CROSS-STAGE LOGIC AND ARCHITECTURAL SYNTHESIS: WITH APPLICATIONS TO SPECIALIZED CIRCUITS AND PROGRAMMABLE PROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Gai Liu

December 2018

CROSS-STAGE LOGIC AND ARCHITECTURAL SYNTHESIS: WITH APPLICATIONS TO SPECIALIZED CIRCUITS AND PROGRAMMABLE PROCESSORS

Gai Liu

Cornell University 2018

Technology scaling, architectural innovations, and electronic design automation (EDA) are the three pillars supporting the exponential growth in computer hardware performance for the past six decades. With the traditional CMOS scaling approaching its end, there is an urgent need to explore novel techniques in the latter two aspects to sustain the long-standing trend of ever increasing computing performance and energy efficiency. This thesis studies new logic and architectural synthesis techniques that aim to significantly improve both productivity and quality for the digital hardware design. We re-examine the boundaries in the traditional EDA flow with the goals of (i) identifying and overcoming deficiencies in existing, well-established logic-level optimization methods, and (ii) raising the level of abstraction to ease architectural-level exploration for hardware specialization.

A common theme in this thesis is cross-stage optimization, where the synthesis decisions at an early stage are made aware of downstream optimization in an efficient manner to maximize the quality of results (QoRs). More specifically, we apply cross-stage optimization to tackle four challenging synthesis problems at logic and architectural level. At the logic level, we investigate both exact and approximate synthesis techniques: (P1) PIMap improves the quality of logic optimization by iteratively restructuring the logic network guided by technology

mapping; (P2) SCALS generates approximate circuits with statistical guarantees. At the architectural level, we target both specialized and programmable engines: (P3) ElasticFlow compiles irregular loop nests into specialized accelerators optimized for average-case performance; (P4) ASSIST synthesizes an instruction set architecture (ISA) description into programmable processor.

BIOGRAPHICAL SKETCH

Gai Liu was born to Xiusheng Liu and Yanping Tang in the city of Changsha, China on November 4, 1990. Gai grew up in Wuhan, China, where he attended the No.11 High School. During high school, Gai developed his love for science while practicing solving difficult chemistry and physics questions for the college entrance exam. In 2009, Gai was admitted to the Department of Electrical Engineering and Computer Science at Peking University, where he spent the best four years of his life absorbing knowledge of all kinds. Despite his love for computer science, he decided to major in solid state physics in college, partly because of his desire to understand the physical world, and partly because of being intimated by the extraordinarily smart programming geeks among his classmates.

After his undergraduate studies, he decided to pursue the PhD in the United States of America, and was luckily enough to be admitted into the School of Electrical and Computer Engineering at Cornell University under the guidance of Professor Zhiru Zhang. At Cornell University, Gai developed his knowledge and understanding of electronic design automation, particularly in the areas of high-level synthesis and logic synthesis. Gai married his wife, Xueying Zhang, in 2013. Their son, Anderson Ziyan Liu, was born in November 2015.

ACKNOWLEDGEMENTS

I am grateful to my family, friends, and mentors who have helped me along the way both personally and professionally. Without their support, this dissertation would not have been possible.

First and foremost, I would like to thank my advisor Zhiru Zhang, who has been a true source of knowledge, passion, and support throughout my time at Cornell University. I thank Zhiru for encouraging me to work on various challenging projects, for providing valuable ideas and suggestions throughout my graduate school career, and for helping me develop the taste for good research. Zhiru's passion for conducting creative yet rigorous research will be the inspiration in my future career. I would also like to thank the rest of my thesis committee, Amit Lal, Christoph Studer, and Adrian Sampson. I have truly enjoyed working with Amit on the Sonic FFT project. His approach to tackle seemingly impossible challenges taught me the importance of creative thinking. I also greatly appreciate his devoted support during the ups and downs of my graduate studies. I thank Christoph for providing invaluable suggestions and comments during my PhD. I admire his ability to quickly reduce any problem to the fundamental. His questions and suggestions bring another level of depth to the topics I studied. Last but not the least, I thank Adrian for being a continuous source of ideas, suggestions, and positivity. I learn something new and useful every time I interact with him, and his persistence for perfection elevated my understanding of what good research is supposed to be.

I would like to thank Professor Christopher Batten for being a truly inspirational scholar. His rigor and discipline towards research will always be the bar I strive to achieve in my future career. I am grateful for all the fond memories in the Zhang research group and in Cornell CSL. Special thanks to Steve Dai

and Mingxing Tan for bringing me up to speed in life and research at Cornell, and for the many research collaborations in the past years. Thank Steve also for being available to chat about anything at any time. I thank Ritchie Zhao for the collaborations on HLS related projects, and for always being direct and honest in research and in life. I would like to thank Chang Xu for the collaboration on the fruitful DATuner project. I thank Yuan Zhou, Zhenghong Jiang, Nitish Srivastava, Yi-Hsiang Lai, Ecenur Ustun, Hanchen Jin, Yuwei Hu, and Shaojie Xiang for all the activities we had in the research group, and for making the group such an enjoyable place to work and have fun. Thank Cunxi Yu for bringing me different perspectives on research and the industry. Your suggestions have always been invaluable. I would also like to thank my friends at CSL, Xiaodong Wang, Yao Wang, Tao Chen, Skand Hurkat, Khalid Al-Hawaj, Charles Jeon, Shunning Jiang, and Wenmian Hua for being the source of support and fun during my PhD.

Most importantly, I thank my parents, Xiusheng Liu and Yanping Tang, for all their extremely hard work that made me who I am, for always being understanding and supportive no matter the situation, and for creating an environment of curiosity and fairness at home. My parents have always been exceptional role models, whose wisdom is the treasure of my life. I deeply thank my wife Xueying Zhang, for always being supportive and understanding, and for always reminding me of the big picture. I am grateful for all the greatest joy that my son Anderson brings to me. My family are the source of energy in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since the origin of integrated circuits in the 1960s, digital systems are playing an increasingly important role in the everyday life of individuals, the modern society, and the future of humanity. Examples of such digital systems include the Internet, online social networks, augmented and virtual reality devices, digital health, digital banking systems, self-driving cars, smart cities, the Human Genome database, space exploration vehicles, and scientific supercomputing [11, 16, 35, 52, 54, 59, 61, 85, 98, 102]. The continuous exponential improvements in the performance, efficiency and cost reduction of digital systems have been one of the major driving forces to the world's technological and economic developments in the past 60 years [87]. Such an exponential growth has been the synergistic effects of the three major aspects:

1. **Technology scaling**. Moore's Law and Dennard's Scaling serve as the foundation for technology scaling since 1965. Moore's Law predicts that the number of transistors that can be economically integrated on a single chip doubles every two years, while Dennard's Scaling gives transistor design rules to enable exponential increase in transistor performance with constant power density. The semiconductor industry has been following Moore's Law and Dennard's Scaling until around 2010, during which computers were made faster, cheaper and more power efficient simply by moving to the next technology node. However, Dennard's Scaling has ended, and Moore's Law has slowed down in recent years [87]. Koomey's Law, which states the exponential improvement of energy efficiency in modern computing hardware, still holds in the post Moore's Law era [56].

The continuation of Koomey's law relies not only on improvements in transistor technology, but also the innovations in hardware architecture, design tools, and programming abstractions.

2. **Architectural innovations**. With the slowdown of Dennard's Scaling around 2006, the computer architecture community made a significant shift towards multi-core systems that extract performance through parallel execution [25]. In addition to the limited amount of parallelism in certain applications, scaling of parallel computers is also constrained by power density. The issue of power density is also known as "Dark Silicon" [36], which describes the phenomenon that only a small fraction of a large chip can be turned on at the same time. To address this problem, specialized computing, together with System-on-Chips (SoCs) comes into play, where a sea of customized accelerators speed up the target applications, giving a boost to the overall system performance.

3. **EDA innovations**. The electronic design automation (EDA) refers to the process of modeling the behavior of a digital system, synthesizing the high-level description of a design (e.g., Verilog or SystemC) down to the corresponding physical implementation, and verifying that the synthesized implementation achieves the required specification. EDA algorithms and techniques aim to maximize the benefit from technology scaling to deliver faster and more reliable end products. New device technologies also present challenges and optimization opportunities to the EDA flows and algorithms. More recently, there is an increasing effort in raising the level of abstraction to manage the design complexity of large-scale SoCs through technologies such as high-level synthesis (HLS) [22] and transaction-level modeling [14].

Traditionally, technology scaling is the primary driving force of the performance and efficiency of digital systems. However, physical limitations in technology scaling have led to a growing interest in architectural innovations. Consequently, a new wave of non-traditional system architectures have been proposed that incorporate heterogeneity, specialization, and approximation as a means to improve performance under strict power and energy constraints [12, 21, 36, 37, 44, 82, 86]. Commercially, domain-centric architectures have gained traction into main-stream computing systems, where examples include IBM's Cell Processor [43], the Tensor Processing Unit [51], Microsoft's Catapult project [81], Intel's Vision Processing Unit [8]. System architects exploring these more specialized approaches generally lack effective EDA tools for achieving productive design space exploration and fast design closure. Needless to add, modern EDA tools are already under immense pressure to cope with system-on-chip (SoC) devices that continue to scale in capacity and complexity in accordance with Moore's law — a long-standing challenge widely known as "bridging the design productivity gap". Consequently, improving the quality of applicability of the EDA tools will not only generate better designs from existing architectures, but also catalyze the innovation of new architectures through automating the design process of such architectures.

This thesis focuses on the synthesis portion of the EDA problem, including both algorithms and the workflow of the key synthesis steps. Specifically, we re-examine the boundaries in the traditional EDA flow with the goals of (i) identifying and overcoming deficiencies in existing, well-established logic-level optimization methods, and (ii) raising the level of abstraction to ease architectural-level exploration for hardware specialization.

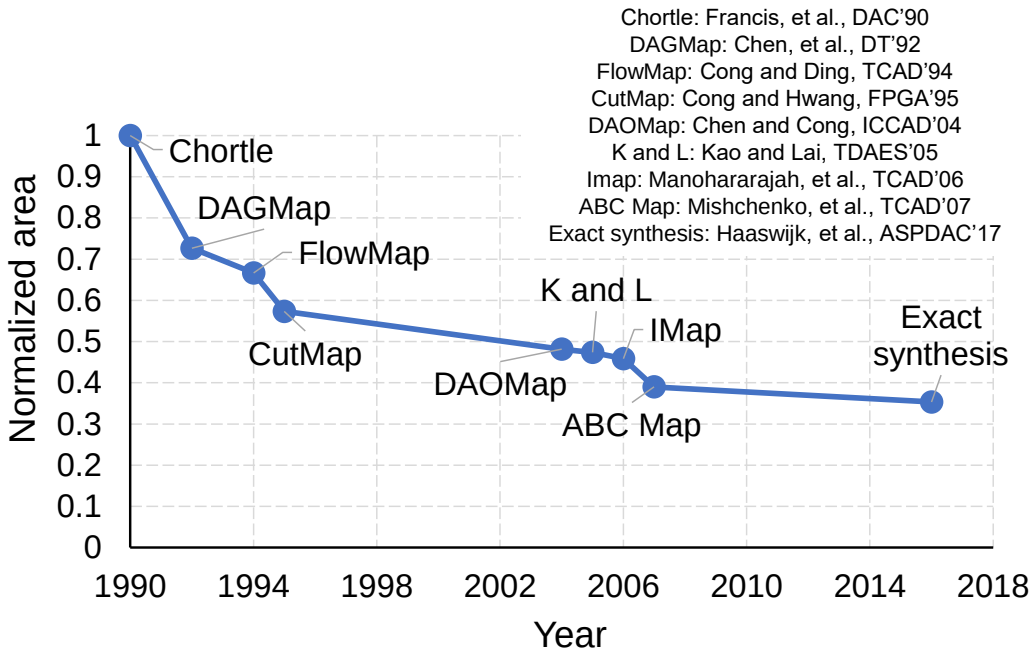


Figure 1.1: **Quality improvement of academic FPGA logic synthesis tools over the past 30 years —** In this study, we measure the quality improvement as the normalized area after mapping to 6-LUTs over a set of commonly-used benchmarks.

Broadly speaking, the EDA flow refers to the process of producing the specification of an integrated circuit from user's design description. The EDA flow usually relies heavily on design automation algorithms and tools to deliver high-quality solutions under tight time-to-market requirement. While technology scaling and architectural innovations have been the two well-recognized driving forces for technology advancement, algorithmic and methodical improvements in the EDA flow help extract extra performance out of the new architectural concepts and transistor technologies. To further illustrate this point, we study the quality improvement of the logic synthesis step in the EDA flow orthogonal of technology and architectural improvements. Using the same set of MCNC benchmarks [109], we collect the quality-of-results after mapping to 4-input lookup tables as the reference technology. Figure 1.1 summaries the

quality of results for the representative academic FPGA logic synthesis tools, where we compare their normalized performance with regard to minimizing the area of the circuit after technology mapping. Over the course of 30 years, there is a 3X improvement in area *in the logic synthesis step alone* (or an equivalent of 8% improvement per technology generation).

It is reasonable to extrapolate that if we consolidate the quality improvements across the stages in the EDA flow, the overall improvement will be the multiplicative contributions from each individual stage. This is non-trivial even when compared to the exponential performance and efficiency improvements brought by technology scaling at 6% to 31% frequency improvement per technology node [36]. In other words, we can extract additional significant QoR improvement by "simply" writing better tools for the digital design flow!

While technology scaling is limited by the physical dimensions of the transistors, advancement in the EDA flow is fundamentally governed by algorithmic complexity — a limit that evidently leaves significant room for improvements. (e.g., Cong and Minkovich show that designs synthesized from the state-of-the-art FPGA logic synthesis tools are 70X to 500X worse in area than the optimal results for a set of synthetic benchmarks [23].)

1.1 Overview of the EDA Flow

We first give an overview of a typical EDA flow, then discuss in more detail the architectural synthesis and logic synthesis steps in the flow that are the most relevant to this thesis. In the EDA flow, we start with a high-level description of a circuit design (e.g., an FFT circuit, a neural network accelerator, or a simple

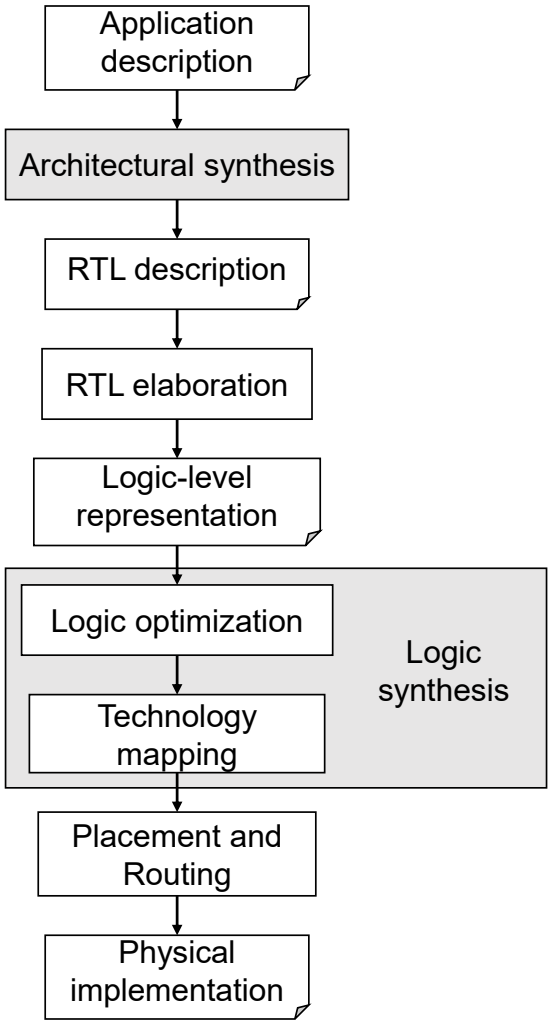


Figure 1.2: **A typical EDA flow.**

RISC-V processor), and synthesize the corresponding physical implementation by going through a sequence of *feed-forward* transformation stages. Typically, an EDA flow is abstracted into a sequence of individual steps to manage complexity, where each step generally involves solving one or more NP-hard optimization problems. Figure 1.2 shows a typical EDA flow. Starting from user's application, the behavioral-level description is first transformed into a timed register-transfer level (RTL) description using architectural synthesis techniques such as high-level synthesis. The RTL description is then elaborated and trans-

lated into a technology-independent logic representation, where IP blocks are inferred during logic synthesis. The technology-independent logic representation is also optimized during the logic synthesis phase using a sequence of heuristic transformations. The optimized logic-level circuit is then mapped to the specific technology library using technology mapping algorithms. The mapped circuit then goes through placement and routing (PnR) to obtain the physical layout. At this stage, the physical layout contains the necessary information for chip tapeout.

Architectural synthesis Architectural synthesis refers to the step that translates designer's high-level description of the application (e.g., in C/C++, SystemC) to the register-transfer level description of the hardware architecture, which includes the synthesis for fixed-function circuits and programmable architectures. The goal of the architectural synthesis step is to generate an optimized hardware architecture based on the performance constraints and/or designer's intent. The target hardware architectures can be broadly classified into two categories. Fixed-function circuits are application-specific architectures that can only execute one specific type of workloads. Examples include FFT circuit, AES encryption circuit, and fixed-function image processing pipeline. Programmable architectures are systems that have various degrees of programmability, which can execute a range of workloads and are usually compiled from a software-level description down to instructions of the architecture. CPUs, GPUs and programmable accelerators are examples of programmable architectures.

For both fixed-function circuits and programmable architectures, the designer provides a functional-level description of the design in the form of a software reference design or an ISA description. The corresponding EDA flow is

then invoked to generate the hardware implementation.

Logic synthesis Logic synthesis refers to the step of translating a logic-level representation (e.g., Boolean expressions, technology-independent gate-level logic networks) to an optimized, technology-specific logic representation. In this thesis, we define the logic synthesis step to include the logic optimization phase and the technology mapping phase. The logic optimization phase optimizes the logic circuit independent of the underlying physical technology. While the technology mapping step transforms the optimized logic circuit to a representation specific to the underlying physical technology. Examples of the technologies include ASIC standard cell libraries when targeting ASICs, or lookup-table tables when targeting FPGAs.

Challenges of EDA Flow Improving the quality, runtime, and scope of EDA algorithms is a challenging task that requires continuous research investment. There is currently no "ultimate recipe" in sight that solves the EDA problem once and for all. This is mainly due to the following challenges of the EDA flow:

- **Complexity within EDA stage**. Each stage of the EDA flow usually involves solving one or more NP-hard problems. As a result, there is a inherent tradeoff between quality-of-results (QoRs) and runtime. Significant efforts are needed to devise high-quality heuristic algorithms for these NP-hard problems.
- **Complexity across EDA stages**. The optimization objective of each stage is usually a heuristically determined abstraction that does not always cor-

relate with the downstream flow. This miscorrelation between the objectives of the different stages can result in missed opportunities in the final QoR.

- **Need for raising the level of abstraction**. Allowing the designer to specify the design at a higher level improves design productivity, facilitates design space exploration, and speeds up time-to-market. However, raising the level of abstraction complicates the design automation process, since the EDA algorithm needs to imply design decisions that are underspecified in the input description, and makes effective tradeoffs between alternative design decisions.

1.2 Case Studies of Drawbacks of the Traditional Approach

Here we present two case studies on evaluating the traditional approach for logic synthesis and architectural synthesis. We show that the conventional wisdom in these two well-studied areas does not always lead to the optimal quality-of-results, which motivates the need for cross-stage optimization.

Logic optimization can worsen technology mapping We study the impact of commonly used logic transformations on the gate count in the logic network as well as the corresponding LUT count after technology mapping. Traditional approaches usually assume a correlation between the gate count in the logic network and the LUT count after technology mapping, which serves as the principle for technology-independent optimizations [73]. That is, the goal of the technology-independent optimization phase is to minimize the delay and/or

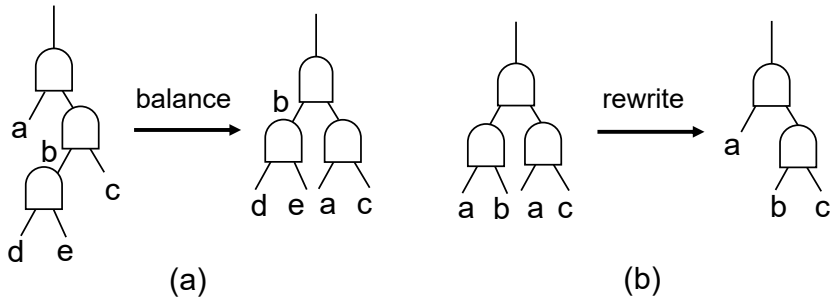


Figure 1.3: **Illustration of two common logic transformations —** (a) `balance`: balance the depth of the netlist using associative transform $a(bc) = (ab)c = (ac)b$. (b) `rewrite`: replace a sub-netlist with an equivalent but smaller one.

area of the gate-level logic network. Here we examine the validity of this commonly-used heuristic.

A logic transformation (or a move) applies optimization on the logic network in order to reduce the size or the number of levels of the network. Figure 1.3 shows two common logic transformations. The balancing transformation [72] tries to balance the depth of different paths in the netlist using associative transformations in the form as $a(bc) = (ab)c = (ac)b$. A rewriting transformation [73] visits each node in the network in a topological order, and enumerates all K-feasible cuts of the subject node. The Boolean function of each cut is then computed and matched against all the equivalence classes of K-variable functions. After trying all the available circuit representations for the given node, the rewriting move picks the one with the largest improvement.

