Timing Yield



(b) Leakage power of adaptive blocks normalized to baseline

Figure 4.10: Results for ICCAD 2014 benchmarks.

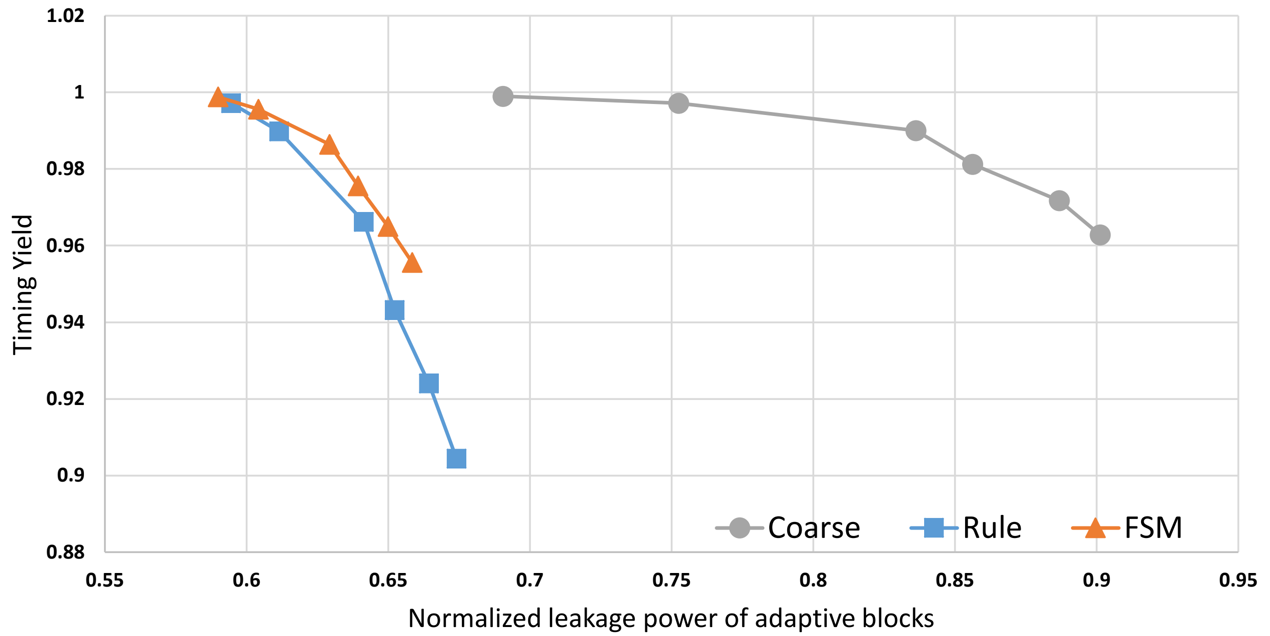


Figure 4.11: Timing yield vs. leakage power for circuit b19 with different timing constraints. For each curve, results on the right are from tighter timing constraints.