Our experimental methodology is to iteratively perturb a given logic network (or a sub-network) to generate a sequence of equivalent design points with varying sizes in terms of gate count and LUT count. More specifically, we use two different strategies to perturb the logic network. The *gate-centric perturbation* enumerates a set of logic transformations to the input network, then greedily accepts the resulting logic networks that reduce the gate count. This way we it-

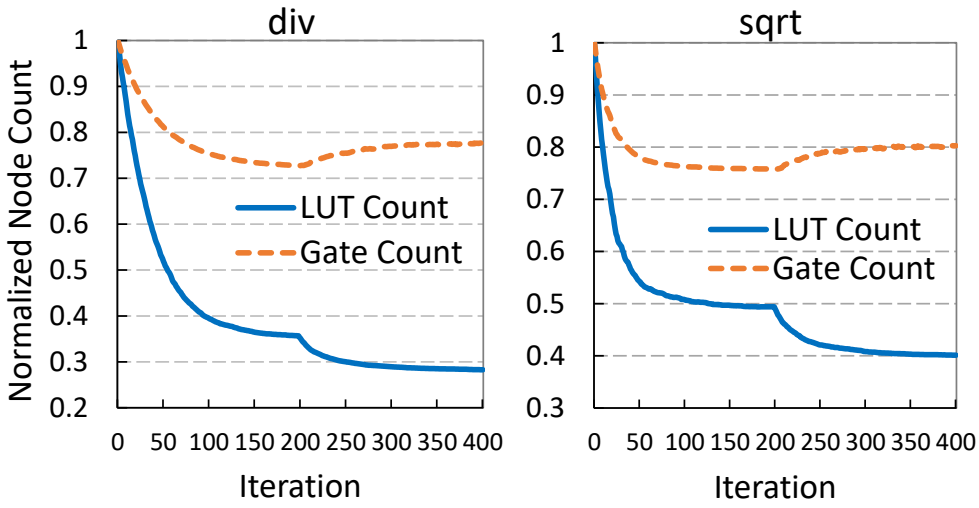


Figure 1.4: **Correlation between gate count in the logic network and post-mapping LUT count —** For the first 200 iterations, we perturb the logic network with the objective of reducing gate count. After 200 iterations, we change the objective to reducing LUT count.

eratively generate a sequence of design points with decreasing number of gates, at the same time, the LUT count of each design point is also recorded. With the second strategy called LUT-centric perturbation, we also iteratively apply a set of candidate transformations to the logic network and measure the LUT count after each transformation. However, we only accept the transformations that reduce the LUT count of the resulting mapped netlist. We record both gate count and LUT count upon the acceptance of each transformation.

Here we evaluate two representative designs from the EPFL arithmetic benchmark suite [2], and use and-inverter graph (AIG) as the gate-level representation of the logic network. We use the aforementioned method to apply three transformations in the ABC logic synthesis framework [10] (balance, refactor, rewrite) to generate 400 intermediate design points for each benchmark. Notably, we employ the gate-centric perturbation for the first 200 iterations, and switch to LUT-centric perturbation mode afterwards.

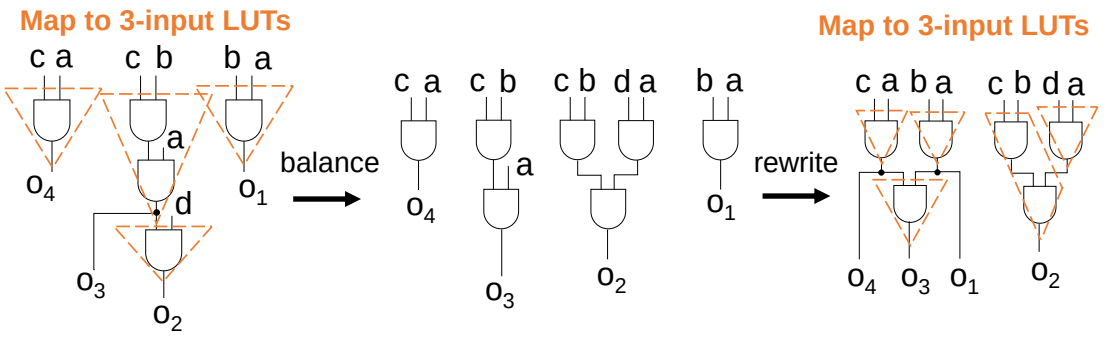


Figure 1.5: **An illustrative example showing the drawback of the existing logic synthesis approach.**

Figure 1.4 shows the normalized LUT count and gate count during the 400 iterations of perturbations. During the initial phase of gate-centric perturbation, the decrease of LUT count coincides with the gate count reduction. Eventually, both descending curves level off, which seems to suggest that little room is left for improving area. Interestingly, when switching to LUT-centric perturbation after 200 iterations, we observe further reduction in LUT count with an increasing gate count. While we are only presenting two benchmarks here due to space limitation, we observe from our experiments very similar trends (to Figure 1.4) across a broad range of designs.

To further understand the disconnect between the target of logic optimization and the result of technology mapping, we study the illustrative example shown in Figure 1.5. In this example, we apply a sequence of two logic transformations, balance and rewrite, to the initial gate-level logic network, and observe the transformed circuit both before and after technology mapping. At the gate level, the optimizations reduce the depth of output O_2 by one at the cost of increasing the gate count by one. However, after technology mapping, the LUT level of O_2 stays at two, while the LUT level of O_3 increases by one. In addition, the LUT count increases by one. Since the logic optimization is unaware of its

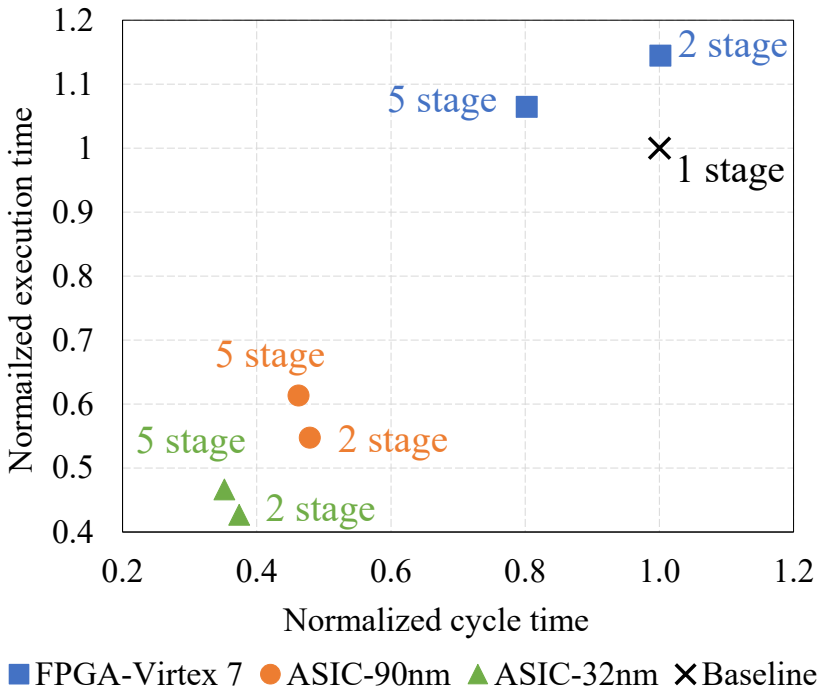


Figure 1.6: **Normalized cycle time and execution time for ASIC and FPGA targets with different pipeline structures —** Execution time is measured as the total CPU time over seven representative kernels. Results are normalized to the one-stage implementation of the corresponding technology.

effect on technology mapping, the final design QoR becomes worse after the logic transformations.

Physical implementation effect is hard to model during architectural exploration A common strategy to improve the cycle time of a processor design is pipelining. However, determining the optimal pipeline schedule during architectural exploration is a nontrivial task without post-PnR timing information. Figure 1.6 shows the implementation results of three different pipeline schedules targeting RISC-V 32I ISA across three different technologies, where the one-, two-, and five-stage implementations are manually optimized versions from [19]. The execution time of each design point is measured as the sum of

CPU times over seven representative kernels (sorting, vector-vector addition, soft integer multiplication, recursive function calls, integer programming, etc.), which measures the design performance in terms of the runtime needed to finish the seven kernels.

For designs targeting Virtex 7 FPGAs, both the two- and five-stage implementations require longer execution time when compared to the one-stage counterpart. This is mainly due to the tradeoff between cycle time and instructions per cycle (IPC) of the pipelined designs: the five-stage design achieves a shorter cycle time, but has a lower IPC due to more frequently pipeline stalling and squashing. The net result of this tradeoff is unlikely to be accurately determined before physical implementation and cycle-accurate simulations. In addition, the two-stage implementation fails to improve the critical path delay of the one-stage design. For ASIC targets, although increasing the number of pipeline stages reduces cycle time, the trend for execution time is again non-intuitive. For both 90nm and 32nm targets, the two-stage implementations achieve shorter total execution time for the seven kernels in this experiment when compared to their five-stage counterparts.

1.3 Enabling Cross-Stage Optimization in EDA Flow

A common drawback of the previous two case studies is that the optimization algorithm only considers its effect within a single stage in the EDA flow, and relies pre-determined heuristics for guiding the optimization process. As we have shown in Section 1.2, there exists a significant miscorrelation between the optimization objective at the early stages of the EDA flow and the actual QoR in the

downstream stages. To recover the quality loss due to these miscorrelations, we argue for an improved EDA flow with cross-stage optimization, where the synthesis decisions at an early stage are made aware of downstream optimization in an efficient manner to maximize the QoRs.

Our proposed approach is based on the concept of scale-out design automation — a methodology that uses a large number of parallel computing nodes to improve the quality of EDA algorithms. Our approach is motivated by the following observations:

1. With the emergence of cloud computing, computing resources are rapidly becoming abundant and inexpensive, motivating a "big compute" approach towards design automation. In such a "big compute" scheme, an EDA problem can be solved on the datacenter scale in a massively parallel way, which can potentially significantly increase the design QoR while maintaining a total runtime similar to the traditional EDA tools.
2. Many of the EDA problems can be solved using stochastic optimization or autotuning-based techniques where a "perturb-then-measure" procedure is effective in incrementally improving the design QoR. Such frameworks enable an incremental improvement design methodology that gradually improves the design QoR as more computing resources become available.

Specifically, we list the key ingredients that enable the cross-stage optimization in the EDA flow:

1. **Automatic generation of new design points**. Automatically synthesizing a high-quality design from the high-level description enables fast iterations between different stages of the EDA flow. As we raise the level of

abstraction of the input to the EDA flow, it is important to devise effective techniques for behavioral-level synthesis.

2. **Eager evaluation of new design points**. The EDA flow should utilize the abundant computing resources to eagerly evaluate the design choices made at the upper-stream stages. In contrast to the traditional methods which estimate the quality of a proposed design point using a heuristic QoR proxy within a single stage, our proposed approach executes one or more downstream stages for each proposed design point to obtain more accurate QoR information.
3. **Intelligent exploration of the design space**. For realistic designs, it is computationally intractable to evaluate every design point in the design space. Thus, it is important to utilize intelligent design space exploration algorithms to maximize the reward of evaluating a small subset of the design points.

This thesis centers around these three key enablers for cross-stage optimization in the EDA flow. We will apply cross-stage optimization to four aspects of the EDA flow spanning logic synthesis and architectural synthesis.

1.4 Thesis Structure and Contributions

This thesis consists of four individual projects spanning the topics of logic synthesis and architectural synthesis with a common theme of improving quality-of-results using cross-stage optimization. We use Figure 1.7 and Table 1.1 to summary the structure and main techniques used in the remaining chapters of this thesis.

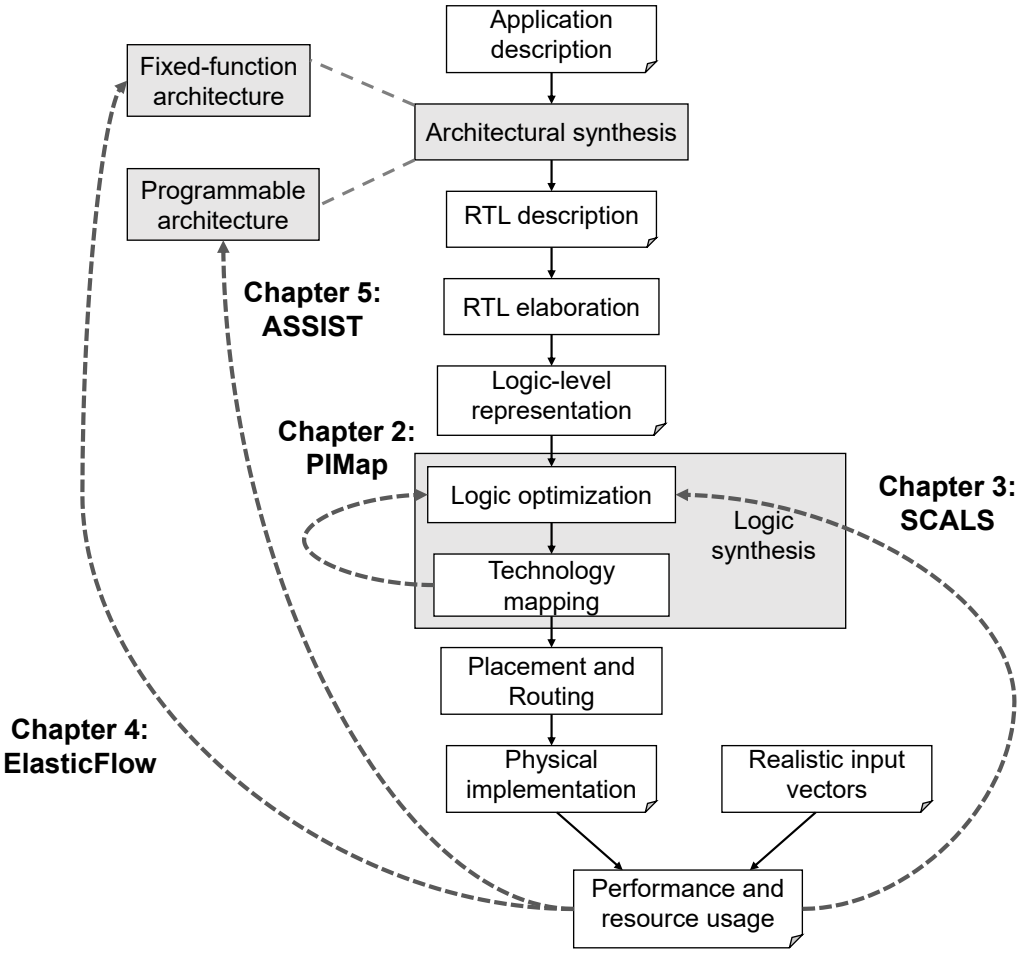


Figure 1.7: **Overview of the thesis structure —** Dashed arrows indicate the feedback mechanisms used in cross-stage optimizations.

Chapters 2 and 3 concern improving the quality of logic synthesis through cross-stage optimization. Specifically, Chapter 2 studies the classic logic synthesis problem where the circuits optimized by the logic synthesis step are equivalent to their logic-level representation. We propose PIMap, which couples logic transformations and technology mapping under an iterative improvement framework to minimize the circuit area for LUT-based FPGAs. In each iteration, PIMap randomly proposes a transformation on the given logic network from an ensemble of candidate optimizations; it then invokes technology mapping and makes use of the mapping result to determine the likelihood of accepting

Table 1.1: **Summary of cross-stage optimization techniques used in each chapter.**

Automatic design point generation	PIMap	Randomized proposal of logic transformations
	SCALS	Randomized proposal of logic transformations
	ElasticFlow	HLS with source-to-source transformation
	ASSIST	ISA to RTL synthesis w/ configurable pipeline structure
Eager evaluation	PIMap	Use technology mapping result
	SCALS	Use technology mapping and simulation results
	ElasticFlow	Use physical implementation and simulation results
	ASSIST	Use physical implementation result
Intelligent design space exploration	PIMap	Stochastic optimization
	SCALS	Stochastic optimization
	ElasticFlow	ILP-based optimization
	ASSIST	Autotuning

the proposed transformation. To mitigate the runtime overhead, we further introduce parallelization techniques to decompose a large design into multiple smaller sub-netlists that can be optimized simultaneously. Experimental results show that our approach achieves promising area improvement over a set of commonly used benchmarks. Notably, PIMap reduces the LUT usage by up to 14% and 7% on average over the best-known records for unconstrained area minimization for the EPFL arithmetic benchmark suite.

Chapter 3 introduces SCALS, which extends PIMap to approximate logic synthesis to generate inexact implementations of logic functions in exchange for better design qualities such as area, timing and power consumption. We propose a statistically certified approximate logic synthesis framework using techniques from stochastic optimization, and integrate it into a state-of-the-art parallelized technology mapper. During the synthesis process, our framework continuously monitors the quality of the generated designs using statistical testing, leading to approximate designs that adhere to user-specified error constraints with a high confidence level. Experimental results demonstrate significant area and timing improvements over the exact counterparts with small and control-

lable amount of deviations from the exact outputs. While PIMap only requires feedback from technology mapping, SCALS also requires gate-level simulation results using representative test vectors to evaluate the accuracy impact of the proposed approximate logic transformations.

Chapters 4 and 5 focus on improving architectural synthesis with cross-stage optimization, where the common theme is to use downstream implementation results to guide the design decisions made during architectural exploration. Chapter 4 discusses techniques to improve modern HLS tools to handle irregular loop nests. Modern high-level synthesis tools commonly employ pipelining to achieve efficient loop acceleration by overlapping the execution of successive loop iterations. While existing HLS pipelining techniques obtain good performance with low complexity for regular loop nests, they provide inadequate support for effectively synthesizing irregular loop nests. For loop nests with dynamic-bound inner loops, current pipelining techniques require unrolling of the inner loops, which is either very expensive in resource or even inapplicable due to dynamic loop bounds. To address this major limitation, we propose ElasticFlow, a novel architecture capable of dynamically distributing inner loops of an loop nest with irregular workloads to an array of processing units (LPUs) in an area-efficient manner. The proposed LPUs can be either specialized to execute an individual inner loop or shared among multiple inner loops to balance the tradeoff between performance and area. To enable efficient implementation of the ElasticFlow architecture for various workloads, we use profiling results from realistic test vectors to guide the design of the ElasticFlow architecture through an integer linear program formulation. We evaluate ElasticFlow using a variety of real-life applications and demonstrate significant performance improvements over a state-of-the-art commercial HLS tool for Xilinx FPGAs.

Chapter 5 proposes ASSIST, a synthesis framework that generates RTL description of optimized single-issue in-order processors from ISA descriptions. Given an ISA, ASSIST synthesizes processors with different pipeline schedules ranging from a combinational processor datapath to a seven-stage pipelined datapath. Using the RISC-V 32I ISA as a case study, we show that when compared to manually optimized RISC-V processor implementations of the same class, ASSIST generates various design points from the same ISA, some of which dominates the manually written counterparts in the area-performance Pareto frontier. To automate the exploration of the optimal pipeline schedule, we interface ASSIST with the autotuning framework OpenTuner [4] to efficiently navigate the complex search space of the pipeline schedules. We use the ASSIST framework to synthesize processors with cryptographic instruction set extensions and show that ASSIST provides a flexible framework to synthesize processors with user-defined ISAs without any user intervention during the synthesis process. The synthesized cryptographic cores achieve significant performance improvements over implementations using the baseline ISAs.

The major contributions of this thesis include:

1. We observe and study the deficiencies in the logic and architectural synthesis stages in the traditional EDA flow, and show that significant QoR gains can be achieved by addressing these deficiencies through cross-stage optimization.
2. Centered around the idea of cross-stage optimization, we present four major techniques and the corresponding tools that improve the EDA flow in areas of traditional logic synthesis, approximate logic synthesis, accelerator architecture synthesis, and programmable processor synthesis.

3. We experimentally demonstrate that the proposed approaches and tools achieve significant improvements over state-of-the-art techniques and results, showing that cross-stage optimization is a general technique that can be applied to various stages of the EDA flow.

1.5 Collaborations, Funding, and Previous Publications

This thesis would not have been possible without the contributions from the colleagues in the Zhang Research Group at Cornell University. My advisor Zhiru Zhang was integral in all four projects. The idea behind PIMap [67] was partially motivated by Mingxing Tan's study on applying autotuning to optimizing the logic transformation sequence in logic synthesis. Mingxing Tan proposed the initial ideas for ElasticFlow, and collected the majority of the benchmarks used in ElasticFlow [64, 96]. He proposed and implemented the LPA architecture and the corresponding synthesis techniques in ElasticFlow. He was also the main contributor for the experimental results on design space exploration, LPU sharing, and comparison with CGPA in ElasticFlow. Steve Dai proposed and implemented the LPU allocation algorithm in ElasticFlow. Ritchie Zhao participated in the writeup of the publication for ElasticFlow, and provided suggestions on many aspects of the ElasticFlow techniques.

This thesis was supported in part by NSF Awards #1337240, #1453378, #1512937, #1618275, a DARPA Young Faculty Award D15AP00096, and a research gift from Xilinx, Inc.