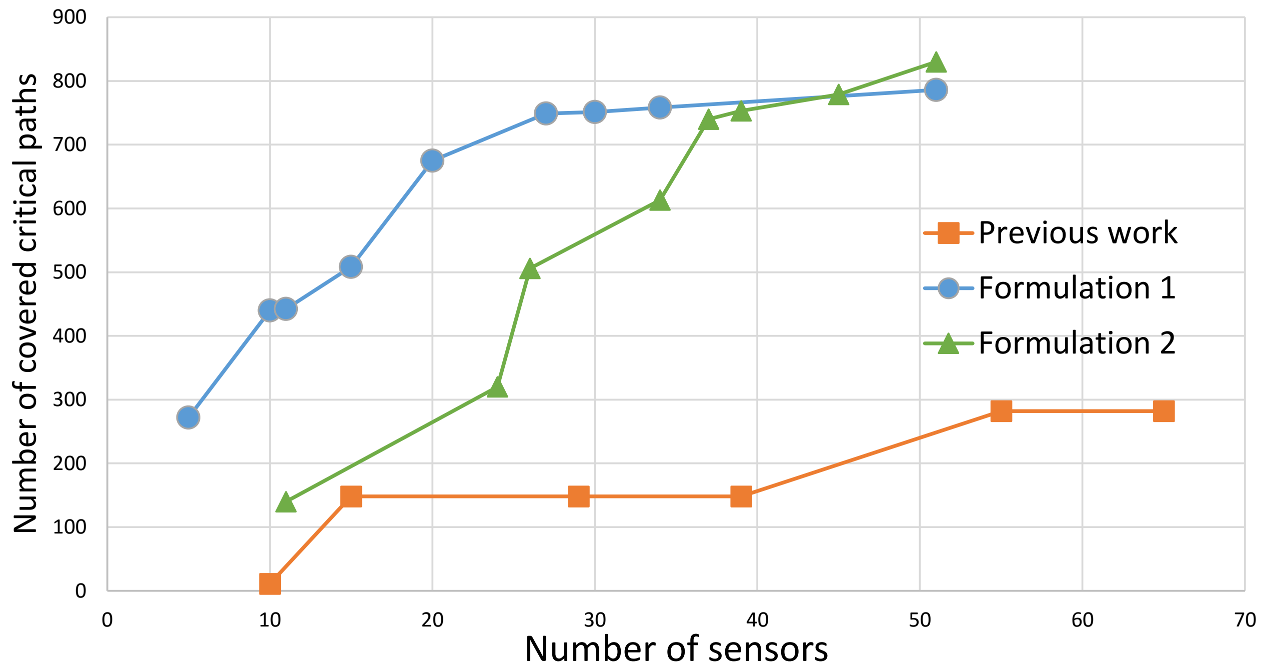
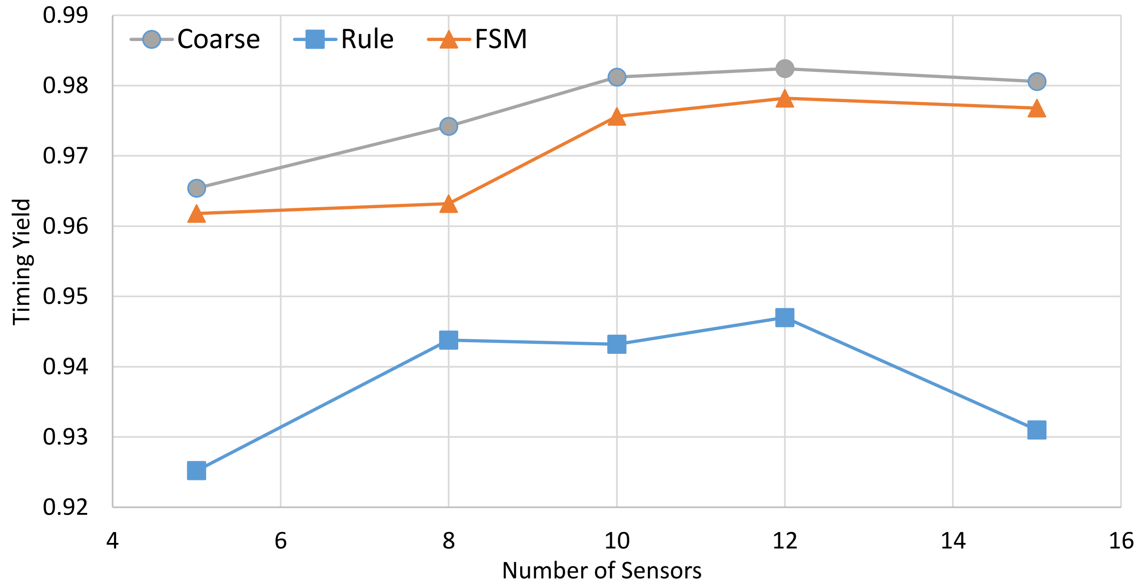
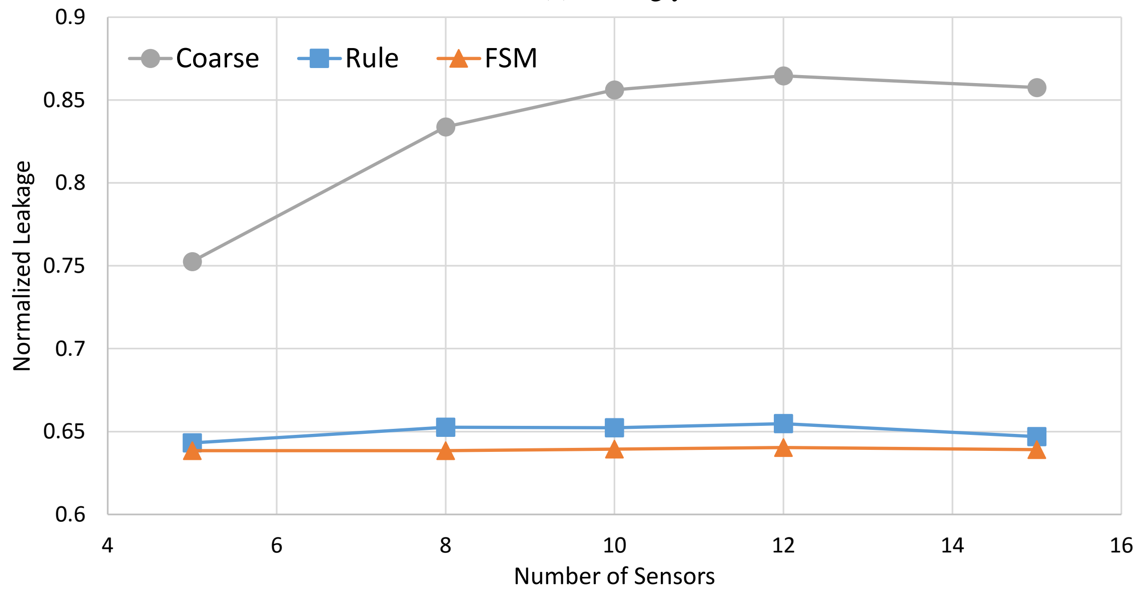