The complete list of publications is shown below:

1. Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar 2018

2. Steve Dai, Gai Liu, and Zhiru Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018

3. Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018

4. Gai Liu and Zhiru Zhang. Statistically Certified Approximate Logic Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2017

5. Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(11):1817–1830, Nov 2017

6. Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. Enabling Adaptive Loop Pipelining in High-Level Synthesis. *Asilomar Conference on Signals, Systems, and Computers*, Oct 2017

7. Gai Liu and Zhiru Zhang. A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping. *Int'l Symp. on*

Field-Programmable Gate Arrays (FPGA), Feb 2017

8. Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017
9. Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017
10. Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, and Zhiru Zhang. Improving High-Level Synthesis with Decoupled Data Structure Optimization. *Design Automation Conf. (DAC)*, Jun 2016
11. Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2015
12. Gai Liu and Zhiru Zhang. A Reconfigurable Analog Substrate for Highly Efficient Maximum Flow Computation. *Design Automation Conf. (DAC)*, Jun 2015
13. Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014
14. Gai Liu, Ye Tao, Mingxing Tan, and Zhiru Zhang. CASA: Correlation-Aware Speculative Adders. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Aug 2014

We briefly summarize the publications that are not discussed in the rest of the thesis. The Celerity project [31] studies the architecture and design method-

ology for an open-source 511-core RISC-V tiered accelerator fabric, where I participated in the synthesis of a binarized neural network accelerator using the high-level synthesis methodology. I collaborated with Steve Dai on proposing a scalable approach that solves the resource-constrained scheduling problem exactly using a joint SDC and SAT formulation [27], where I contributed to the initial formulation based on SDC and SAT. Rosetta is a realistic HLS benchmark suite for software programmable FPGAs [114]. I helped develop and refine several benchmarks in the suite. I worked with Chang Xu and other collaborators on DATuner [108], an autotuning framework for FPGA compilation using a parallel bandit-based approach. I helped refine the DATuner technique during various stages of the project. In [30], I contributed to the technique to enable dynamic hazard resolution for pipelining irregular loops in HLS. I participated in the technical discussion and implementation with Ritchie Zhao and other collaborators in [112], where we developed improvements to HLS using decoupled data structure optimization. In [66], I proposed a reconfigurable substrate based on analog circuit components, whose steady-state voltage values at the output nodes represent the solution to the corresponding maximum flow problem. I participated in the collaboration on the XLOOPS project targeting architectural specialization for inter-iteration loop dependence patterns [92]. I was responsible for the energy consumption model of the proposed architecture. I led the project on designing efficient approximate adders using correlation-aware speculation [63]. The proposed approximate adders achieve significant performance and area improvements over the previously proposed designs.

CHAPTER 2

PIMAP: GUIDING LOGIC SYNTHESIS WITH TECHNOLOGY MAPPING

An important step in the EDA flow is called technology mapping, which transforms a gate-level Boolean logic network[1] into a functionally equivalent netlist composed of look-up tables (LUTs). Minimizing the depth and the total LUT count of the mapped netlist are two of the typical optimization goals for an FPGA-targeted technology mapper.

A key challenge to technology mapping is that the quality of the mapping solution depends heavily on the structure of the input logic network. However, it is well known that the problem of restructuring the network for depth- or area-optimal technology mapping is NP-hard [38]. Modern FPGA synthesis tools usually apply a series of structural optimizations as a heuristic to transform the input logic network to be more amicable for technology mapping and other downstream optimizations [49, 73]. Examples of the commonly used logic optimizations include balancing the levels of different paths in a logic network (i.e., balancing), and replacing a sub-network with a smaller one that realizes the same function (i.e., rewriting). In practice, such logic optimizations are usually interleaved with each other and repeatedly applied to better optimize the logic network. While such transformations can effectively reduce the complexity of the logic network in terms of the gate count and/or the number of logic levels, we argue that there still exists considerable room in improving the FPGA mapping quality based on two important observations:

[1] In the rest of the chapter, we use the term *logic network* to denote a pre-mapping gate-level Boolean logic network.

- The mainstream FPGA synthesis frameworks use a fixed predetermined sequence of pre-mapping logic transformations that may not always generate high-quality logic structures. For example, the popular academic tool ABC provides synthesis scripts with more than 20 different optimization sequences [10]. Since the efficacy of these sequences varies across different designs, it is very challenging for a user to quickly identify the best sequence to employ given a new design.
- Quality improvements from logic-level optimizations do not always lead to better quality of results after technology mapping. Specifically, as we have shown in Chapter 1.2, minimizing the gate count or the number of logic levels may not necessarily translate to reduced LUT count or depth in the final mapped netlist, thereby creating a gap between the optimality at the logic stage and the technology mapping stage.

Clearly, reducing the depth and area of logic network does not necessarily translate to performance improvements or area savings after mapping. To address this challenge, we propose PIMap — a parallelized iterative improvement approach to LUT mapping. Unlike existing methods that decouple the logic transformations from technology mapping, PIMap makes use of the actual mapping results to guide a series of randomly proposed structural optimizations. Proposing logic transformations in a probabilistic way allows PIMap to explore a larger design space that cannot be uncovered by fixed optimization sequences. According to our experimental results, PIMap consistently outperforms the state-of-the-art LUT mapping solutions for area and delay optimizations.

Since iterative improvement usually comes with nontrivial runtime over-

head, we further propose techniques to decompose a large netlist into multiple smaller sub-netlists, and optimize these sub-netlists in parallel across multiple machines. This parallelization framework enables PIMap to handle large circuits with more than 40 thousand LUTs, with a synthesis time in the range of tens to hundreds of seconds. In addition, PIMap also allows the users to easily explore the trade-offs between the design quality and the synthesis effort in runtime.

Our primary technical contributions are as follows:

- We provide a quantitative study on the (mis)correlation between the gate count reduction in the pre-mapping logic network and the LUT count savings after technology mapping, demonstrating the drawbacks of the mainstream logic synthesis techniques.
- We propose a stochastic iterative improvement algorithm and associated parallelization techniques to enable efficient mapping-in-the-loop area optimization for LUT-based FPGAs.
- We devise a network repartitioning scheme that enables logic optimization across sub-netlist boundaries, which leads to significant improvements in the solution quality.
- We extend the area minimization flow to delay optimization by proposing techniques to extract delay-critical sub-netlists and minimize the output arrival times.
- We show promising improvements in area reduction for a set of common benchmarks, including breaking many best-known records for the EPFL arithmetic benchmark suite.

- We demonstrate the effectiveness of the delay optimization flow for a set of common combinational and sequential benchmark circuits, including improving 13 out of the 20 records in the EPFL benchmark suite.

2.1 Preliminaries

In this section, we discuss the basics of technology mapping and common logic transformations used in PIMap.

2.1.1 Overview of Technology Mapping

Generally speaking, technology mappers are divided into structural mappers and functional mappers [74]. Structural mappers consider the input logic network as fixed, and attempt to cover the circuit with K-input LUTs. Functional mappers are allowed to modify the structure of the logic network before mapping to LUTs. In this chapter we focus on functional mappers for generating higher-quality mapping solutions.

Before covering the logic network with LUTs, functional mappers usually apply a sequence of logic transformations to the network, which we call *moves*. The goal of these moves is to prepare the network for technology mapping so that the subsequent LUT covering step can generate high-quality results in terms of LUT depth or LUT count. We first describe the mechanism of covering a logic network with LUTs.

During the LUT covering step in technology mapping, we view the logic

network as a directed acyclic graph, where the nodes represent logic gates and the edges capture the connections between the gates. We define a cone C_v at node v as the sub-netlist of v and some of its predecessors so that any path from a node in C_v to v is entirely contained in C_v. A cone is said to be K-feasible if there are no more than K nodes outside C_v that have edges pointing to the nodes in C_v. A cut of C_v is defined to be the set of input nodes of C_v.

In LUT-based FPGAs, we can implement any K-feasible cones using a K-input LUT. Consequently, the mapping problem reduces to the problem of optimally covering the input graph with K-feasible cones [78]. A LUT covering framework generally consists of cut enumeration, cut ranking, cut selection, and final mapping generation. Cut enumeration explores all K-feasible cuts at each node, while cut ranking evaluates the quality of the cuts based on the optimization objective. Cut selection determines the optimal cut for each node based on the ranking information to generate the final covering solution.

2.2 PIMap Framework for Area Optimization

PIMap decomposes a large circuit netlist into smaller sub-netlists, and uses an iterative routine to minimize the area of these sub-netlists in parallel. The area minimization routine integrates commonly used logic transformations and technology mappers to progressively improve the design quality. In this section, we describe the PIMap techniques in detail.

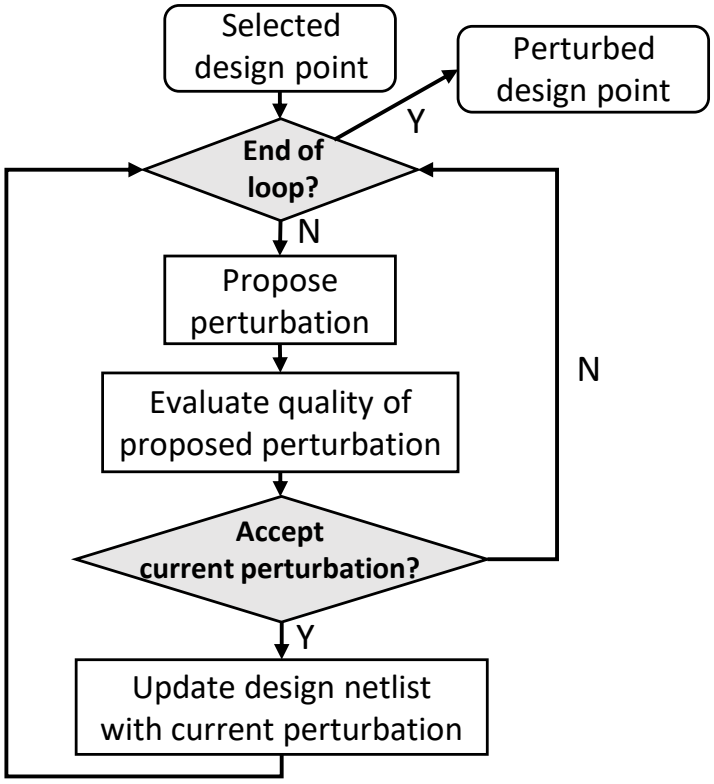


Figure 2.1: **Flow of the iterative perturbation routine.**

2.2.1 Iterative Area Minimization

The very core of PIMap is an iterative area minimization framework that repeats three major steps: (1) proposing logic transformation moves, (2) evaluating the quality of the move through technology mapping, and (3) determining whether to accept the proposed move. Figure 2.1 sketches the high-level design flow of this iterative procedure.

Proposing a transformation move PIMap makes use of a collection of logic transformation moves, denoted as set T. Each move in T is capable of optimizing a given logic network for a certain target, such as reducing the number of nodes in the circuit and balancing the node levels of different paths. We further

associate T with a discrete probability distribution named P, where the probability of selecting the i^{th} at any iteration is denoted as p_i. At the beginning of each iteration, PIMap randomly chooses one logic transformation from T based on P. The transformed network is then evaluated by invoking an existing area-minimizing technology mapping algorithm.

Evaluating a move In this step, the transformed netlist is first mapped to K-input LUTs using an existing area-oriented technology mapper. With unconstrained area optimization, we directly tie the quality metric Q of a proposed move to the number of LUTs in the mapped circuit netlist (denoted as N_{LUT}). We note that Q can be extended to include other user-specified factors such as the number of gates in the pre-mapping logic network.

Accepting a move After obtaining the quality metric of the currently proposed move Q_{curr} and that of the previous iteration, denoted as Q_{prev}, we use the Markov Chain Monte Carlo (MCMC) method to probabilistically determine whether to accept the proposed move [41]. In particular, we employ the Metropolis-Hastings algorithm [47] for calculating the acceptance probability. This process is detailed in Algorithm 1, which dictates that if the quality of the current move is better than the previous one, we accept the current move unconditionally. Otherwise, we accept the move with a small probability that decreases exponentially as Q_{curr} increases. The parameter γ controls the tolerance of accepting a locally inferior move, which is empirically determined by profiling the quality of reference designs with different values of γ. Once a move is accepted, we update Q_{prev} to be Q_{curr}, save the updated network, and continue with a new proposal. On the other hand, if the current move is rejected,

we do not update Q_{prev} and directly proceed to the next iteration. During the search procedure, we also keep track of the best mapping result and the corresponding circuit netlist. We return the best result at the end of the iterative area minimization routine.

Algorithm 1 Calculating acceptance probability

```
if Q_curr < Q_prev then
    Accept the current move
else
    // rand(): random number between 0 and 1
    if rand() < e^{-γ(Q_curr/Q_prev)} then
        Accept the current move
    else
        Reject the current move
```

2.2.2 Netlist Extraction and Parallel Optimization

To enable parallel optimization of multiple sub-netlists, PIMap automatically extracts a user-configurable number of non-overlapping sub-netlists from a mapped netlist, and optimize them in parallel through multithreading. Figure 2.2 conceptually illustrates the netlist extraction and parallel optimization steps. Given an input logic network, we first map it into a circuit netlist composed of LUTs shown as the triangles in Figure 2.2. We then partition the netlist into multiple sub-netlists, and apply the area minimization technique described above to optimize the sub-netlists in parallel. After optimizing the sub-netlists, we recombine them into a single netlist, and start the next trial of the sub-netlist extraction and optimization. We define a trial as the four steps including sub-netlist extraction, converting a mapped netlist back to logic network, iterative area minimization, and sub-netlists recombination.

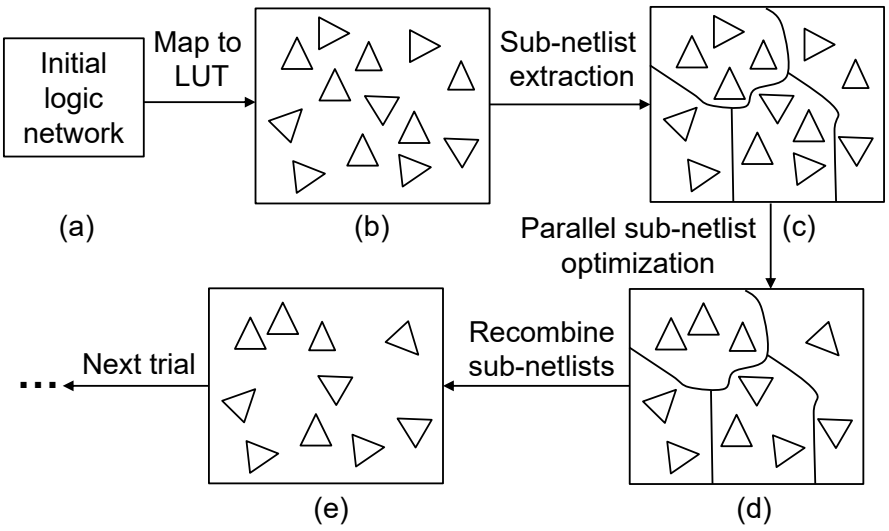


Figure 2.2: **Illustration of netlist decomposition and parallel optimization —** (a) Original logic network. (b) Netlist after LUT mapping, where each triangle represents a LUT, the connections between LUTs are omitted. (c) The four sub-netlists after sub-netlist extraction. (d) The four sub-netlists after optimization. (e) The netlist after recombining the four optimized sub-netlists.

Partitioning mapped netlists Algorithm 2 describes the steps required to partition a mapped netlist to enable effective parallelization. More specifically, the inputs to our partitioning algorithm include (1) a netlist that has already been mapped to LUTs, (2) a parallelization factor n, and (3) a size constraint M for each sub-netlist, the goal is to extract n non-overlapping sub-netlists with each of which containing no more than M LUTs. It is worth noting that partitioning the mapped netlist allows us to easily merge the optimized sub-netlists to regenerate the complete LUT netlist. More importantly, any improvement to a sub-netlist will directly contribute to the overall LUT savings in the recombined netlist.

When generating a sub-netlist, our algorithm first randomly picks a *seed*, and expands the sub-netlist using breadth-first search (BFS) from the seed until the number of LUTs in the sub-netlist reaches M. When constructing the

Algorithm 2 Extracting sub-netlists

Input: A mapped netlist G_0, a parallelization factor n, and a sub-netlist size constraint M.
Output: n sub-netlists $\{G_1, G_2, ..., G_n\}$, each of which contains M LUTs.

```
// G_res is the residual graph of G_0
```
Initialize $G_{res} = G_0$, and $G_1 = G_2 = ... = G_n = \varnothing$
```
// extract the i^th sub-netlist
```
for i *from* 1 *to* n **do**
    Randomly pick a node j in G_{res}
    Start from j, visit G_{res} in breadth-first order:
    for *each node* k *during traversal* **do**
        Add node k to G_i
        Remove node k from G_{res}
        if *size of* G_i *reaches* M **then**
            break
    // Determine primary inputs (PI) and primary outputs (PO) of G_i
    for *each node* k *in* G_i **do**
        for *each fan-in* l *of* k **do**
            if l *is not assigned inside* G_i **then**
                Add l to the PI set of G_i
        if *(*k *is a PO of* G_{res}*) or (*k *is used in* G_{res}*)* **then**
            Add k to the PO set of G_i
return $\{G_1, G_2, ..., G_n\}$

sub-netlists, we also maintain a residual graph that contains the nodes not yet added to any sub-netlists. The residual graph is initialized to be the same as the original netlist, and will gradually decrease in size as more sub-netlists are extracted. After generating the first sub-netlist, the algorithm will pick another random seed, and extract the next sub-netlist from the residual graph until all the n sub-netlists have been generated. In case BFS cannot find a cluster of size M, the algorithm extracts another cluster and append it to the sub-netlist until the sub-netlist reaches a size of M LUTs. After the partitioning step, our algorithm assigns the primary inputs (PI) and primary outputs (PO) of each sub-netlist by identifying the nodes that have external fan-ins as well as those

that fanout to external nodes.

Optimizing sub-netlists After obtaining the sub-netlists from the previous step, PIMap distributes them to available computing resources for independent optimization. We create one thread for each sub-netlist, and assign threads to machines to balance the load. Optionally, PIMap allows the user to use multiple threads to optimize different copies of the same sub-netlist in parallel to increase the likelihood of generating a high-quality solution. After all threads finish execution, a master thread collects the optimized sub-netlists, and combine them to reform the entire design. This combining process involves concatenating all the sub-netlists into a single netlist, and remove the PIs and POs of each individual sub-netlists. Since all sub-netlists are of equal or very similar size, the runtime of different threads are similar to each other. Consequently, the workloads of different threads are highly balanced.

2.2.3 Overall Flow

Figure 2.3 shows the overall flow of PIMap. PIMap takes the initial logic network as the input, and first uses an area-oriented technology mapper to transform the logic network into a mapped netlist. PIMap then uses the sub-netlist extraction technique detailed in Section 2.2.2 to extract a number of sub-netlists. Since the iterative area minimization requires a gate-level logic network, we apply a netlist decomposition technique, such as structural hashing, to convert the mapped sub-netlist back to the corresponding logic sub-networks. These logic sub-networks are subsequently optimized using the iterative area minimization technique, which generates the optimized version of the mapped sub-

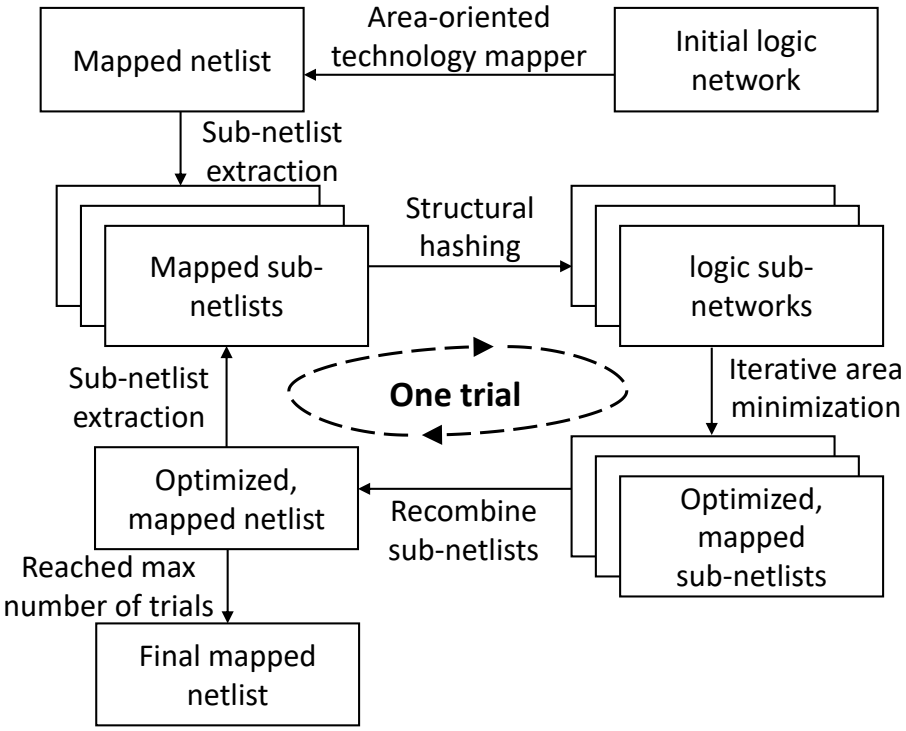


Figure 2.3: **Overall synthesis flow of PIMap.**

netlists. PIMap then recombines these optimized netlists into a single netlist that is equivalent to the original design. We define a trial as the four steps including sub-netlist extraction, converting a mapped netlist back to logic network, iterative area minimization, and sub-netlists recombination.

2.3 Extension to Delay Optimization

We extend the PIMap framework detailed in Section 2.2 to delay optimization with the goals of (1) minimizing the delay of the circuit after technology mapping, and (2) reducing the area of the mapped circuit when possible without increasing the delay. We discuss two techniques to achieve these goals. First, we augment the iterative area minimization routine by considering the input

arrival time information and output arrival time constraints to enable delay optimization of the sub-netlists. Second, we enhance the netlist extraction algorithm to extract delay-critical regions of the overall design for effective delay optimization.

2.3.1 Iterative Logic Transformation under Required Time Constraint

Given a sub-netlist with arrival times for each of the inputs and the required times for each of the outputs, the iterative logic transformation routine aims to find an optimized sub-netlist implementation that (1) satisfies the output required times, (2) minimizes the arrival time of the outputs, and (3) reduces the circuit area after technology mapping when possible. Figure 2.4 illustrates the flow for iterative logic transformation for delay optimization, where we highlight the steps that differ from the flow in Figure 2.1.

Computing input arrival time and output required time Given an extracted sub-netlist in LUTs, we first annotate the inputs of the sub-netlist with their arrival times by examining the original circuit from which the sub-netlist was extracted. This is carried out by topologically traversing the original circuit from the primary inputs and computing the accumulated delay of the input nodes to the sub-netlist. Similarly, to compute the required time for the outputs of the sub-netlist, we traverse the original circuit in reverse topological order. During the traversal, we annotate the output nodes of the sub-netlist with their required times. The required time of a node n, defined as the maximum delay that does

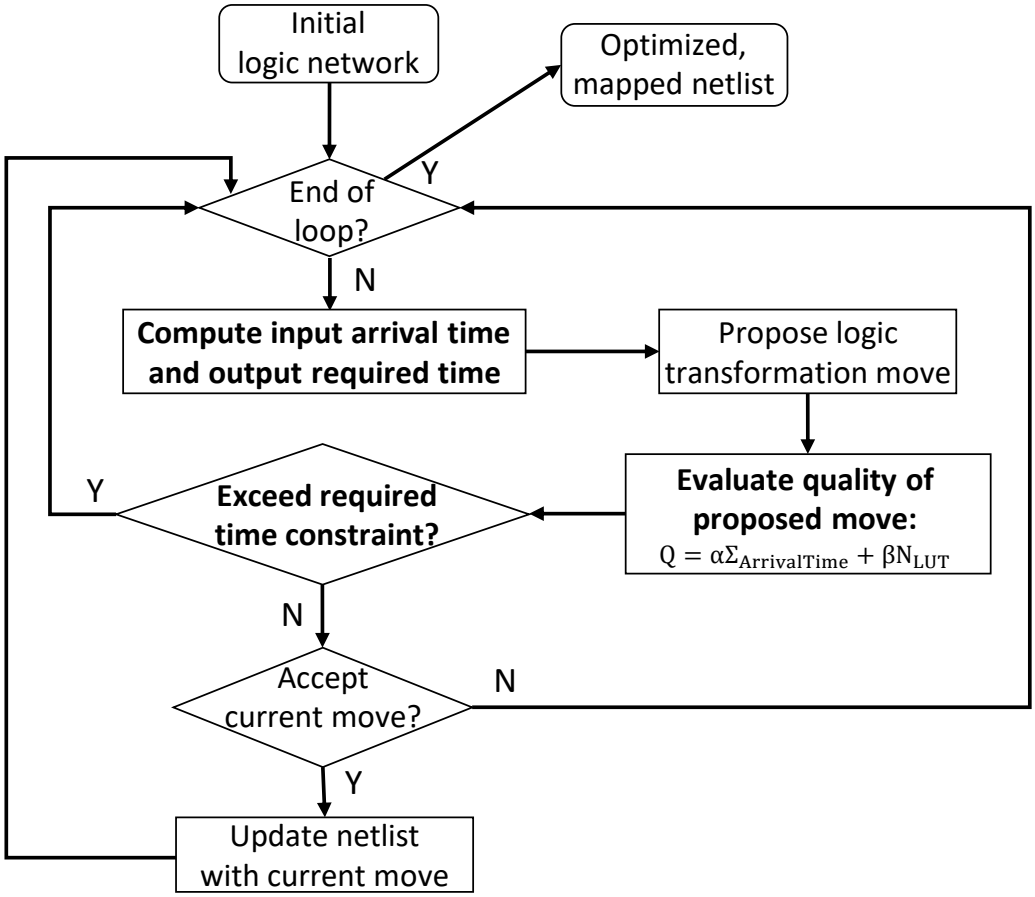


Figure 2.4: **High-level flow chart of the iterative logic transformation routine for depth optimization —** We highlight the steps that differ from the iterative area minimization flow.

not affect the overall delay of the original circuit, can be computed during the reverse topological traversal by examining the set of fanout nodes of node n, denoted as *fanout(n)*. The required time $RT(n)$ of the current node n is then computed as $RT(n) = \min_{i \in fanout(n)} RT(i) - delay(LUT)$. Figure 2.5 shows an example of calculating the arrival and required times of the nodes in a netlist, where we highlight the calculation for the required time of node c.

Optimizing sub-netlists under required time constraint After annotating the sub-netlist with the arrival time and required time information, we iteratively

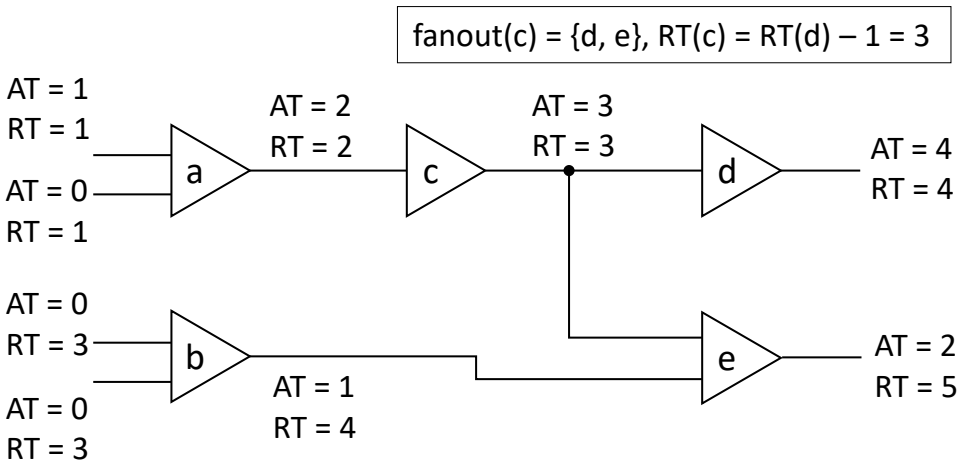


Figure 2.5: **Illustrative example of calculating the arrival and required times of the nodes in a mapped netlist —** `AT` = arrival time; `RT` = required time. Each triangle represents a LUT. Here we assume that each LUT has a delay of 1, and the arrival times of the PIs and the required times of the POs are given.

transform the sub-netlist by extending the flow for area minimization. At each iteration, the proposed transformation heuristically optimizes the sub-netlist. The size and the output levels of the transformed circuit are then measured after technology mapping. If any of the outputs in the transformed sub-netlist has an arrival time exceeding its required time, the proposed move is immediately rejected. Otherwise, we compute the sum of the output arrival times in the transformed sub-netlist, which will be used to determine whether to accept the transformation. The quality of the proposed transformation move Q_{curr} is computed as $Q_{curr} = \alpha\Sigma_{ArrivalTime} + \beta N_{LUT}$, which is a weighted sum of total output arrival times ($\Sigma_{ArrivalTime}$) and the circuit size after technology mapping (N_{LUT}). The weights α and β are hyperparameters that trade off delay optimization effort and area recovery effort. We use Algorithm 1 for deciding whether to accept the proposed transformation using the quality of the proposed transformation Q_{curr} and that of the previous iteration (i.e., Q_{prev}). We note that our approach can potentially be extended to enable multi-objective design space exploration,

where we construct a Pareto-optimal frontier in delay and area. However, in this chapter, we focus on delay minimization and only recover area after an optimized delay is achieved. We use the technique in Section 2.3.2 to dynamically adjust the area recovery effort, which ensures that the final solution maintains an optimized delay during the optimization process.

2.3.2 Dynamic Adjustment of Area Recovery Effort

The values of the hyperparameters α and β in the iterative logic transformation routine controls the tradeoff between optimization efforts for delay minimization and area reduction. In one extreme case where $\alpha = 0$, PIMap reduces the circuit area while satisfying the required time constraint, which is equivalent to solving the delay-constrained area recovery problem. For the other extreme scenario where $\beta = 0$, PIMap tries to minimize the circuit delay without any consideration on the size of the generated circuits.

In PIMap, we adjust the values of α and β at runtime to achieve an optimized strategy for the design at hand. We implement a simple mechanism to automatically adjust the efforts for delay optimization versus area recovery by tuning the values of α and β. Specifically, we start with $\beta = 0$ to minimize circuit delay in the first trial. Every time when the current trial does not improve the circuit delay, we gradually increase the value of β so that PIMap will emphasize on reducing the size of the current circuit. We observe that this strategy achieves the goal of prioritizing delay minimization while reducing the circuit area after reaching an optimized delay value.

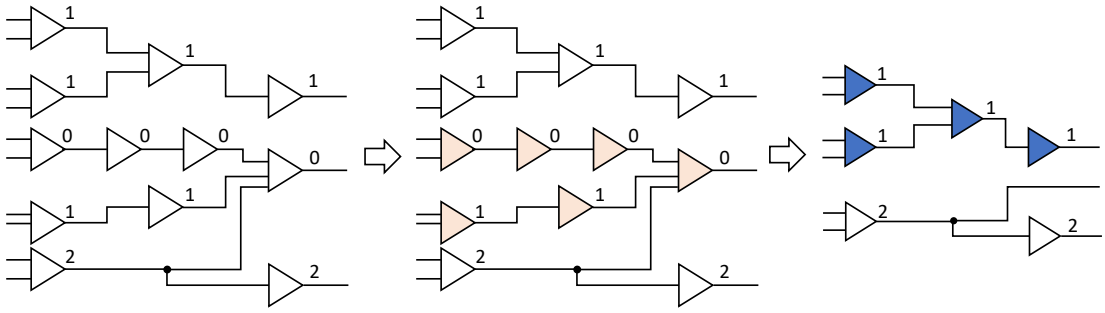


Figure 2.6: **Illustrative example of the delay-critical netlist extraction process** — We extract two sub-netlists with a critical value of one. Each triangle represents a LUT, the number next to each LUT indicates its slack value. The extracted sub-netlists are highlighted with colors.

2.3.3 Delay-Critical Netlist Extraction

For delay optimization, PIMap extends the netlist extraction technique in Section 2.2.2 to focus the optimization efforts on promising sub-netlists that are more likely for delay improvements. To extract delay-critical sub-netlists, PIMap first computes the timing slack for every node in the initial mapped circuit. Starting from the primary output with the smallest timing slack, where ties are broken randomly, PIMap traverses the netlist towards the primary inputs while adding the visited nodes only when its timing slack is no greater than a critical value. In practice, a critical value of three LUT levels works well for generating delay-critical sub-netlists. After extracting the first sub-netlist, PIMap starts over and extracts the next sub-netlist from the remaining nodes in the original netlist. Figure 2.6 uses an example to depict the process of extracting delay-critical netlists with a critical value of one.

We note that the delay-critical netlist extraction algorithm will unlikely extract the same set of sub-netlists in different trials mainly due to the following reasons: (1) the randomized tie-break rule for selecting the starting primary out-

put nodes tends to choose different subsets of primary outputs; (2) the netlist structure changes non-trivially after each trial, so the extracted sub-netlists will differ even when starting from the same primary output.

2.4 Experimental Results

We implement the PIMap techniques in C. We integrate the logic optimization moves and the technology mapper in ABC [10] as a static library into PIMap. Throughout the experiment, we use ABC's native AIG as the gate-level representation. In our experiment, we use PIMap to refine designs that are already optimized by existing technology mappers (e.g., the best known results from the EPFL benchmark suite [2]). In our experiment, the set of logic transformation techniques T consists of three elements: `balance`, `rewrite`, and `refactor`, with a uniform probability distribution $P = \{1/3, 1/3, 1/3\}$. For each design, we execute 40 trials, and each trial contains 100 iterations of mapping-guided logic optimization. For parallelization, we partition the original design to up to 16 sub-netlists, where each sub-netlist contains up to 100 LUTs. We run PIMap on up to eight machines, and each machine has a quad-core Xeon CPU operating at 2.7GHz. We use two well-known benchmark suites to evaluate the effectiveness of PIMap: the 10 largest designs in the MCNC benchmark suite [109], as well as the EPFL benchmark suite [2]. This collection of benchmarks contains a diverse set of designs ranging from common arithmetic units to realistic industrial designs. These designs also greatly differ in size.

Table 2.1: **Area reduction using PIMap on the 10 largest MCNC combinational benchmarks —** `Base` = the baseline designs synthesized using ABC's `compress2rs` script followed by an area-oriented technology mapper (command `if -a -K 6`); `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; `Time` = runtime in seconds; `Improv` = improvement in size between PIMap and the baseline designs.

	Base		5 Trials				10 Trials				40 Trials			
Designs	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
alu4	455	9	425	13	22.3	6.6%	405	15	42.9	11.0%	393	13	168.8	13.6%
apex2	526	12	493	15	22.2	6.3%	488	15	43.1	7.2%	439	17	177.4	16.5%
apex4	568	9	555	13	18.1	2.3%	541	13	38.3	4.8%	526	13	162.4	7.4%
des	631	9	544	8	31.9	13.8%	509	8	62.2	19.3%	477	8	253.0	24.4%
ex1010	606	9	589	11	18.8	2.8%	584	13	39.4	3.6%	556	15	158.5	8.3%
ex5p	332	10	324	11	16.3	2.4%	319	12	34.0	3.9%	304	12	136.9	8.4%
misex3	382	9	352	9	18.6	7.9%	333	10	36.3	12.8%	298	9	153.0	22.0%
pdc	1251	14	1219	19	31.8	2.6%	1200	22	66.6	4.1%	1150	19	266.5	8.1%
seq	627	10	606	12	22.1	3.3%	596	11	43.2	4.9%	567	12	177.0	9.6%
spla	1251	14	1222	18	32.5	2.3%	1191	18	63.8	4.8%	1133	25	250.8	9.4%
geomean						4.8%				7.4%				12.4%

Table 2.2: **Area reduction using PIMap on the EPFL arithmetic benchmarks —** `Base` = the best known results on EPFL benchmarks [2]; `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; `Time` = runtime in seconds; `Improv` = improvement in size between PIMap and the baseline designs.

	Base		5 Trials				10 Trials				40 Trials			
Designs	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
adder	201	73	196	68	19.2	2.5%	196	68	37.7	2.5%	194	66	150.5	3.5%
shifter	512	4	512	4	21.1	0.0%	512	4	41.1	0.0%	512	4	164.5	0.0%
divisor	3813	1542	3636	1490	53.1	4.6%	3527	1431	104.3	7.5%	3331	1277	418.1	12.6%
hyp	44635	4194	44095	4341	195.5	1.2%	43677	4431	394.9	2.1%	42164	4542	1604.3	5.5%
log2	7344	142	7036	133	60.9	4.2%	6904	129	119.8	6.0%	6749	119	491.5	8.1%
max	532	192	525	190	28.1	1.3%	525	190	57.6	1.3%	522	190	222.3	1.9%
mult	5681	120	5184	97	64.6	8.7%	5069	90	133.7	10.8%	4986	86	544.9	12.2%
sine	1347	62	1273	57	40.3	5.5%	1261	57	81.2	6.4%	1235	56	332.7	8.3%
sqrt	3286	1180	3246	1198	52.1	1.2%	3200	1188	103.8	2.6%	3127	1154	412.1	4.8%
square	3800	116	3380	77	94.1	11.1%	3346	77	184.8	11.9%	3281	74	730.3	13.7%
geomean						4.1%				5.2%				7.2%

2.4.1 Unconstrained Area Minimization

Table 2.1 shows the results of unconstrained area minimization for the 10 largest MCNC combinational benchmarks. For this set of benchmarks, we first apply ABC's `compress2rs` logic optimization script targeting area reduction. Based on our experiments with the available ABC synthesis scripts, `compress2rs` achieves the best area results for the majority of the designs. The optimized logic network is then mapped into 6-input LUTs using ABC's area-optimized mapper with command `if -a -K 6`. For PIMap, we record the size, depth, and runtime after 5, 10 and 40 trials. We also report the improvement of LUT counts in the PIMap-optimized designs over the baseline designs.

PIMap is able to reduce the LUT count by 4.8% on average after five trials, and 12.4% after 40 trials. For `des` and `misex3`, PIMap is able to reduce the size by more than 20%, showing the effectiveness of PIMap compared to ABC. The runtime of the 10 benchmarks are similar due to the similar sizes of the designs, averaging around 20 seconds for five trials, and 160 seconds for 40 trials. Although the runtime of PIMap is noticeably higher than existing mappers, which usually take less than a second for designs of similar sizes, we argue that PIMap is still valuable and viable in a high-effort FPGA implementation mode where technology mapping is unlikely the performance bottleneck.

We further apply PIMap to the EPFL arithmetic benchmark suite, and compare our results with the best known mapping records from the EPFL database that are publicly available [2]. Table 2.2 shows the comparison between PIMap and the existing best known results (used as baseline). PIMap is able to improve nine out of the 10 best known mapping results, with an average improvement of 7.2%. Notably, PIMap reduces the LUT count for `divisor`, `mult`, and `square`

by more than 12%. In addition, PIMap improves the depth in eight out of the 10 designs even though it is not intended for depth optimization in this particular use case. We conjecture that existing area-oriented mappers that generate the best known min-area solutions have to make an unnecessary compromise in depth to gain additional area savings. We also note that even for the largest design `hyp` that has more than 44 thousand LUTs, the PIMap runtime remains reasonable, owing to the fact that we optimize multiple small sub-netlists in parallel instead of directly optimizing the entire design.

2.4.2 Scalability of Parallel Optimization

Figure 2.7 shows the scalability of PIMap. In our experiment, we partition an input netlist to up to 16 sub-netlists, each of which contains up to 100 LUTs. We select four large benchmarks in the EPFL benchmark suite, and study the runtime required to achieve a fixed area target. The area target of each benchmark is set to be the area of the PIMap-optimized design using one sub-netlist and 100 trials. In this experiment, we use four parallel threads to optimize one sub-netlist, which requires up to 64 threads in total for the 16 sub-netlists.

As shown Figure 2.7, PIMap scales reasonably well up to 16 sub-netlist partitions across multiple designs. In particular, PIMap scales near-linearly up to eight sub-netlists. With more sub-netlists, the overhead of netlist decomposition and reassembly becomes nontrivial and prevents PIMap from achieving the ideal speedup.

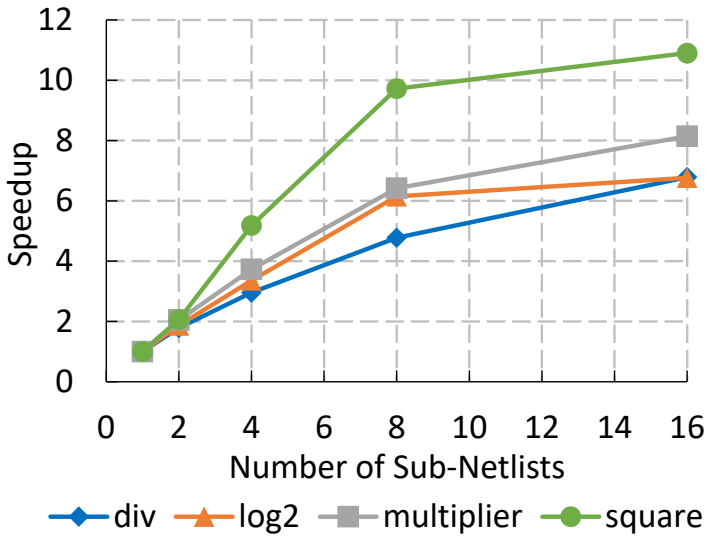


Figure 2.7: **Scalability of the parallel optimization technique —** We use PIMap for area reduction to test the scalability of parallelization. We measure the runtime to achieve a specific area target and plot the speedup in runtime versus number of sub-netlist partitions.

2.4.3 Runtime Breakdown of PIMap

Figure 2.8 shows the relative runtime of the four main steps in PIMap. Sub-netlist generation refers to the step of decomposing the original netlist into sub-netlists. The logic transformation step first proposes a transformation move, then applies the selected move to the network. The LUT mapping step converts the logic networks into LUTs using ABC's built-in technology mapper named `if`. The quality evaluation step calculates the quality of the proposed transformation and decides whether to accept the proposed move.

Not surprisingly, the LUT mapping consumes the largest portion of the runtime, followed by quality evaluation and logic transformation. These three steps together dominate the runtime since they need to be iteratively invoked for many times in each trial (100 in our experiment). The runtime of the sub-netlist generation step is negligible for most of the benchmarks since the BFS-based ex-

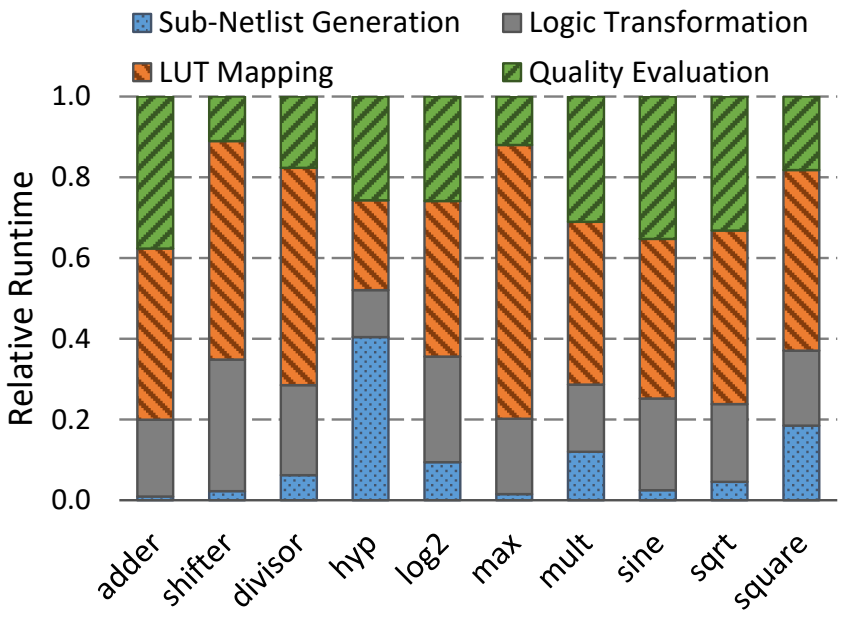


Figure 2.8: **Runtime breakdown of PIMap.**

traction algorithm scales linearly as the size of the netlist. For `hyp`, the runtime of sub-netlist generation is noticeably higher than the other designs since it is significantly larger than other designs. Nevertheless, the runtime of sub-netlist generation for `hyp` is still on the same order of the other steps.