Figure 4.12: Comparison among different sensor deployment methods on b19.



(a) Timing yield



(b) Normalized leakage power of adaptive blocks

Figure 4.13: Impact of the number of sensors on b19.

5. CONCLUSION

This research addresses several issues in synthesis especially high-level synthesis. The methods and techniques achieve better power-efficiency. In addition to that, the high-level synthesis techniques for hardware such as FPGAs achieve significant runtime speedup.

With the slowing down of Moore's law, it relies increasingly on innovations of architectures and designs to improve the power-efficiency. These emerging architectures brought new challenges for synthesis techniques. To take advantage of these new designs and architectures, the synthesis tool needs to consider more metrics such as the error rate and magnitude for joint HLS, approximate computing and synthesis runtime of HLS.

In this research, A novel error model is proposed to measure the error incurred by approximate computing and bitwidth optimization. In a task graph (modeled by a directed acyclic graph), the error caused by internal operations may cancel in later nodes. Moreover, the existing error metrics are difficult to integrate with the integer-linear programming model for high-level synthesis. The error metric should be able to predict the accumulated error at the primary outputs according to the error in the internal nodes. The error variance is just such an error model. If the errors caused by individual operations are independent with each other, the variance can be accumulated to get the error variance at the primary output. Based on this model, two HLS flow are designed. One is ILP based simultaneous precision optimization and high-level synthesis and the other is Knapsack-based precision optimization followed by list scheduling. Our results show that the Knapsack-based method is one order of magnitude faster than the ILP based method. However, the ILP method could get the optimal results subject to the constraints.