2.4.4 Impact of Sub-Netlist Size on PIMap Runtime

Figure 2.9 shows the impact of the sub-netlist size on the PIMap runtime to achieve a fixed area target, defined as the area after 100 trials using a sub-netlist size of 20 LUTs. We partition the designs up to 16 sub-netlists. For smaller designs that do not admit 16 sub-netlists, we partition them into as many sub-netlists as feasible.

The runtime in Figure 2.9 is normalized to the longest runtime of the corre-

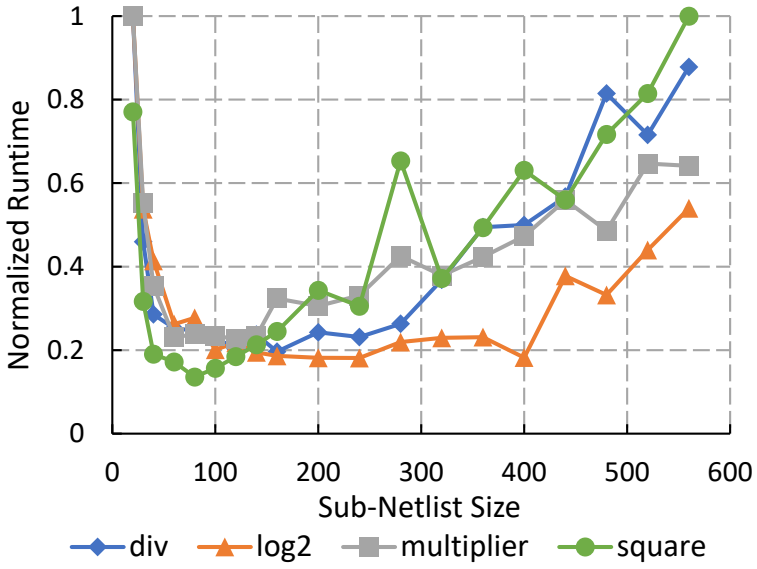


Figure 2.9: **Impact of sub-netlist size on PIMap runtime.**

sponding design.

We observe that across the four benchmarks, the runtime inflection point is around the size of 100 LUTs. With smaller sub-netlists, each PIMap optimization thread runs faster, but overall progress may be slow since each sub-netlist only covers a small fraction of the entire design.

The size of the sub-netlists also influences the quality of the PIMap optimization. We run PIMap targeting area minimization on two benchmarks (`div` and `sqrt`) using sub-netlist sizes ranging from 5 to 3000 LUTs. We partition the design up to 16 sub-netlists. For smaller designs that do not admit 16 sub-netlists, we partition them into as many sub-netlists as feasible. Figure 2.10 shows the reduction of LUTs as a function of the number of trials. We observe that a sub-netlist size of 100 to 200 LUTs leads to the best results. For small sub-netlists with only 5 to 20 LUTs, the results are suboptimal due to the limited optimization opportunity constrained by the small sub-netlists. On the other hand, an

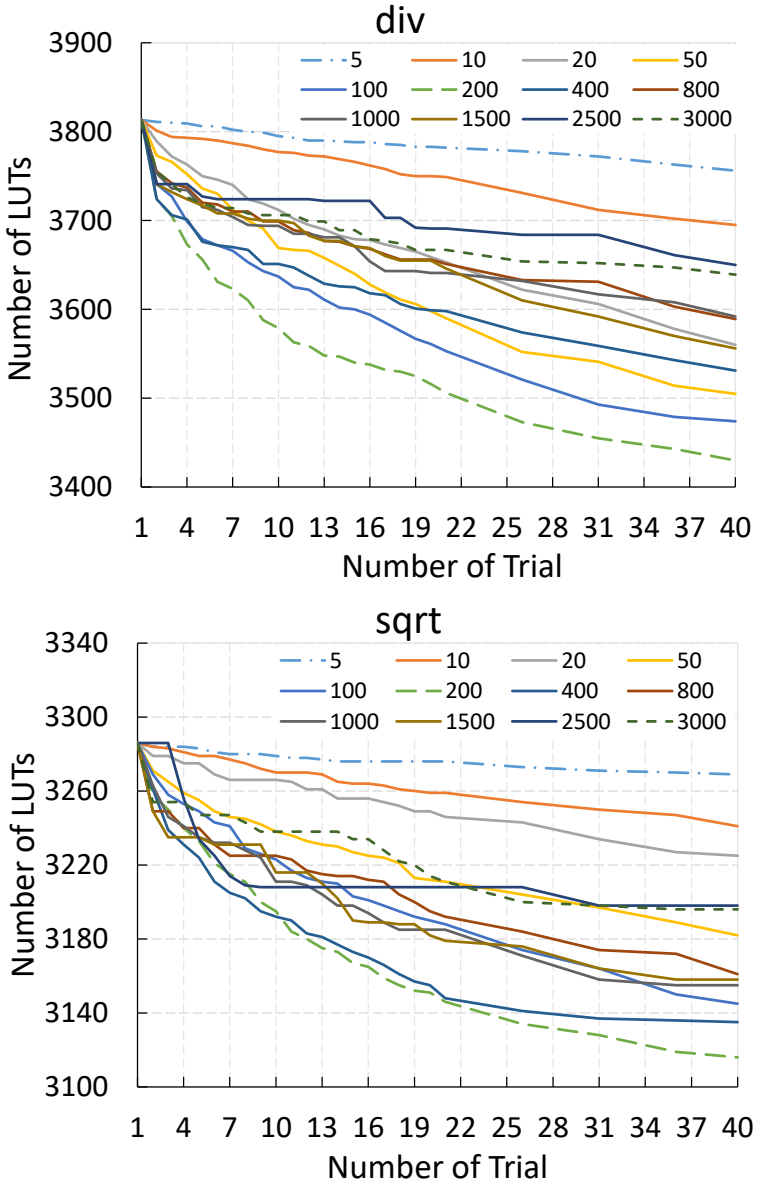


Figure 2.10: **Impact of sub-netlist size on PIMap quality —** We run PIMap on the same design with varying sub-netlist sizes ranging from 5 to 3000 LUTs. The legend indicates different sub-netlist sizes.

overly large sub-netlist results in very similar network structures across trials, which also reduces the optimization opportunity between trials.

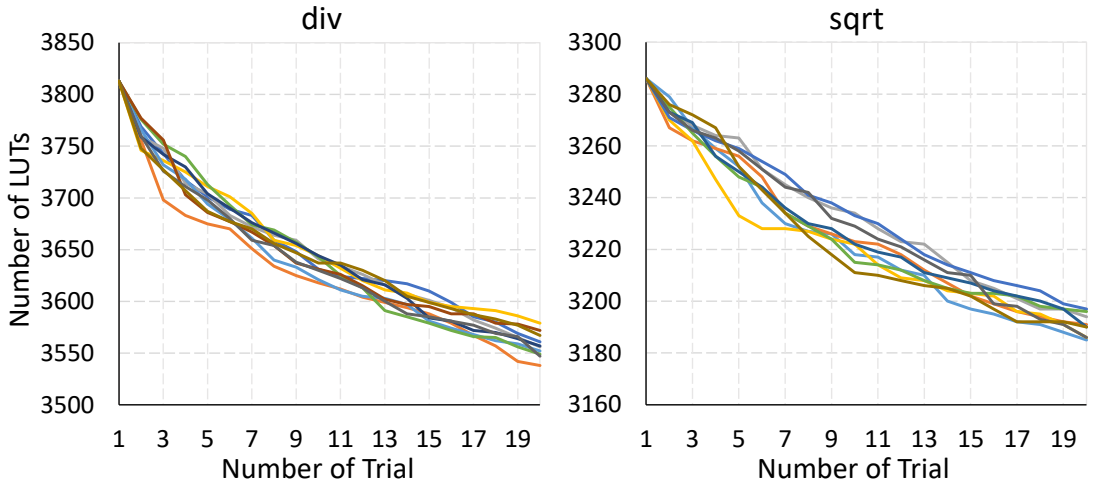


Figure 2.11: **PIMap convergence experiments over independent runs —** For each design, we conduct 10 runs of identical experiments to investigate the effects of randomness on the solution quality.

2.4.5 Convergence Experiment over Independent Runs

Randomized selection of logic transformations and the randomized repartitioning technique are the two sources of randomness during the PIMap optimization process. In Figure 2.11, we study the effects of the randomness on the solution quality. For each design, we conduct 10 identical runs of experiment, and measure the reduction in LUT count. For each run, we partition the netlist into eight sub-netlists, where each sub-netlist contains 100 LUTs. We observe that the independent experiments achieve similar quality of results despite the randomness in the optimization process. For each design, the maximum difference in the solution quality is within 1% across the 10 runs. We also observe that the final netlists after the optimization are similar in structure, showing that PIMap empirically converges to an optimized final solution regardless of the randomness during the optimization process.

Table 2.3: **Area reduction using PIMap with 10 second runtime limit —** `Base` = the best known results on EPFL benchmarks [2]; `PIMap` = solution of PIMap after 10 seconds. We highlight the designs that are improved by PIMap.

	Base		PIMap	
Designs	Size	Dpt	Size	Dpt
adder	201	73	**197**	69
shifter	512	4	512	4
divisor	3813	1542	**3787**	1536
hyp	44635	4194	44635	4194
log2	7344	142	**7305**	144
max	532	192	**526**	190
mult	5681	120	**5594**	118
sine	1347	62	**1309**	62
sqrt	3286	1180	**3279**	1181
square	3800	116	**3675**	102

2.4.6 Area Reduction under a Tight Runtime Limit

Table 2.3 shows the performance of PIMap under a tight runtime limit, which is set to be 10 seconds. In this case, PIMap achieves less area savings but still manages to improve the best-known mapping results in eight out of the 10 EPFL benchmarks.

2.4.7 LUT Count vs. Gate Count Reduction

Figure 2.12 shows the LUT count and the corresponding gate count in the AIG of the same design during the optimization process in PIMap, normalized to their initial values. For the four benchmarks, the LUT count decreases as the number of trials increases. However, we observe an opposite trend in gate count during the optimization, which agrees with our correlation study in Section 1.2.

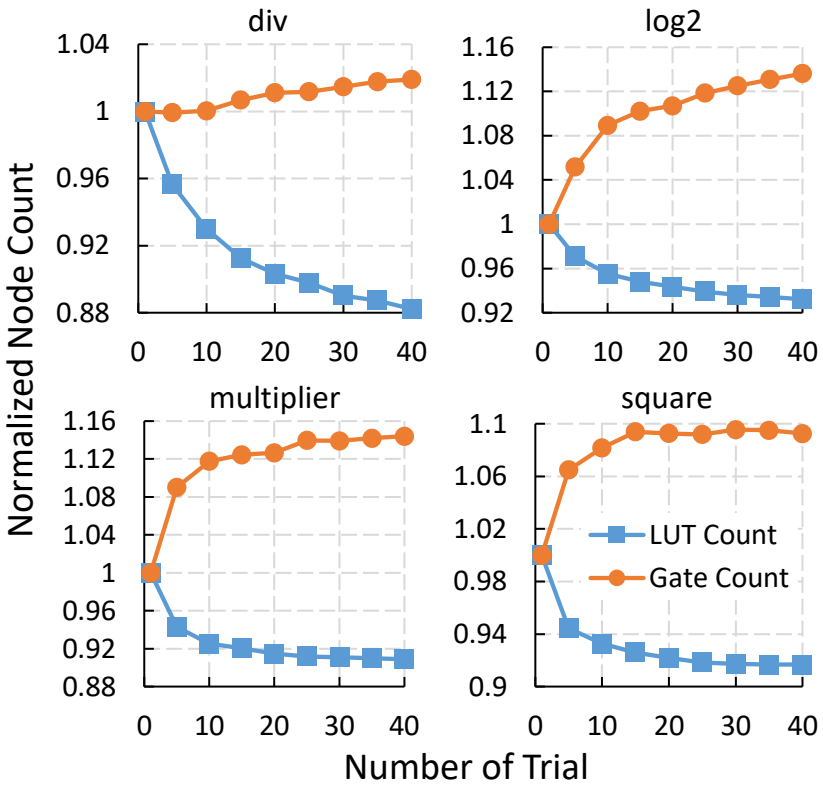


Figure 2.12: **Relation between LUT count and AIG gate count at various design points of the same design.**

2.4.8 Delay Optimization for Combinational Circuits

Table 2.4 shows the results of applying PIMap delay optimization to the 10 largest MCNC combinational benchmarks [109]. The baseline designs are generated using ABC's delay-optimizing script `resyn2` and mapped to 6-LUTs using delay-oriented technology mapper (`if -K 6`). PIMap is able to reduce the maximum LUT depth by one in three designs (`ex5p`, `pdc`, and `spla`). Out of the remaining seven designs where PIMap does not improve the maximum depth, we observe that PIMap can improve the average depth of the designs in three cases (`alu4`, `misex3`, and `seq`), where the average depth is measured as the number of LUT levels averaged over all the primary outputs in the design. Reducing the average depth of a design can potentially benefit the downstream

Table 2.4: **Delay optimization using PIMap on the 10 largest MCNC combinational benchmarks —** `PI/PO` = number of primary inputs and primary outputs in the design; `Base` = the baseline designs synthesized using ABC's delay-optimizing `resyn2` script followed by an delay-oriented technology mapper (command `if -K 6`); `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Max Dpt` = depth of the design defined as the highest LUT level; `Avg Dpt` = average depth of the design defined as the average LUT level over all primary outputs of the design; `Time` = runtime in seconds. We highlight the designs whose maximum or average depth is improved by PIMap.

		Base			5 Trials				40 Trials			
Designs	PI/PO	Size	Max Dpt	Avg Dpt	Size	Max Dpt	Avg Dpt	Time	Size	Max Dpt	Avg Dpt	Time
alu4	14/8	511	5	4.50	492	5	**4.38**	10.2	461	5	**4.38**	79.9
apex2	39/3	674	6	6.00	647	6	6.00	10.4	575	6	6.00	78.6
apex4	9/19	588	5	4.68	575	5	4.68	11.0	575	5	4.68	84.1
des	256/245	818	5	3.23	818	5	3.23	12.1	818	5	3.23	98.8
ex1010	10/10	655	5	5.00	655	5	5.00	11.2	655	5	5.00	89.9
ex5p	8/63	351	5	3.67	404	**4**	**3.03**	11.0	404	**4**	**2.98**	81.5
misex3	14/14	443	5	5.00	413	5	**4.93**	10.2	387	5	**4.93**	77.9
pdc	16/40	1431	7	5.93	1713	**6**	**5.05**	11.6	1713	**6**	**5.05**	89.2
seq	41/35	693	5	4.49	669	5	**4.40**	10.0	669	5	**4.34**	80.8
spla	16/46	1392	7	5.63	1599	**6**	**4.83**	11.6	1599	**6**	**4.78**	88.9

CAD flow for generating faster circuit after physical design. While the delay algorithm prioritizes depth reduction, the impact of logic transformations on area is also considered during the optimization flow. Consequently, for the seven designs that PIMap does not improve depth, PIMap reduces the size of the circuits in five designs. The size of the remaining two designs (`des` and `ex1010`) stay unchanged.

We further apply PIMap to the latest depth records (version 2017.1) of the EPFL benchmark suite [2] to further improve the quality of the existing circuits. These depth records are used as the starting point for PIMap optimization. For each design, we execute PIMap for 100 trials. The runtime per trial for each design is almost identical to the per-trial runtime shown in Table 2.2, since the additional steps for depth optimization require negligible runtime. The results are shown in Table 2.5. According to the rule from the EPFL benchmark competition[2], PIMap improves the existing best records in 13 out of the 20 designs by reducing the circuit area while maintaining the same depth of the previous records. Noticeably, PIMap achieves significant area reduction while maintaining the best-known depths (e.g., 7.8%, 7.0% and 6.3% area reductions for `cavlc`, `router` and `sine`, respectively). In addition, two designs in Table 2.5 (`mem ctrl` and `decoder`) also achieve the best-known area while maintaining the best-known depth. The design `router` optimized by PIMap is larger than its best-area counterpart by only one LUT (53 LUTs vs. 52 LUTs), but improves the depth of the best-area version by two (4 levels vs. 6 levels).

[2]The EPFL delay records rank designs by their depth after mapping to 6-input LUTs. Designs with the same depth are further ranked by their sizes (measured as the number of 6-input LUTs).

Table 2.5: **Comparison with depth records on EPFL benchmarks —** `Best Known Record` = the best known delay results on EPFL benchmarks [2]; `PIMap` = solution of PIMap after 100 trials; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level. We highlight the designs that are improved by PIMap.

	Best Known Record		PIMap	
Designs	Size	Dpt	Size	Dpt
adder	511	5	511	5
shifter	512	4	512	4
divisor	47964	230	47964	230
hyp	146302	573	**145236**	573
log2	9218	55	9218	55
max	882	10	882	10
mult	8215	28	**8105**	28
sine	1801	30	**1690**	30
sqrt	11680	254	**11391**	254
square	4038	11	**3992**	11
arbiter	2884	5	**2871**	5
alu ctrl	29	2	**27**	2
cavlc	115	4	**106**	4
decoder	272	2	**270**	2
i2c controller	244	3	244	3
int2float	41	3	**40**	3
mem ctrl	2490	7	**2486**	7
priority	157	4	157	4
router	57	4	**53**	4
voter	1469	12	**1423**	12

2.4.9 Delay Optimization for Sequential Circuit with Retiming

The optimization techniques in PIMap can be naturally extended to handle sequential circuits by treating the inputs and outputs of the registers as primary outputs and primary inputs of the combinational blocks, respectively. Table 2.6 shows the PIMap delay optimization results on 10 large MCNC sequential circuits that combine the retiming technique to further improve the delay. Retiming is a common sequential circuit optimization technique that adjusts the locations of the registers to optimize the overall circuit delay [33]. We observe

Table 2.6: **Depth reduction for sequential circuits with retiming on 10 large MCNC sequential benchmarks —** `Base` = the baseline designs synthesized using ABC's `resyn2` script followed by a depth-oriented technology mapper (command `if -K 6`); `PIMap` = solution of PIMap after 10 trials without applying retiming; `Base + Retime` = results after applying retiming on `Base`; `PIMap + Retime` = solution of PIMap after 10 trials, where retiming is applied after each trial; `Reg` = number of registers in the design; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; We highlight the designs whose depth is improved by PIMap.

	Base			PIMap			Base + Retime			PIMap + Retime		
Designs	Reg	Size	Dpt	Reg	Size	Dpt	Reg	Size	Dpt	Reg	Size	Dpt
b14	245	1272	13	245	1295	**11**	448	1461	8	502	1496	8
bigkey	224	579	3	224	575	3	672	1094	2	672	1094	2
clma	33	2965	9	33	2879	9	41	3011	9	42	2887	**8**
frisc	886	1776	14	886	1774	**9**	1276	2199	7	1288	2257	**6**
fsm8_16_13	8	386	5	8	386	5	8	390	5	8	477	**4**
mm30a	90	266	25	90	273	25	90	266	25	90	340	**18**
s38417	1462	2535	7	1462	2535	7	1462	2547	7	1462	2497	7
s38584.1	1260	2322	6	1260	2279	6	1267	2394	5	1267	2361	5
s9234.1	135	289	5	135	293	5	189	331	4	186	326	4
sort	136	387	14	136	387	**13**	146	410	13	146	448	**12**

that the logic transformations in PIMap generally not only optimize the combinational blocks between register boundaries, but also present optimization opportunities for retiming to achieve better quality. In Table 2.6, the baseline designs are generated using ABC's `resyn2` script combined with delay-optimizing technology mapper `if -K 6`. For comparison, we also measure the quality of the circuits after applying PIMap without retiming, and after applying retiming on the baseline designs. Both versions of the PIMap optimized designs are generated with 10 trials. For the case of PIMap with retiming, we apply ABC's `retime` command at the end of each trial. Without applying retiming, PIMap is able to improve the depths of three designs when compared to the baseline circuits. With retiming enabled, PIMap improves the depth of the designs in five out of the 10 designs when compared to the baseline with retiming, demonstrating the effectiveness of PIMap delay optimization on sequential circuits.

2.5 Related Work

Mishchenko, et al. [73] describe a number of efficient rewriting techniques on AIGs, which serve as the basis for the logic transformations used in this work. The majority-inverter graph (MIG) proposed by Amarú, et al. [3] provides an alternative logic representation using three-input majority nodes and regular/-complemented edges. MIG is shown to be beneficial for improving mapping quality in a number of cases. This is complementary to PIMap, since our iterative improvement framework is agnostic to logic representations.

Yang, et al. [110] propose a new way of logic synthesis by maintaining a pre-

computed library of optimal or near-optimal circuits for small practical functions. Their logic synthesis flow matches and replaces small circuit components in a new design to the elements in the precomputed library. However, this approach can only find optimal or near-optimal solution for small functions with no more than 12 inputs, and become sub-optimal for functions with more inputs. PIMap is orthogonal to [110] and it is not limited by the input size of the sub-netlist. It is also possible to incorporate Boolean matching techniques as new transformation moves in our iterative improvement framework.