A mapping-based HLS flow is developed to map the LLVM IR to pipelined circuits in hardware. Hardware such as FPGAs have special architectures that can be reprogrammed. They also have plenty of registers and distributed memories available once manufactured. These resources are suitable to perform high-throughput computing with pipelining and parallelizing. This research designs a HLS technique that targets pipelined circuits first. Partially pipelined or non-pipelined

circuits then can be simplified from the pipelined circuits later by an optional phase of resource optimization. Pipeline interlocking to address pipeline hazards is also provided, that can achieve better power-efficiency and allows more flexible input patterns. This mapping-base HLS provides better controllability and an order of magnitude faster speed compared to a commercial tool.

We also develop synthesis techniques to automatically construct the control circuits for fine-grained adaptive supply voltage systems. Most previous supply voltage adjustment techniques are coarse-grained. This research proposes fine-grained voltage adjustment techniques following gate clustering techniques and statistical static timing analysis techniques for adaptive supply voltage. The gate clustering is able to group gates into several clusters and these clusters are managed by different power domains. We use the voltage interpolation technique to provide two levels of voltage to each power domain. Then, the entire circuit is operating at an effective supply voltage depending on the voltage levels of every power domains. A delay sensor placement technique is also addressed with balanced gate coverage and timing criticality.

REFERENCES

[1] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2015.

[2] C. Li, S. S. Sapatnekar, and J. Hu, "Control synthesis and delay sensor deployment for efficient asv designs," in *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 64, ACM, 2016.

[3] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pp. 409–414, IEEE Press, 2011.

[4] K. Chapman, "Saving costs with the SRL16E," *Xilinx techXclusive*, p. 6, 2008.

[5] X. Liang, G.-Y. Wei, and D. Brooks, "Revival: A variation-tolerant architecture using voltage interpolation and variable latency," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pp. 191–202, IEEE, 2008.

[6] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU Gems*, 2007.

[7] K. E. Batcher, "Sorting networks and their applications," in *spring joint computer conference*, pp. 307–314, ACM, 1968.

[8] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE TCAD*, vol. 10, no. 4, pp. 464–475, 1991.

[9] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[10] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, IEEE, 2004.

[11] "Bsv high-level hdl: Best-in-class, general purpose high-level synthesis (hls) tools." `http://bluespec.com/54621-2/`, 2018.

[12] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards, "Hardware synthesis from a recursive functional language," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*, pp. 83–93, IEEE, 2015.

[13] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 330–336, IEEE Computer Society Press, 1988.

[14] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Design Automation Conference, 2006 43rd ACM/IEEE*, pp. 433–438, IEEE, 2006.

[15] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 461–466, IEEE, 2003.

[16] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *SOC Conference, 2006 IEEE International*, pp. 199–202, IEEE, 2006.

[17] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[18] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, ACM, 2011.

[19] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pp. 127–134, IEEE, 2010.

[20] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[21] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–4, IEEE, 2014.

[22] W. Xu, S. S. Sapatnekar, and J. Hu, "A simple yet efficient accuracy configurable adder design," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pp. 1–6, IEEE, 2017.

[23] M. B. Taylor, "Bitcoin and the age of bespoke silicon," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, p. 16, IEEE Press, 2013.

[24] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.

[25] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[26] G. E. Blelloch and B. M. Maggs, "Parallel algorithms." `https://www.cs.cmu.edu/~guyb/papers/BM04.pdf`, 2004.

[27] K. Agarwal and S. Nassif, "Characterizing process variation in nanometer cmos," in *Proceedings of the 44th annual Design Automation Conference*, pp. 396–399, ACM, 2007.

[28] J. Keane and C. H. Kim, "Transistor aging," *IEEE Spectrum*, vol. 48, no. 5, pp. 28–33, 2011.

[29] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, pp. 1–8, 2010.

[30] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, *et al.*, “Razor: A low-power pipeline based on circuit-level timing speculation,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Computer Society, 2003.

[31] J. W. Tschanz, J. T. Kao, S. G. Narendra, R. Nair, D. A. Antoniadis, A. P. Chandrakasan, and V. De, “Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage,” *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1396–1402, 2002.

[32] Y. Shen and J. Hu, “Gpu acceleration for pca-based statistical static timing analysis,” in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pp. 674–679, IEEE, 2015.

[33] A. Lu, H. He, and J. Hu, “Proximity optimization for adaptive circuit design,” in *Proceedings of the 2016 on International Symposium on Physical Design*, pp. 91–97, ACM, 2016.

[34] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, “Modeling and synthesis of quality-energy optimal approximate adders,” in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pp. 728–735, IEEE, 2012.

[35] K. Nepal, Y. Li, R. Bahar, and S. Reda, “Abacus: A technique for automated behavioral synthesis of approximate computing circuits,” in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 361, European Design and Automation Association, 2014.

[36] J. Miao, A. Gerstlauer, and M. Orshansky, “Approximate logic synthesis under general error magnitude and frequency constraints,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 779–786, IEEE Press, 2013.

[37] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar, “Optimal design of jpeg hardware under the approximate computing paradigm,” in *Proceedings of the 53rd Annual Design Automation Conference*, p. 106, ACM, 2016.

[38] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[39] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pp. 667–673, IEEE, 2011.

[40] J. Huang, J. Lach, and G. Robins, "A methodology for energy-quality tradeoff using imprecise hardware," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 504–509, ACM, 2012.

[41] W.-T. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Statistical analysis and modeling for error composition in approximate computation circuits," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 47–53, IEEE, 2013.

[42] W. Xu, S. S. Sapatnekar, and J. Hu, "A simple yet efficient accuracy-configurable adder design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[43] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1760–1771, 2013.

[44] D.-U. Lee, A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006.

[45] S. Lee and A. Gerstlauer, "Fine grain word length optimization for dynamic precision scaling in dsp systems," in *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*, pp. 266–271, IEEE, 2013.

[46] K.-I. Kum and W. Sung, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, 2001.

[47] J. Cong, Y. Fan, G. Han, Y. Lin, J. Xu, Z. Zhang, and X. Cheng, "Bitwidth-aware scheduling and binding in high-level synthesis," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 2, pp. 856–861, IEEE, 2005.

[48] "Nangate freepdk15 generic open cell library." `http://www.nangate.com/?page_id=64&id=2481`, 2014.

[49] "Synopsys VCS verilog compiled simulator." `https://www.synopsys.com/verification/simulation/vcs.html`, 2014.

[50] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 330–335, IEEE, 1997.

[51] IBM, "Cplex optimizer." `https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer`.

[52] A. Putnam and et al, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.

[53] "Xilinx vivado high-level synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[54] C. Pilato and F. Ferrandi, "Bambu: A free framework for the high level synthesis of complex applications," *University Booth of DATE*, vol. 29, p. 2011, 2012.

[55] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*, pp. 129–146, Springer, 2008.

[56] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ACM SIGPLAN Notices*, vol. 34, pp. 174–184, ACM, 1998.

[57] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the*

ACM/SIGDA international symposium on Field programmable gate arrays, pp. 9–18, ACM, 2013.

[58] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," *ASPLOS*, pp. 651–665, 2016.

[59] "Synopsys synphony c compiler." `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html`.

[60] "Cadence stratus high-level synthesis." `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`.

[61] R. S. Nikhil and K. R. Czeck, "BSV by example," 2010.

[62] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–7, IEEE, 2013.

[63] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on fpgas using high level synthesis," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 456–463, IEEE, 2014.

[64] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for fpgas," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pp. 529–534, IEEE Computer Society, 2007.

[65] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 3, p. 29, 2012.

[66] J. Keinert, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, *et al.*, "Systemcodesigneran automatic esl synthesis approach by design space exploration and be-

havioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 1, 2009.