STOKE [88] uses stochastic search to optimize x86 programs by randomly rewriting the x86 assembly instructions. Both STOKE and our approach randomly propose transformations using MCMC sampling to explore a large design space. Besides the different application domains, our work differs from STOKE in two major aspects: (1) STOKE focuses on using local moves that modify a single instruction at a time, while we make use of the logic rewriting techniques applied to multiple nodes in the network; (2) STOKE can only handle small programs with around one hundred instructions. In contrast, PIMap makes use of parallel optimization to effectively handle much larger circuits with tens of thousands LUTs.

2.6 Conclusions

We propose PIMap, a parallelized iterative improvement framework for area-oriented FPGA technology mapping. PIMap iteratively proposes logic transformation moves to optimize an input logic network for LUT mapping, and uses the actual mapping result to evaluate the quality of a proposed move. To

improve the runtime, PIMap decomposes a large circuit netlist into multiple smaller sub-netlists, and optimizes them in parallel across different machines. Experimental results demonstrate significantly improvement in mapping quality for both unconstrained area optimization and depth-constrained area optimization compared to the state-of-the-art technology mappers. As a future direction, we plan to investigate global restructuring techniques on the logic network to further improve the quality of PIMap.

CHAPTER 3

SCALS: DATA-DRIVEN APPROXIMATE LOGIC SYNTHESIS

Approximate computing is an emerging design paradigm aiming to largely improve the quality of result (QoR) such as performance, area and efficiency at the cost of carefully-controlled errors [45]. Representative application domains include data mining and machine learning, where infrequent errors at the outputs may not significantly degrade user experience, and image/signal processing, where errors of small magnitude may not be perceivable by the end users [17, 100].

This chapter focuses on approximate logic synthesis, a key step to automate the process of creating approximate circuits based on an exact logic function. The problem of approximate logic synthesis can be viewed as the natural extension of the traditional logic synthesis problem presented in Chapter 2. In approximate logic synthesis, we relax the synthesis constraints so that inexact outputs are allowed. Existing approaches to approximate logic synthesis usually simplify a logic network in a technology-independent manner by removing logic gates or connections between gates, subject to constraints on error rate and/or error magnitude [69, 70, 91].

We use Figure 3.1 as a motivational example to illustrate a typical approximate logic synthesis flow and its drawbacks. Given an initial design as shown in Figure 3.1 with an error rate constraint of 10%, one possible simplification that an existing synthesis algorithm would make is to remove gate g_2. This move decreases both the gate count and output level of the logic network by one, but generates incorrect results for two input patterns $a = 0/1, b = 0, c = 1, d = 1$ (out of the 16 input combinations in total). Synthesis methods such as in [105, 106]

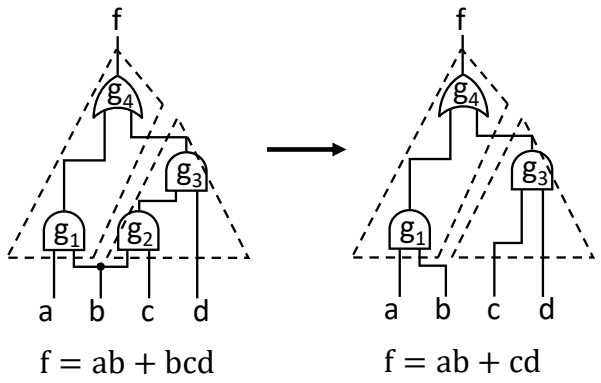


Figure 3.1: **An example illustrating the drawback of existing approximate synthesis technique —** The approximate synthesis algorithm removes gate g_2, reducing the gate count and output level both by one. However, this approximation does not reduce the area or depth of the final netlist after mapping to 3-input LUTs (as highlighted with dashes).

then sample N number of input vectors uniformly and simulate the circuit with these input vectors to estimate the error rate, where N is small relative to the total number of possible input combinations.[1] If none of the sampled input vectors leads to the two erroneous outputs[2], the synthesis would incorrectly conclude that the generated design satisfies the 10% error rate constraint.

In addition to inaccurate characterization of the error behavior, the current approach may also fail to improve the actual quality of results of the circuit after technology mapping. With the example in Figure 3.1, neither the area nor the timing is improved after the approximation. We attribute such unfavorable outcome to three major drawbacks of the conventional approximate logic synthesis techniques:

Disconnect from downstream flow While existing approaches are effective in simplifying the gate-level logic network, they usually do not take into account

[1] Although it is possible to exhaustively test all input patterns in this small example, exhaustive testing quickly becomes infeasible for moderately large circuits.

[2] Under uniform sampling, the likelihood of observing such an event is $(14/16)^N$, which is significant for small values of N.

the impact of logic simplification on the QoR after mapping to a specific technology library, such as lookup tables (LUTs) for FPGAs.

Misrepresentation of realistic input distributions Realistic datasets rarely follow uniform distribution. Using test vectors drawn from uniform distribution, such as in [105, 106], can lead to incorrect conclusions on error metrics. In addition, synthesis techniques that explicitly rely on this assumption of input distribution will not work for other types of input distributions.

Lack of statistical rigorousness Using random samples to measure the error metrics of an approximate design is inherently a statistical process. In the language of statistical inference, existing methods of equating sampled error behavior with the true error behavior fail to distinguish the difference between sample statistics and population statistics. For example, although the sample mean of a design's error magnitude correlates with its population mean, they are in general not equal to each other due to statistical noise. Capturing such noise through the lens of statistical testing is crucial in assuring a high-confidence evaluation of the error metrics.

In addition, measuring the errors of reasonably large circuits can only be done using a small subset of randomly sampled input vectors. This inherently statistical process can potentially lead to a measured error rate that deviates significantly from the true error rate due to statistical noise. For the example in Figure 3.1, an extreme case happens when the sampled test vectors do not contain any input patterns (i.e., $a = 0/1, b = 0, c = 1, d = 1$ for the current example) that would cause incorrect outputs, leading to a measured error rate of 0% although the generated design is in fact inexact.

To overcome these obstacles, we propose a statistically certified approximate logic synthesis (SCALS) framework, which extends PIMap [67], a state-of-the-art FPGA logic synthesis and mapping framework to generate approximate designs. Following the techniques in PIMap, SCALS couples logic simplification with technology mapping to iteratively simplify the circuit. During the synthesis process, SCALS continuously monitors the quality of the intermediate design points using the technique of statistical testing, leading to a final approximate circuit that adheres to user-specified error constraints.

Our primary technical contributions are as follows:

- We are the first to apply statistical testing techniques to approximate logic synthesis to generate approximate designs with user-specified statistical guarantees.
- We propose a generic approximate logic synthesis framework that can effectively handle various input distributions, error metrics and technology targets.
- We show that our approach achieves better QoR than existing synthesis techniques for both ASIC and FPGA targets while providing statistical guarantees on the error metrics.

3.1 SCALS Techniques

In this section we first describe the approximate logic synthesis problem. We then introduce our proposed techniques on generating approximate logic and hypothesis testing of the specified error metrics.

3.1.1 Problem Formulation

We study the problem of synthesizing approximate designs with user-specified error constraints under a given probability distribution of the input values. Specifically, given a combinational logic network composed of technology-independent logic gates. SCALS minimizes the area and/or delay of the generated design after mapping to a specific technology library (e.g., for LUT-based FPGAs).

For error constraints, we focus on two representative error metrics, error rate and mean relative error magnitude, although our framework is generic enough to handle other types of error metrics as well. Error rate (ER) is defined as the probability that the approximate circuit generates incorrect outputs when the input test vectors are drawn randomly from a specific input distribution. Mean relative error magnitude (MREM) measures the population mean of the error magnitude relative to the exact output.

3.1.2 Overall Flow

SCALS extends the PIMap flow [67] to approximate logic synthesis. Figure 3.2 shows the overall flow of SCALS, where the modifications to PIMap and additional steps in SCALS are highlighted. Figure 3.3 details the iterative logic optimization step in SCALS. Starting with an initial gate-level logic network, we first map it to the targeted technology. We then use the sub-netlist extraction algorithm in PIMap to exact a user-specified number of sub-netlists, each containing a predefined number of LUTs. Each extracted sub-netlist is then independently optimized in parallel using the iterative logic optimization routine.

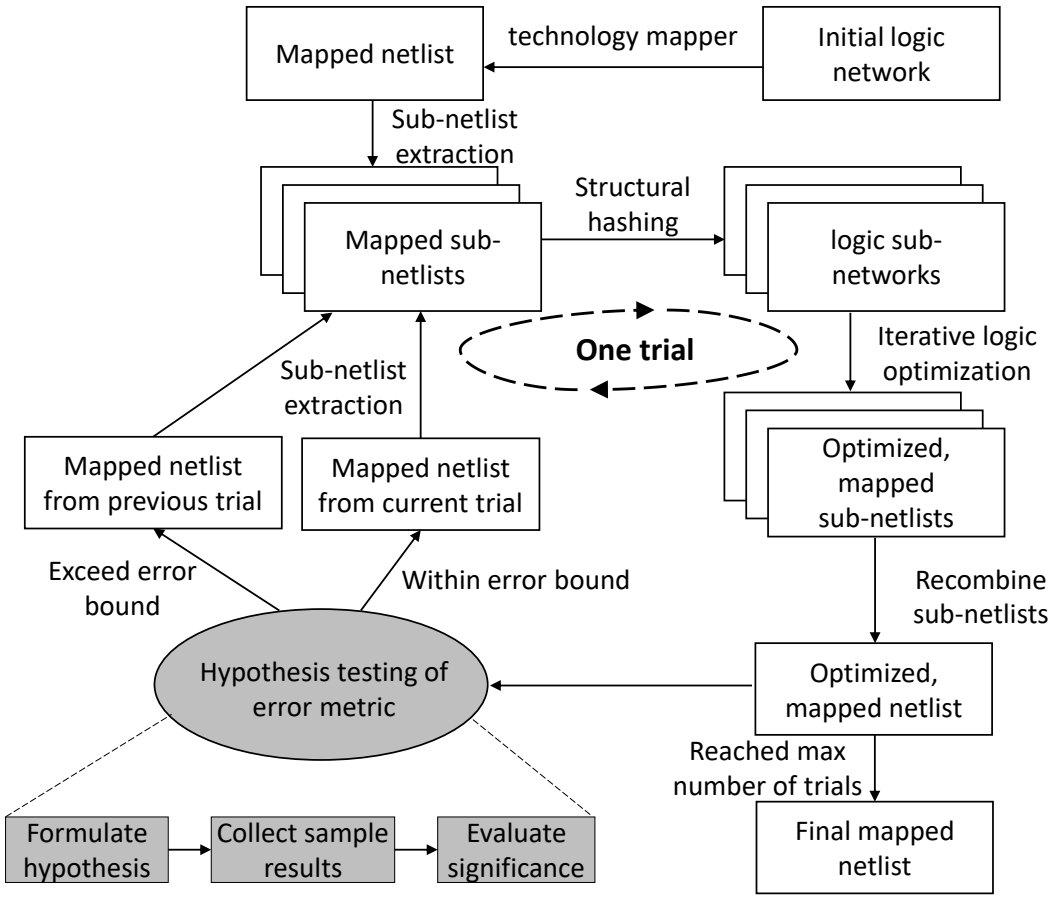


Figure 3.2: **Overall flow of SCALS.**

After recombining the optimized sub-netlists, we evaluate the generated design using statistical hypothesis testing. If the generated design satisfies the error requirement, SCALS accepts the design and uses it for the next trial. Otherwise, SCALS discards the current design and proceeds to the next trial using the design from the previous trial.

3.1.3 Iterative Logic Optimization

SCALS uses a collection of logic transformation moves denoted as the set $T = E \cup A$. For each logic transformation move i in T, we associate it with

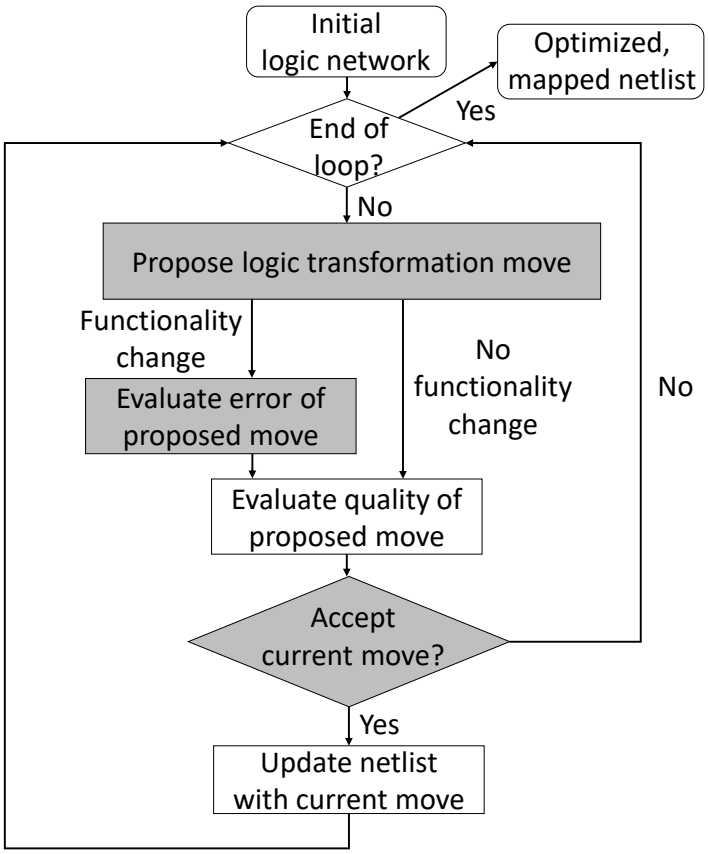


Figure 3.3: **The iterative logic optimization step in SCALS.**

a probability p_i. During each iteration of the iterative logic optimization step, we select one logic transformation i from T with probability p_i.

Exact transforms E is the set of logic transformations that do not alter the functionality of the input design. The set of exact logic transformations contain three commonly used moves, i.e., $E = \{\texttt{balance}, \texttt{rewrite}, \texttt{refactor}\}$. These transformations either balance the logic depths of difference paths in the logic network, or reduce the gate count in the network by logic rewriting [73].

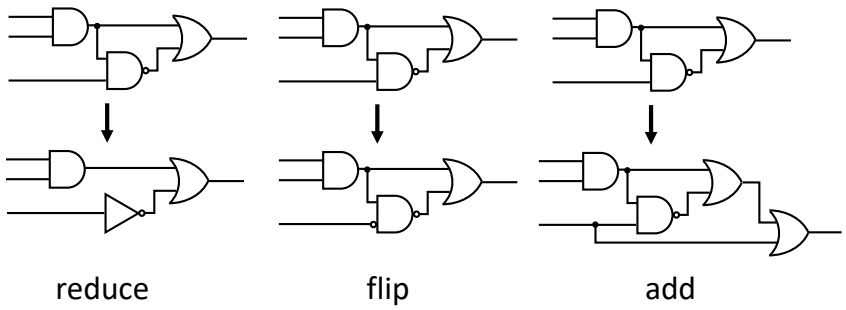


Figure 3.4: **Illustration of approximate transformations.**

Approximate transforms A is the set of logic transformations that simplify the logic network but may generate incorrect outputs. In SCALS, this set includes three types of moves, i.e., A ={`reduce`, `flip`, `add`}. Figure 3.4 illustrates the effects of the three approximate logic transformations. The `reduce` transformation randomly selects one logic gate in the logic network and removes a randomly-selected fanin of the logic gate. If the selected logic gate has only one fanin before removal, then the logic gate itself will be removed, with its fanin node directly connected to the fanouts of the original logic gate. Similar to `reduce`, the `flip` transformation randomly selects one logic gate in the logic network and inverts one of its randomly-selected fanins. Finally, the `add` transformation adds a two-input logic gate with randomly-selected functionality to the logic work, where its two fanin nodes and one fanout node are randomly selected from the existing nodes in the logic network.

Evaluating and accepting a transformation After applying the selected logic transformation to the logic network, SCALS immediately maps the logic network to the targeted technology, and measures the area, denoted as $Area_m$. If an approximate logic transformation is applied, SCALS also estimates the impact of the approximation on the primary outputs through logic simulation. After obtaining the post-mapping area $Area_m$ and the error metric EM, SCALS

calculates a quality metric of the current transformation as a weighted sum of $Area_m$ and EM, i.e., $Q_{curr} = \alpha \cdot Area_m + \beta \cdot EM$, where Q_{curr} is then compared with the quality metric from the previous iteration (i.e., Q_{prev}). Specifically, we use the Markov Chain Monte Carlo (MCMC) method to probabilistically determine whether to accept the proposed move [41]. In particular, we employ the Metropolis-Hastings algorithm for calculating the acceptance probability [47]. The Metropolis-Hastings algorithm dictates that if the quality of the current move is better than the previous one, we accept the current move unconditionally. Otherwise, we accept the move with a probability of $e^{-\gamma(Q_{curr}/Q_{prev})}$, which decreases exponentially as Q_{curr} increases.

Since the iterative logic optimization routine operates on extracted sub-netlists, it is important to be able to estimate error behaviors at the primary outputs from local outputs of the sub-netlists. We use the following procedure to achieve this goal. Using input vectors drawn from the user-specified input distribution, SCALS first simulates the entire design to obtain a set of test vectors for the nodes serving as the inputs to the sub-netlists. These test vectors will then be used to simulate all the sub-netlists in parallel, and generate local outputs for each sub-netlist. To estimate whether a particular primary output O_i is correct, we go through each of the transitive fanins of O_i. If any of these transitive fanins is an output of an extracted sub-netlist S_i, and S_i generates incorrect result during simulation of the sub-netlist, then we conservatively infer that O_i is incorrect. We then use this information to calculate the error metrics (EM) such as ER and MREM at the primary outputs. We note that this is a conservative estimation for O_i due to the possible scenario where the error at S_i is not observable at O_i given the specific test pattern. Nonetheless, this error estimation scheme provides a quick way of inferring global error behavior from

sub-netlists without the need of simulating the entire design every iteration.

3.1.4 Hypothesis Testing of Error Constraint

At the end of each trial, SCALS evaluates the error metric of the generated approximate design using statistical hypothesis testing [103]. Given an error metric EM together with the constraint $EM \leq C$, we formulate the null hypothesis $H_0 : EM > C$ and the alternative hypothesis $H_1 : EM \leq C$. To show that the error metric stays within the constraint, the null hypothesis needs to be rejected under a user-specified confidence level CL. After selecting an appropriate test statistic for the error metric, SCALS generates N number of samples by simulating the approximate design using input vectors drawn from the corresponding input distribution. Using the test statistic and observed samples, we evaluate the probability (P-value) of observing the output samples assuming the null hypothesis holds true. Finally, SCALS makes conclusion on the null/alternative hypotheses based on the test outcome.

Testing error rate constraint SCALS uses the binomial test [103] to examine the error rate of an approximate design. Given a hypothesis that an approximate design has an error rate of p under certain input distribution, SCALS samples N outputs from the circuit using independently-drawn input vectors and compare them with the expected correct outputs. We denote the observed number of incorrect outputs as n. If the null hypothesis is true, then the number of incorrect outputs in the N output samples should follow the binomial distribution, i.e., $n \sim B(N, p)$. SCALS evaluates the P-value of the observed event, and determines the outcome of the test by comparing the P-value and CL.

Testing mean relative error magnitude constraint In statistical inference, a T-test [103] is used for testing the population mean with unknown variance. Given a total of N samples with a sample mean $\bar{X}$, a sample standard deviation S and the population mean μ to be tested, the test statistic $T = \frac{\bar{X}-\mu}{S/\sqrt{N}}$ follows the T-distribution [103], i.e., $T(t) \sim \frac{\Gamma(\frac{v+1}{2})}{\sqrt{v\pi}\Gamma(\frac{v}{2})}(1+\frac{t^2}{v})^{-\frac{v+1}{2}}$, where Γ is the gamma function, and v is the number of degrees of freedom ($v = N-1$). SCALS uses the T-test for testing a mean relative error magnitude constraint. Similar to testing the error rate, SCALS first samples the relative error magnitude using N input vectors drawn from the user-supplied input distribution. SCALS then calculates the P-value using the test statistic and the observed samples. Based on the calculated P-value and CL, SCALS either accepts or rejects the null hypothesis.

Extension to other error metrics While SCALS focuses on error rate constraint and mean relative error magnitude constraint, we can extend SCALS to handle other types of error constraint by drawing connection between the specific error metric and its corresponding test statistic. For example, another interesting test is on whether a given approximate design generates unbiased outputs under certain input distribution. Unbiased designs are particularly useful for applications where an approximate module is used repeatedly because the errors could potentially cancel each other out. Such a test can be formulated as testing whether the population mean of the error magnitude is equal to zero, which can be done using the T-test as detailed above. In scenarios where we are interested in constraining the variance of the error, we can use χ^2 test [103] to examine whether an error metric satisfies a constraint on the population variance.

3.2 Experimental Results

We implement the SCALS techniques in C as extensions to the ABC logic synthesis framework [10]. We target mapping to FPGA LUTs using the EPFL combinational benchmark suite [2]. We report the synthesis results under four representative input distributions including uniform, Gaussian, exponential, and bimodal distributions. For each design, SCALS runs for 20 trials, and each trial contains 100 iterations of the iterative logic optimization routine. Each hypothesis testing step uses a sample size of 10000 test vectors to determine the validity of the hypotheses. We partition the original design to up to 16 sub-netlists, where each sub-netlist contains up to 100 LUTs. We run our experiments on up to eight machines, each with a quad-core Xeon CPU operating at 2.7GHz.

3.2.1 Arithmetic Circuit under Relative Error Magnitude Constraint

Table 3.1 shows the area and depth comparisons with the exact counterparts for the arithmetic benchmarks in the EPFL benchmark suite [2] under mean relative error magnitude constraint for various input distributions. The baseline designs are the best-known 6-LUT mapping results from the EPFL records [2]. As an example to demonstrate the effectiveness of SCALS, we enforce a mean relative error magnitude constraint of $\frac{1}{2^9}$ ($\approx 0.2\%$). All designs pass hypothesis testing on error magnitude for a confidence level of 0.95. We observe that some designs (e.g., `hyp`, `sqrt`, and `square`) are highly error tolerate, requiring only 10% to 15% of the area of the original designs to meet the error constraint.

We also observe significant depth reduction for the majority of the designs due to the simplified logic structure in the approximate designs. SCALS is able to achieve similar QoRs regardless of the different input distributions, showing that SCALS can flexibly adapt to the input characteristics and generate high-quality approximate designs for various types of input distributions.

3.2.2 Random-Control Circuit under Error Rate Constraint

Table 3.2 shows the area and depth reduction of the EPFL random-control designs generated from SCALS targeting 6-input LUTs under various input distributions. In this experiment, we require that the error rates of the generated designs do not exceed 1% with a confidence level of 0.95. Similar to the results of arithmetic designs, the QoR improvement of the approximate designs is design specific. Noticeably, for design `priority`, a 128-to-7 priority encoder, SCALS achieves almost 10x reduction in both area and depth. On the other hand, designs such as `alu ctrl` that have simple internal logic, SCALS could not simplify its logic at all.

3.2.3 Comparison with Existing Work Targeting ASICs

We compare our approach with the single-selection algorithm in [105], which represents the state-of-the-art approximate logic synthesis method for ASICs. Since the single-selection algorithm focuses on area reduction after technology-mapping, we mainly compare the post-mapping area results of the generated approximate designs. In this experiment, we use seven MCNC benchmarks

Table 3.1: **Area and depth reduction for arithmetic circuit under mean relative error magnitude constraint with various input distributions —** `Exact` = Exact designs from the EPFL benchmark record; `SCALS` = Approximate designs synthesized using our method; `Size` = Area of the circuit measured as the number of 6-input LUTs; `Dpt` = Depth of the circuit in terms of 6-input LUTs; `Ratio` = The ratio of size of depth of `SCALS` over `Exact`. The average error magnitude is constrained to be within $\frac{1}{2^9}$ ($\approx 0.2\%$) of the correct output value. Numbers in bracket indicate the size/depth ratio over the corresponding exact design. All designs pass hypothesis testing with a confidence level of 0.95.

	Exact		SCALS							
			Uniform		Gaussian		Exponential		Bimodal	
Designs	Size	Dpt	Size	Dpt	Size	Dpt	Size	Dpt	Size	Dpt
adder	192	64	137 (0.71)	10 (0.16)	139 (0.72)	12 (0.19)	129 (0.67)	14 (0.22)	142 (0.74)	9 (0.14)
shifter	512	4	461 (0.90)	4 (1.00)	478 (0.93)	4 (1.00)	487 (0.95)	4 (1.00)	452 (0.88)	4 (1.00)
divisor	3268	1208	3232 (0.99)	1068 (0.88)	3122 (0.96)	842 (0.70)	1580 (0.48)	268 (0.22)	2651 (0.81)	699 (0.58)
hyp	40406	4532	3662 (0.09)	142 (0.03)	4201 (0.10)	152 (0.03)	4115 (0.10)	134 (0.03)	4028 (0.10)	133 (0.03)
log2	6574	119	6401 (0.97)	108 (0.91)	6529 (0.99)	118 (0.99)	6564 (1.00)	118 (0.99)	6485 (0.99)	117 (0.98)
max	523	189	184 (0.35)	19 (0.10)	180 (0.34)	16 (0.08)	130 (0.25)	3 (0.02)	158 (0.30)	22 (0.12)
mult	4923	90	1337 (0.27)	36 (0.40)	1959 (0.40)	45 (0.50)	1043 (0.21)	35 (0.39)	1057 (0.21)	25 (0.28)
sine	1229	55	1219 (0.99)	54 (0.98)	1218 (0.99)	54 (0.98)	1196 (0.97)	54 (0.98)	1205 (0.98)	55 (1.00)
sqrt	3077	1106	338 (0.11)	112 (0.10)	236 (0.08)	77 (0.07)	344 (0.11)	114 (0.10)	352 (0.11)	108 (0.10)
square	3246	74	490 (0.15)	19 (0.26)	867 (0.27)	27 (0.36)	771 (0.24)	20 (0.27)	2909 (0.90)	39 (0.53)

Table 3.2: **Area and depth reduction for random-control circuit under error rate constraint with various input distributions —** `Exact` = Exact designs from the EPFL benchmark record; `SCALS` = Approximate designs synthesized using our method; `Size` = Area of the circuit measured as the number of 6-input LUTs; `Dpt` = Depth of the circuit in terms of 6-input LUTs; `Ratio` = The ratio of size of depth of `SCALS` over `Exact`. The error rate is constrained to be within 1%. Numbers in bracket indicate the size/depth ratio over the corresponding exact design. All designs pass hypothesis testing with a confidence level of 0.95.

	Exact		SCALS							
			Uniform		Gaussian		Exponential		Bimodal	
Designs	Size	Dpt	Size	Dpt	Size	Dpt	Size	Dpt	Size	Dpt
arbiter	409	23	251 (0.61)	13 (0.57)	170 (0.42)	7 (0.30)	159 (0.39)	6 (0.26)	153 (0.37)	5 (0.22)
alu ctrl	27	2	27 (1.00)	2 (1.00)	26 (0.96)	2 (1.00)	26 (0.96)	2 (1.00)	26 (0.96)	2 (1.00)
cavlc	101	6	100 (0.99)	5 (0.83)	99 (0.98)	6 (1.00)	100 (0.99)	5 (0.83)	99 (0.98)	6 (1.00)
decoder	270	2	270 (1.00)	2 (1.00)	270 (1.00)	2 (1.00)	270 (1.00)	2 (1.00)	270 (1.00)	2 (1.00)
i2c controller	227	7	205 (0.90)	6 (0.86)	209 (0.92)	7 (1.00)	168 (0.74)	4 (0.57)	168 (0.74)	4 (0.57)
Int2float	28	6	26 (0.93)	6 (1.00)	25 (0.89)	6 (1.00)	26 (0.93)	4 (0.67)	23 (0.82)	4 (0.67)
mem ctrl	2354	22	2086 (0.89)	15 (0.68)	1895 (0.81)	14 (0.64)	1316 (0.56)	11 (0.50)	1468 (0.62)	8 (0.36)
priority	110	26	12 (0.11)	3 (0.12)	13 (0.12)	2 (0.08)	11 (0.10)	3 (0.12)	54 (0.49)	21 (0.81)
router	52	6	31 (0.60)	2 (0.33)	31 (0.60)	2 (0.33)	35 (0.67)	4 (0.67)	32 (0.62)	2 (0.33)
voter	1301	17	1299 (1.00)	17 (1.00)	1298 (1.00)	17 (1.00)	1299 (1.00)	17 (1.00)	1299 (1.00)	17 (1.00)

Table 3.3: **Comparison with a state-of-the-art approximate logic synthesis method for ASIC targeting area minimization [105] —** `Base` = Initial exact designs reported by [105]; `[105]` = Results of the single-selection algorithm in [105]; `SCALS` = Results from our approach; `Time` = Runtime in seconds. Both `[105]` and `SCALS` enforce a 1% error rate constraint. The designs generated by SCALS pass the hypothesis testing at a confidence level of 0.95. The area and delay numbers are normalized to the area and delay of a unit size inverter, respectively.

	Base		Wu, et al. [105]			SCALS				SCALS vs. [105]
Designs	Delay	Area	Time (s)	Area	Ratio vs. Base	Time (s)	Delay	Area	Ratio vs. Base	Ratio vs. [105]
c880	40.4	599	93	497	0.83	507	36.7	494	0.82	0.99
c1908	60.6	1013	394	654	0.65	93	45.2	324	0.32	0.50
c2670	67.3	1434	702	935	0.65	137	35.4	748	0.52	0.80
c3540	84.5	1615	172	1554	0.96	101	54.0	1396	0.86	0.90
c5315	75.3	2432	263	2352	0.97	251	47.5	2245	0.92	0.95
c7552	159.8	2759	533	2527	0.92	476	155.7	2396	0.87	0.95
alu4	51.5	2740	1000	2433	0.89	607	45.3	1099	0.40	0.45
geomean					0.83				0.63	0.76

Table 3.4: **Comparison with a state-of-the-art approximate logic synthesis method for FPGA targeting depth-constrained area minimization [106] —** `Base` = Exact designs generated from ABC [10]; `Size` = Area of the circuit measured as the number of 4-input LUTs; `Dpt` = Depth of the circuit in terms of 4-input LUTs. Both approaches generate designs with an error rate constraint of 5%. The designs generated by SCALS pass the hypothesis testing at a confidence level of 0.95. Both approaches do not increase the depth of the baseline designs. No runtime information is provided for these set of designs in [106].

	Base		Wu, et al. [106]		SCALS		SCALS vs. [106]
Designs	Size	Depth	Size	Ratio vs. Base	Size	Ratio vs. Base	Ratio vs. [106]
c432	97	10	79	0.81	55	0.57	0.70
c880	128	8	102	0.80	107	0.84	1.05
c1908	122	9	50	0.41	88	0.72	1.76
c2670	295	7	242	0.82	224	0.76	0.93
c3540	346	12	325	0.94	305	0.88	0.94
c5315	503	9	468	0.93	439	0.87	0.94
c7552	593	8	486	0.82	440	0.74	0.91
alu4	710	7	483	0.68	411	0.58	0.85
alu2	160	12	136	0.85	135	0.84	0.99
apex6	253	6	197	0.78	210	0.83	1.07
dalu	425	11	349	0.82	329	0.77	0.94
geomean				0.77		0.76	0.98

that are also used in [105][3]. We set the error rate constraint to 1% in Table 3.3. The baseline designs in Table 3.3 are the initial exact designs generated from ABC [10].

Compared to the single-selection algorithm, our approach achieves higher area reduction across all the benchmarks, with an average improvement of 37% over the baseline designs, while [105] achieves an average improvement of 17%. When directly comparing with the single-selection algorithm, our approach reduces the area by 24% on average across the seven benchmarks considered here. We observe similar area improvements for other values of error rates. For example, at 5% error rate, the designs generated from our approach are 10% smaller on average than the designs from [105]. While the approach in [105] does not report the delay numbers of the generated designs, we observe that our approach achieves smaller delay than the baseline designs, mainly due to the simplified logic structure in the approximate designs. The runtime of our approach is in general on the same order of the approach in [105], and the average runtime across the seven benchmarks is smaller than that of the approach in [105].

3.2.4 Comparison with Existing Work Targeting FPGAs

Table 3.4 shows the area reduction compared with a state-of-the-art FPGA approximate logic synthesis algorithm [106] over a set of benchmarks from the MCNC benchmark suite that were used in [106]. The baseline designs are the exact designs generated from ABC [10]. Following the requirement in [106], both approaches enforce a 5% error rate constraint and require that the gen-

[3]The single-selection algorithm [105] reports area results for 12 benchmarks in total. However, we are only able to obtain seven of the benchmarks since the other five benchmarks are not publicly available.

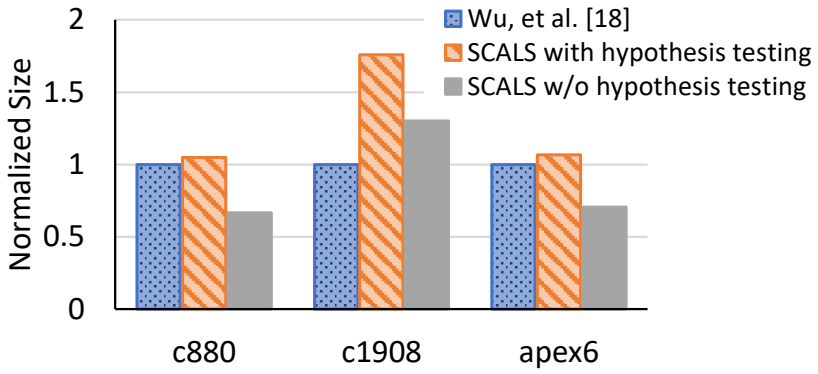


Figure 3.5: **Area comparison after disabling hypothesis testing.**

erated approximate designs do not increase the depth of the baseline designs. We measure the size of the designs as the number of 4-LUTs. SCALS generates smaller designs when compared with [106] for eight out of the 11 designs, showing the effectiveness of SCALS even when compared against highly specialized techniques such as [106].

We note that although SCALS and [106] both enforce a 5% error rate constraint, it is still not an apple-to-apple comparison since the requirement for an approximate design to pass the test of SCALS is stricter than that in [106]. The hypothesis testing step in SCALS would potentially reject an approximate design that passes a simple sampling-based test as in [106]. To understand the impact of applying hypothesis testing on the design QoR, we look into the three designs in Table 3.4 that SCALS fails to improve over [106]. Figure 3.5 shows the size of the generated designs from SCALS with and without the hypothesis testing step. The sizes of the designs are normalized to the corresponding design generated by [106]. We observe that disabling the hypothesis testing step indeed leads to smaller designs.

Table 3.5: **Synthesis results under different confidence levels —** `Size` = Area of the circuit measured as the number of 6-input LUTs; `Dpt` = Depth of the circuit in terms of 6-input LUTs; `CL` = Confidence level for error rate during hypothesis testing. The designs are generated under 1% error rate constraint after 10 trials.

	CL = 0.90		CL = 0.95		CL = 0.99	
Designs	Size	Dpt	Size	Dpt	Size	Dpt
arbiter	348	17	352	17	354	21
mem ctrl	2309	21	2315	21	2354	22
priority	14	2	14	4	16	4
router	32	3	33	3	34	3

3.2.5 QoR vs. Confidence Level Tradeoff

Table 3.5 shows the impact of confidence level during hypothesis testing on the area and depth of the final designs using four random-control circuits under error rate constraint. A higher confidence level requires stronger evidence for certifying that an error constraint is honored. Consequently, a higher confidence level will lead to more conservative designs as shown in Table 3.5, which provides a tradeoff between QoR improvement and statistical confidence of the error behavior.

3.2.6 Design Study: Approximate FIR Filter

We provide a design study using approximate multipliers and adders generated using SCALS from design-specific input vectors to construct a three-tap finite response (FIR) filter. The FIR filter has a passband between 100 to 200Hz and a stopband from 300 to 500Hz. To supply the input patterns for SCALS, we use realistic input waveform that contains two major frequency components at 100Hz and 300Hz with Gaussian noise sampled at 1KHz. All inputs to the

Table 3.6: **Area and delay comparison between an exact FIR filter and an approximate FIR filter from SCALS —** `Exact` = Exact designs generated using ABC's optimization script `resyn2` and technology mapper `map`; `SCALS` = Approximate designs synthesized using our method; `Area` = ASIC circuit area; `Delay` = ASIC circuit delay; `MREM` = Measured mean relative error magnitude. The area and delay numbers are normalized to those of a unit size inverter.

	Multiplier		Adder		FIR	
	Exact	SCALS	Exact	SCALS	Exact	SCALS
Area	55544	12058	2476	465	171234	33607
Area ratio	1.00	0.22	1.00	0.19	1.00	0.20
Delay	215.2	104.6	204.6	32.7	226.9	112.5
Delay ratio	1.00	0.49	1.00	0.16	1.00	0.50
MREM		0.009%		0.001%		0.012%

multipliers and adders are 64-bit fixed-point numbers.

In Table 3.6, the area and delay numbers are the post-synthesis results normalized to the area of a unit-size inverter. The measured MREM is verified at a confidence level of 0.95. The generated approximate FIR filter is 5x smaller and 2x faster than the exact counterpart, while maintaining similarly small MREM (0.012%) of its basic building blocks.

3.3 Related Work

We first summarize the existing research directions on approximate logic synthesis, followed by describing a logic synthesis and technology mapping framework that our work builds on.

We review and summarize a subset of representative work on approximate logic synthesis. Miao, et al. [69] consider approximate logic synthesis under error magnitude constraint as the Boolean relation minimization problem, and devise an efficient heuristic algorithm for iteratively refining the magnitude-

constrained solution to arrive at a solution also satisfying the error frequency constraint. This work is extended by [70] to handle multi-level logic networks, which converts the approximate logic synthesis problem into solving a series of conventional external don't-cares-based network optimizations. Shin, et al. [91] propose techniques for synthesizing approximate two-level circuits by selectively complementing the output values of the minterms to reduce the number of literals in the SOP representation. Venkataramani, et al. [101] discuss approximation techniques by identifying signal pairs in the circuit that assume the same value with high probability, and substitute one for the other. Wu, et al. [105] propose techniques for approximate logic synthesis under error rate constraint, which iteratively picks the most effective nodes in a Boolean network to shrink by approximating their factored-form expressions. They further extend their techniques for mapping to FPGAs by removing wires in the LUT network and changing the functionality of LUTs [106].

3.4 Conclusions and Discussions

We propose SCALS, a statistically certified approximate logic synthesis framework based on parallelized stochastic optimization. SCALS effectively handles various error metrics, technology targets and input distributions in a unified framework, and provides statistical guarantee that the generated designs adhere to user-specified error constraints.

Since SCALS conducts one hypothesis testing for each approximate transformation, the multiple comparisons problem [48] may occur and potentially impact the inference accuracy. To illustrate the problem, consider the extreme case

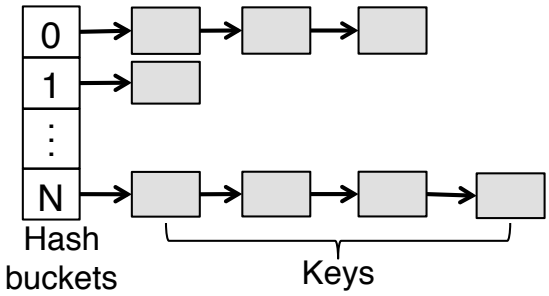
where a sequence of approximate transformations that do not alter the functionality of the circuit at all. Since we are essentially making multiple inferences on the functionally-equivalent circuits, the likelihood of erroneous inferences due to sampling noise increases when compared to a single inference. Multiple testing correction techniques such as the Bonferroni correction [90] can be applied to eliminate the multiple comparisons problem in SCALS. The current version of SCALS does not include mechanisms to handle the multiple comparisons problem partially due to the empirical observation that most approximate transformations do alter the functionality of the underlying circuit.

Another potential extension in SCALS is to speed up the random test vector simulation using techniques such as batch statistical error estimation [93]. Since a large number of potential approximate transformations may exist for a given circuit, estimating the accuracy of each approximate transformation individually can be time consuming. The batch statistical error estimation technique pre-computes the observability of local changes at the primary outputs using observability analysis techniques [71], and estimates the error metrics of various approximate transformations based on the observability of the nodes affected by the approximation. Since the observability computation is reused across different approximate transformations, batch statistical error estimation can significantly reduce the simulation time due to multiple potential transformations.

CHAPTER 4

ELASTICFLOW: TAILORING ACCELERATOR ARCHITECTURE TO IRREGULAR WORKLOADS

In the previous two chapters, we have shown that the cross-stage optimization technique can improve the logic synthesis step by considering the effects of logic transformations on the downstream flow. In the following two chapters, we demonstrate that this general technique can also be applied to architectural synthesis. We first discuss its application to the problem of synthesizing fixed-function circuits, which is also known as HLS. In HLS, one widely used optimization technique is pipelining, which creates a static schedule for the loop (or function) body to allows successive loop iterations (or function invocations) to be overlapped during execution. Pipelining aims to minimize the interval between the initiation of successive loop iterations (or function invocations) to achieve the highest possible throughput for the particular design. While modern HLS tools provide comprehensive support for pipelining a single loop or perfect loop nests, they are unable to effectively optimize *irregular loop nests* that contain dynamic-bound inner loops. Since existing HLS pipelining techniques are largely driven by static scheduling and resource binding, the synthesized hardware cannot adapt to the varying amount of work incurred by the data-dependent loop bounds, resulting in either low performance or poor resource utilization. Unfortunately, such loop nests are commonplace in a variety of important application domains such as scientific computing, social analytics, and in-memory databases, as they are an inextricable part of key operations such as sparse matrix-vector multiplication, graph traversal, and hash lookup. Generating efficient accelerators for applications in these domains remains a serious challenge for contemporary HLS tools [104].



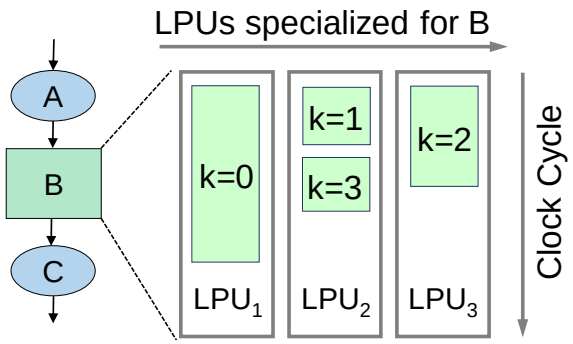
(a) A common hash table implementation which uses separate chaining to resolve collisions.

```
1  for (k : keys_to_find){
2  #pragma pipeline
3    // Stage A
4    hv = Jenkins_hash(k);
5    p = hashtbl[hv].keys;
6
7    // Stage B
8    while (p && p->key!=k)
9      p = p->next;
10
11   // Stage C
12   format_output(p);
13 }
```

(b) `keysearch` kernel using a `while` loop.

```
1  for (k : keys_to_find){
2  #pragma pipeline
3    hv = Jenkins_hash(k);
4    p = hashtbl[hv].keys;
5
6    for (i=0; i<M; i++){
7    #pragma unroll
8      if (p && p->key!=k)
9        p = p->next;
10   }
11
12   format_output(p);
13 }
```

(c) Modified `keysearch` using a `for` loop unrolled `M` times.