[67] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), pp. 165–176, ACM, 2002.

[68] S. Dai, G. Liu, and Z. Zhang, "A scalable approach to exact resource-constrained scheduling based on a joint sdc and sat formulation," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 137–146, ACM, 2018.

[69] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pp. 3–12, IEEE, 2002.

[70] *7 Series FPGAs Configurable Logic Block User Guide*. Xilinx Inc, Sept 2016.

[71] Y. Ben-Asher and N. Rotem, "Automatic memory partitioning: increasing memory parallelism via data structure partitioning," *CODES+ISSS*, pp. 155–162, 2010.

[72] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An integrated and automated memory optimization flow for FPGA behavioral synthesis," *ASP-DAC*, pp. 257–262, 2012.

[73] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," *DAC*, p. 12, 2013.

[74] A. Abdelhadi, *Architecture of block-RAM-based massively parallel memory structures: multi-ported memories and content-addressable memories*. PhD thesis, UBC, 2016.

[75] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM TODAES*, vol. 13, no. 4, p. 61, 2008.

[76] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," *POPL*, 1988.

[77] C. Lattner and V. Adve, "LLVM language reference manual." http://llvm.org/docs/LangRef.html#abstract.

[78] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, 1991.

[79] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," *POPL*, pp. 107–120, 1998.

[80] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE TCAD*, vol. 30, no. 4, pp. 473–491, 2011.

[81] "LegUp documentation 4.0," 2015.

[82] G. E. Blelloch and B. M. Maggs, "Parallel algorithms." https://www.cs.cmu.edu/~guyb/papers/BM04.pdf, 1996.

[83] D. B. Thomas, "Synthesisable recursion for C++ hls tools," in *IEEE ASAP*, 2016.

[84] G. E. Blelloch, "Prefix sums and their applications." https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf, 1990.

[85] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," *FPL*, pp. 531–534, 2012.

[86] R. E. Pattis, "Functions as data: map, filter, reduce." https://www.ics.uci.edu/~pattis/ICS-31/lectures/functionsasdata/functionsasdata.txt, 2013.

[87] S. Sarma, N. Dutt, P. Gupta, A. Nicolau, and N. Venkatasubramanian, "On-chip self-awareness using cyberphysical-systems-on-chip (cpsoc)," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, p. 22, ACM, 2014.

[88] J. W. Tschanz, S. Narendra, R. Nair, and V. De, "Effectiveness of adaptive supply voltage and body bias for reducing impact of parameter variations in low power and high performance microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 5, pp. 826–829, 2003.

[89] T. Chen and S. Naffziger, "Comparison of adaptive body bias (abb) and adaptive supply voltage (asv) for improving delay and leakage under the presence of process variation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 5, pp. 888–899, 2003.

[90] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, J. B. Carter, and R. W. Berry, "Active guardband management in power7+ to save energy and maintain reliability," *IEEE Micro*, vol. 33, no. 4, pp. 35–45, 2013.

[91] K.-N. Shim, J. Hu, and J. Silva-Martinez, "Dual-level adaptive supply voltage system for variation resilience," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1041–1052, 2013.

[92] T. Sato and Y. Kunitake, "A simple flip-flop circuit for typical-case designs for dfm," in *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*, pp. 539–544, IEEE, 2007.

[93] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *VLSI Test Symposium, 2007. 25th IEEE*, pp. 277–286, IEEE, 2007.

[94] Q. Liu and S. S. Sapatnekar, "Capturing post-silicon variations using a representative critical path," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 2, pp. 211–222, 2010.

[95] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 123–134, IEEE, 2008.

[96] Y. Kunitake, T. Sato, and H. Yasuura, "A replacement strategy for canary flip-flops," in *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pp. 227–228, IEEE, 2010.

[97] R. Kumar, B. Li, Y. Shen, U. Schlichtmann, and J. Hu, "Timing verification for adaptive integrated circuits," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 1587–1590, IEEE, 2015.

[98] "Nangate freepdk45 generic open cell library," 2011.

APPENDIX A

TRANSISTOR-LEVEL HDL FOR AN APPROXIMATE ADDER

```
// Library - Design, Cell - MA_appr4_Nan, View - schematic
// LAST TIME SAVED: Nov 14 20:30:24 2014
// NETLIST TIME: Nov 14 21:02:11 2014
`ifndef __MA_APPR4_NAN_V__
`define __MA_APPR4_NAN_V__

`timescale 1ns / 1ns

module MA_appr4_Nan ( Cout, SUM, GND, Vdd, A, B, Cin );

output  Cout, SUM;

inout  GND, Vdd;

input  A, B, Cin;


specify
specparam CDS_LIBNAME  = "Design";
specparam CDS_CELLNAME = "MA_appr4_Nan";
specparam CDS_VIEWNAME = "schematic";
endspecify

pmos  M14 ( SUM, Vdd, cdsNet0);
pmos  M11 ( Cout, Vdd, cdsNet1);
pmos  M0 ( cdsNet1, Vdd, A);
pmos  M1 ( net57, Vdd, A);
pmos  M2 ( net57, Vdd, B);
```

```
pmos  M3 ( cdsNet0, net57, cdsNet1);
pmos  M4 ( cdsNet0, Vdd, Cin);
nmos  M12 ( Cout, GND, cdsNet1);
nmos  M13 ( SUM, GND, cdsNet0);
nmos  M5 ( cdsNet1, GND, A);
nmos  M6 ( cdsNet0, net72, cdsNet1);
nmos  M7 ( net72, GND, Cin);
nmos  M8 ( cdsNet0, net73, Cin);
nmos  M9 ( net73, net74, B);
nmos  M10 ( net74, GND, A);

endmodule

`endif
```

APPENDIX B

TRANSISTOR-LEVEL HDL FOR AN ACCURATE ADDER

```
`ifndef __FA_X1_V__
`define __FA_X1_V__

`timescale 1ns / 1ns

module FA_X1 ( A, B, Cin, Cout, SUM );

supply1 Vdd;
supply0 GND;

output  Cout, SUM;

//inout  GND, Vdd;

input  A, B, Cin;


specify
specparam CDS_LIBNAME  = "Design";
specparam CDS_CELLNAME = "FA_X1";
specparam CDS_VIEWNAME = "schematic";
endspecify

pmos  M27 ( Cout, Vdd, cdsNet0);
pmos  M24 ( SUM, Vdd, net95);
pmos  M0 ( net90, Vdd, A);
pmos  M1 ( net90, Vdd, B);
pmos  M2 ( net117, Vdd, B);
```

```
pmos  M3 ( cdsNet0, net90, Cin);

pmos  M4 ( cdsNet0, net117, A);

pmos  M5 ( net96, Vdd, A);

pmos  M6 ( net96, Vdd, B);

pmos  M7 ( net96, Vdd, Cin);

pmos  M8 ( net95, net96, cdsNet0);

pmos  M9 ( net119, Vdd, A);

pmos  M10 ( net118, net119, B);

pmos  M11 ( net95, net118, Cin);

nmos  M25 ( SUM, GND, net95);

nmos  M26 ( Cout, GND, cdsNet0);

nmos  M12 ( cdsNet0, net88, Cin);

nmos  M13 ( cdsNet0, net116, A);

nmos  M14 ( net88, GND, A);

nmos  M15 ( net88, GND, B);

nmos  M16 ( net116, GND, B);

nmos  M17 ( net95, net94, cdsNet0);

nmos  M18 ( net94, GND, B);

nmos  M19 ( net94, GND, Cin);

nmos  M20 ( net94, GND, A);

nmos  M21 ( net95, net120, Cin);

nmos  M22 ( net120, net121, A);

nmos  M23 ( net121, GND, B);


endmodule


`endif
```