(d)ElasticFlow architecture and parallel execution of a dynamic-bound inner loop.

Figure 4.1: **Irregular loop nest example** – `keysearch`, a kernel from the Memcached benchmark which performs a series of hash lookups; (a) Hash table representation; (b) Original code using a dynamic-bound inner loop to do pointer chasing; (c) Modified code using an unrolled fixed-bound inner loop of `M` iterations; (d) Conceptual illustration of ElasticFlow architecture and dynamic-bound loop execution.

The difficulty presented by irregular loop nests reveals the fundamental weakness of existing pipelining techniques in handling elastic workloads. For example, Figure 4.1(a) depicts a hash table which employs separate chaining,

and Figure 4.1(b) shows `keysearch`, a kernel from the Memcached application [40] which performs a series of key lookups by first computing the hash bucket, and then pointer chasing over the keys in the collision chain of that particular bucket using a while loop. While some collision chains may be very long in pointer chasing kernels, most others contain only a couple of elements. In this example, we are interested in building a pipelined accelerator for this kernel which can achieve a throughput of one lookup per cycle. If we simply pipeline the outer loop without modifying the inner loop, since the bound of the inner loop is unknown at compile-time, the synthesized design must finish all iterations of an inner loop before starting the next outer loop iteration. As a result, the design will be bottlenecked by the throughput of the inner loop, which is much lower than the throughput target.

An alternative is to transform the dynamic-bound loop in Figure 4.1(b) into a fixed-bound loop by either manually changing the code or annotating static loop bounds through the HLS tool. The result is shown in Figure 4.1(c), where we have assumed `M` is the worst-case length of the collision chain in the hash table. With a fixed inner loop bound of `M`, the tool can now unroll the inner loop by a factor of `M` and then pipeline the outer loop body to achieve the desired throughput of one lookup per cycle. However, two major problems remain: (1) The design is very inefficient in area – a good hash function ensures that the vast majority of hash buckets contain at most one key, which requires only one loop iteration. However, we are forced to unroll `M` copies of the loop body to handle the extreme cases of the few buckets that contain multiple keys, appropriating resources that will spend most of their time idle. (2) For many loops, the worst-case loop bound cannot be statically determined. The bound-annotation approach cannot be applied in this case without endangering program correct-

ness because the maximum number of keys in a hash bucket is often unknown.

The fundamental problem showcased by this example is that existing pipelining techniques statically allocate and bind resources for workload that is inherently elastic. This leads to gross over-allocation of resources to handle the worst-case load, which is often unbounded. To efficiently address irregular loop nests with dynamic-bound inner loops, we argue that a new architectural synthesis approach is needed for generating hardware capable of dynamically optimizing loop execution for varying amount of work by adaptively assigning resources for compute as well as memory. The objective is to allocate an appropriate number of compute and memory resources and to maximize the utilization of these resources.

In this chapter we propose *ElasticFlow* – a novel area-efficient architecture and synthesis technique to enable area-effective pipelining of irregular loop nests with dynamic-bound inner loops. For the example in Figure 4.1(b), ElasticFlow generates a dataflow pipeline architecture containing an array of *loop processing units* (LPUs), each of which continuously executes an entire dynamic-bound inner loop to completion. Each iteration of the outer loop dynamically dispatches each of its inner loops (i.e., `Stage B`) to an LPU, allowing multiple outer loop iterations to execute in a pipelined fashion.

Figure 4.1(d) shows the conceptual ElasticFlow architecture and dynamic-bound loop execution for `keysearch`, where each of the hardware blocks `A`, `B`, and `C` corresponds to one of the stages labeled in Figure 4.1(b). In particular, `Stage B`, which contains a dynamic-bound inner loop, is implemented using three LPUs that execute in parallel multiple inner loops from different outer loop iterations. Since the dynamic-bound inner loops are the performance bot-

tleneck of the entire design, executing them in parallel improves the throughput of the entire design.

Determining the appropriate number of LPUs for each dynamic-bound inner loops is a key to achieving the desired performance and resource usage. Besides allowing designer's manual specification, we propose an automatic resource allocation method, detailed in Section 4.3, for determining the LPU configuration. Our proposed resource allocation method first profiles the original design to obtain important performance and resource usage statistics such as average-case loop bound for various stages in the loop nest. Using the profiling results, our LPU allocation algorithm determines the configuration of the LPU arrays for a given throughput target. Finally, we synthesize the complete ElasticFlow architecture by combining predefined architectural templates with design-specific pipelines.

Our major contributions include:

1. We propose a novel pipelined architecture and associated synthesis techniques to effectively accelerate irregular loop nests that contain dynamic-bound inner loops in HLS.
2. We propose an adaptive resource allocation and reallocation technique to reduce hardware overhead and improve pipeline performance for loop nests containing multiple dynamic-bound inner loops.
3. We propose a flexible multi-bank memory architecture to support high-throughput loop processing that is not bottlenecked by memory communication while maintaining a low memory-related hardware overhead.
4. We systematically study the trade-off between performance and resource usage in terms of the number of LPUs, number of memory banks, and

buffer sizes.

5. We demonstrate substantial performance improvement over a best-in-class commercial HLS tool for Xilinx FPGAs for a suite of real-life applications.

4.1 Irregular Loop Nests

This chapter focuses on irregular loop nests with one or more dynamic-bound inner loops. Such loop patterns often arise from operations on less-regular data structures such as sparse matrices, graphs and hash tables. Figure 4.2 shows three real-world application kernels which exhibit this pattern. Notably, each application uses a different data structure which is usually sparse in practice – input values for `PC` tend not to span all the bits, sparse matrices for `SPMV` by definition contain very few non-zero entries compared to the number of matrix columns, and graphs for `SSSP` tend to be sparsely connected. These observations imply that the inner loops of these applications will almost never require the worst-case number of iterations.

The key challenge of pipelining irregular loop nests arises from the compile-time-unknown inner loop bounds, which inhibit static compiler transformations commonly used in traditional pipelining techniques. Unrolling is also extremely inefficient even if the loop pattern possesses a known worst-case bound, as common-case execution will leave most resources idle. One approach is to execute several inner loop iterations concurrently on multiple hardware copies. Unfortunately, this requires there to be no carried dependences in the inner loop, which is not the case in many practical applications as Figures 4.1

```
1 for (i = 0; i < num_imgs; i++){
2 #pragma pipeline
3   val = imgdiff[i];
4   count = 0;
5   while (val){
6     count++;
7     val = val & (val - 1);
8   }
9   pc[i] = count;
10 }
```

(a) PC

```
1 for (i = 0; i < num_rows; i++){
2 #pragma pipeline
3   out[i] = 0;
4   s = row[i]; e = row[i+1];
5
6   for (c = s; c < e; c++){
7     cid = col[c];
8     out[i] += val[c] * vec[cid];
9   }
10 }
```

(b) SPMV

```
1 for (i = 0; i < num_vertices; i++){
2 #pragma pipeline
3   v = vertice[i];
4   newdist[i] = dist[i];
5   for (e = v.edge; e; e = e->next){
6     j = e.target_vertice_id;
7     if (dist[j] + e.weight < tmp)
8       newdist[i] = dist[j] + e.weight;
9   }
10 }
```

(c) SSSP

Figure 4.2: **Representative irregular loop kernels** – (a) `Population Count (PC)` counts the ones in a bit vector, with an inner loop bound equal to the number of set bits. (b) `Sparse Matrix-Vector Multiplication (SPMV)` accesses each element in a sparse matrix, and has an inner loop bound dictated by the number of non-zero matrix entries; (c) `Single-Source Shortest Path (SSSP)` implements one iteration of the Bellman-Ford algorithm, which updates the distance of each node by examining each of its neighbors in the inner loop.

and 4.2 demonstrate. However, we note that in these examples there are *no outer-loop-carried dependences involving the irregular inner loops*, allowing these inner loop instances from different outer loop iterations to be executed in parallel. For the example in Figure 4.1(d), any outer-loop-carried dependences on `Stage A` would not prevent us from parallelizing `Stage B` if the dependences do not involve `Stage B`. Indeed, we observe that such dependence patterns are common in many important applications, leading us to consider an approach where we pipeline across different outer loop iterations by parallelizing the execution of multiple outer loop instances of an inner loop.

4.2 ElasticFlow Architecture

To address the challenges introduced by these irregular loop nests, we present the ElasticFlow architecture, which implements such loop patterns as a multi-stage dataflow pipeline.

This novel architecture dynamically distributes different outer loop instances of the dynamic-bound inner loop to one or more processing units across which different instances of the inner loops execute in a pipelined parallel fashion. Figure 4.3 provides an example which roughly corresponds to the `keysearch` kernel in Figure 4.1(b). Stage `B` implements a dynamic-bound inner loop as a loop processing array (LPA), which consists of multiple loop processing units (LPUs) as well as a *distributor* to distribute work to and a *collector* to collect results from the LPUs. The on-chip *banked memory* employs a multi-bank memory architecture to provide high-throughput data transfer from and to the LPUs. Each LPU contains the full datapath and control for executing the inner

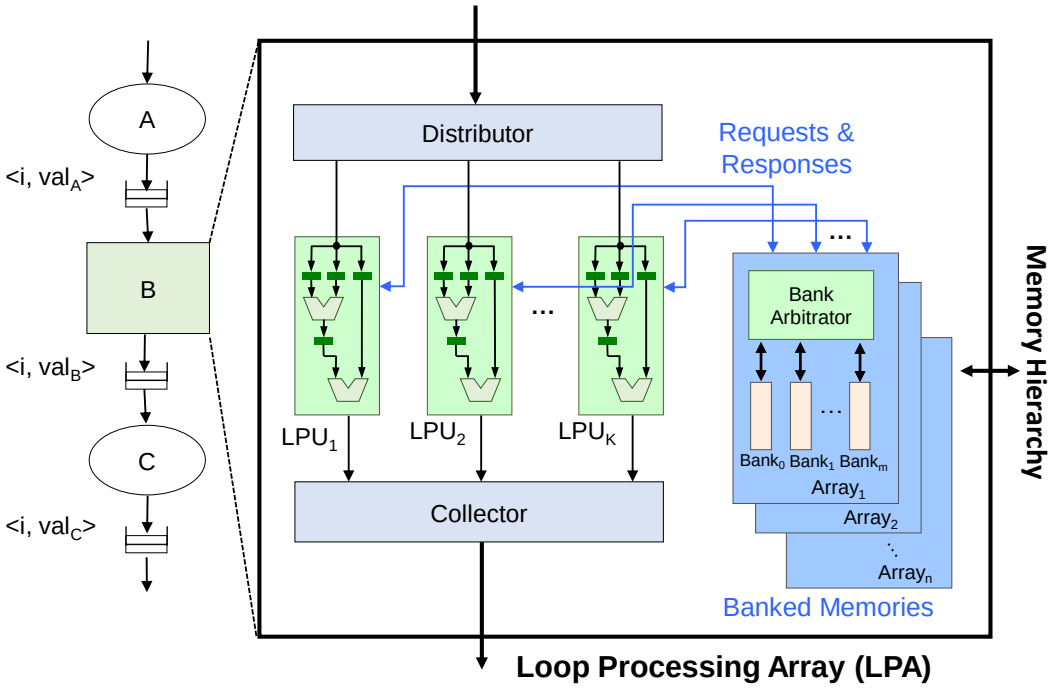
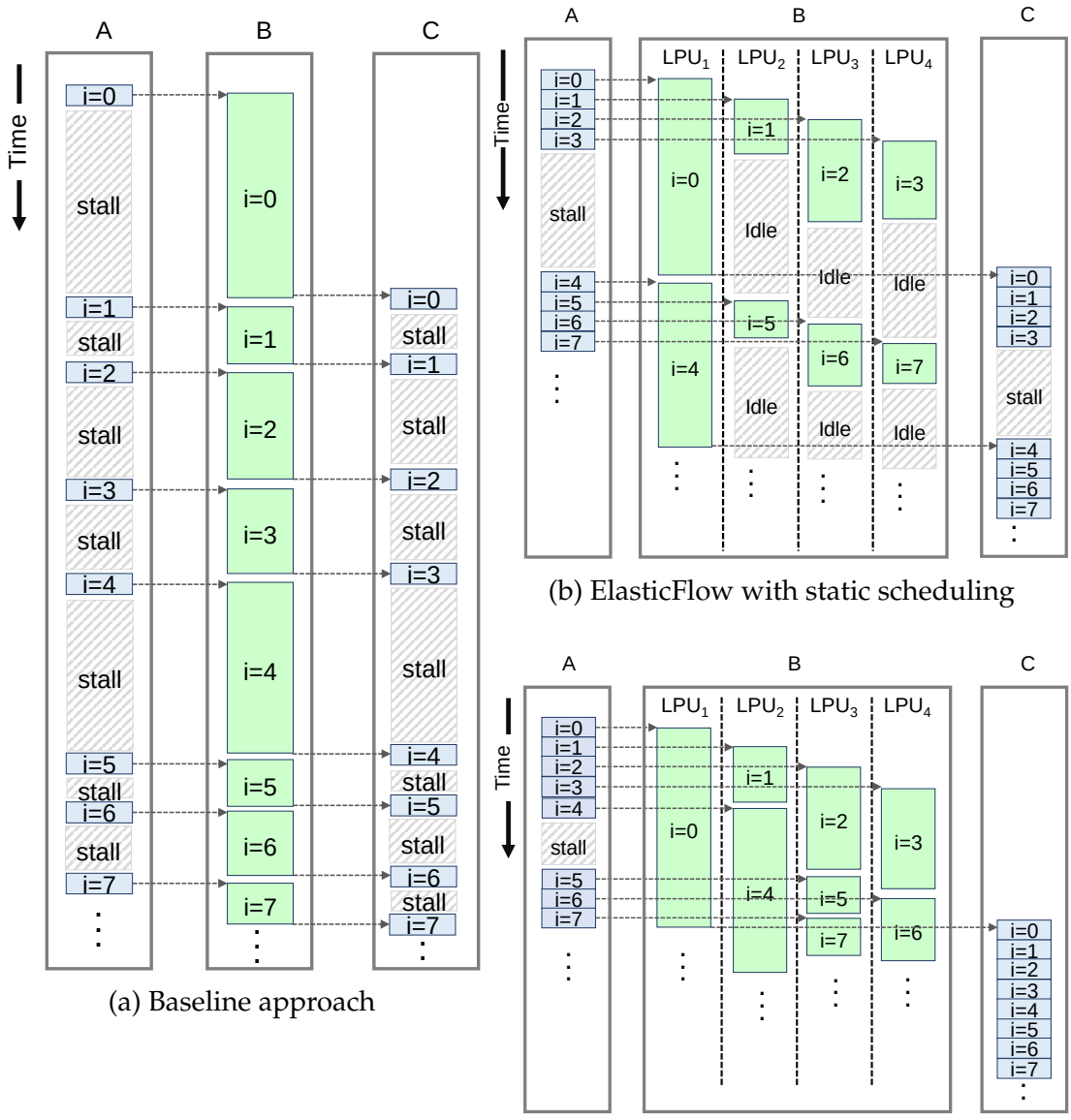


Figure 4.3: **ElasticFlow architecture** – An irregular loop nest is transformed into a multi-stage dataflow pipeline. Each dynamic-bound inner loop is mapped to a loop processing array (LPA), which consists of multiple loop processing units (LPUs). The loop iteration ID (`i`) and live values (`val_A`, `val_B`, `val_C`) are passed through the FIFOs between pipeline stages. A banked memory architecture is used to provide sufficient memory bandwidth to the LPUs.

loop. Because each LPU executes all iterations of an inner loop body until completion for a particular outer loop iteration, the loop-carried dependence within an inner loop is automatically handled by the LPU. Other operations in the loop nest become fixed-latency pipelined stages (i.e., Stages `A` and `C`), which can be synthesized using traditional pipelining techniques. Loop iteration IDs and live values are passed through the FIFOs between pipeline stages.



(a) Baseline approach

(b) ElasticFlow with static scheduling

(c) ElasticFlow with dynamic scheduling

Figure 4.4: **Execution on different pipeline architectures** – The irregular loop nest in this example is mapped to three pipeline stages (`A`, `B`, `C`), where `B` implements a dynamic-bound inner loop. Eight outer loop iterations are shown. (a) Baseline approach uses a sequential datapath for `B`, resulting in frequent pipeline stalls due to low inner loop throughput; (b) ElasticFlow with four parallel LPUs for `B` can improve throughput by overlapping different inner loop instances, but the LPUs are underutilized due to the static scheduling policy; (c) Dynamic scheduling can further increase throughput by improving LPU utilization.

4.2.1 sLPA Architecture

A natural approach is to map each dynamic-bound inner loop to a *single-loop processing array* (sLPA) containing multiple *single-loop processing units* (sLPUs) which execute that inner loop until completion.[1] Figure 4.4 illustrates pipelining using sLPAs with eight iterations of `keysearch`. Figure 4.4(a) shows the baseline approach taken by existing HLS tools, where every inner loop iteration executes serially on stage `B`. The throughput of stage `B` becomes the bottleneck, and the rest of the pipeline is frequently stalled. Figure 4.4(b) shows the sLPA approach using four sLPUs. In this example each outer loop iteration is statically assigned an LPU based on its ID modulo four. While throughput is improved, resource efficiency is poor as the work is skewed towards LPU one, resulting in periods of idling on the other three LPUs. Indeed, it is not possible for a static scheduling policy to efficiently handle work imbalance. To guarantee resource efficiency, ElasticFlow employs a dynamic scheduling policy where an outer loop may dispatch its dynamic-bound inner loop to a free LPU. The resulting improvement in LPU utilization and throughput is shown in Figure 4.4(c). This runtime mechanism marks a fundamental difference between existing pipelining techniques in HLS and ElasticFlow. While it is possible to replicate inner loop modules via unrolling and achieve parallelization in current HLS tools, the number of loop copies must be chosen at compile time to handle the worst-case loop bound, resulting in enormous resource inefficiency. In contrast, the number of ElasticFlow LPUs for a given dynamic-bound inner loop can target the common case, achieving maximum throughput most of the time while conserving resource.

[1] We focus on two-level loop nests for the rest of this chapter, but ElasticFlow can be generalized to multi-level loop nests through hierarchical pipelining.

It is important to note that although different outer loop iterations begin executing on an LPA in-order, they may finish out-of-order because the latency of each inner loop varies depending on the outer loop iteration, and cause incorrect results for many programs. To address this, the *collector* of an LPA implements a *reorder buffer (ROB)* that ensures results are produced in the order indicated by the loop iteration ID.

Because each LPU continuously executes an entire inner loop to completion, inner loop carried dependencies are naturally honored. As discussed in Section 4.1, ElasticFlow restricts that there are no outer loop carried dependencies involving stages synthesized to LPAs. Our synthesis algorithm ensures only loop nests adhering to this restriction will be mapped to this architecture.

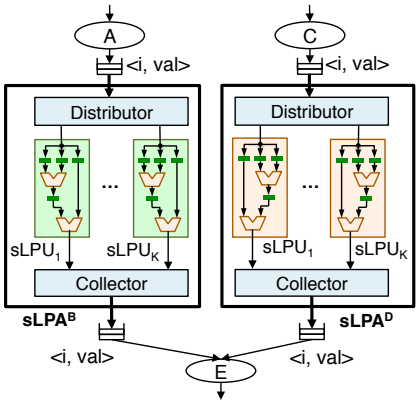
4.2.2 mLPA Architecture

The ElasticFlow architecture can further enable resource sharing at runtime by using *multi-loop processing units* (mLPUs), which have the ability to execute different dynamic-bound inner loops. We illustrate the advantage of this design with Database Join (`dbjoin`) in Figure 4.5(a), a common operation which combines the entries in two hash tables so they can be written to a single new table. The kernel uses two dynamic-bound inner loops (i.e., `B` and `D`), which is synthesized into two sLPAs with three sLPUs each as in Figure 4.5(b). Figure 4.5(c) shows the execution of an example workload which is heavily unbalanced, such that loop B requires many more iterations than loop D. This causes idling in sLPA^{D} as sLPA^{B} bottlenecks pipeline progress.

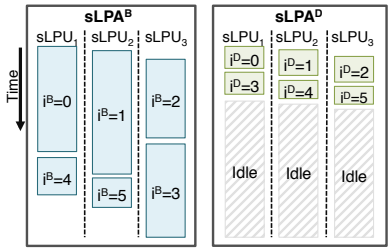
As before, this inefficiency is a consequence of static resource assignment for

```
1  for (i = 0; i < num_keys; i++) {
2  #pragma pipeline
3    // stage A: look up hashtbl1
4    k = input_keys[i]
5    hv1 = Jenkins_hash1(k);
6    p = hashtbl1[hv].keys;
7    // stage B: dynamic-bound loop
8    while (p && p->key!=k)
9      p = p->next;
10   // stage C: look up hashtbl2
11   k = input_keys[i]
12   hv2 = Jenkins_hash2(k);
13   q = hashtbl2[hv].keys;
14   // stage D: dynamic-bound loop
15   while (q && q->key!= k)
16     q = q->next;
17   // stage E: merge the results
18   return format_out(p, q);
19 }
```

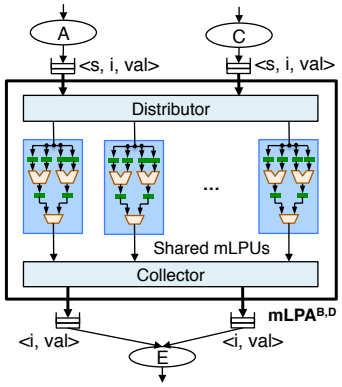
(a) Kernel for `dbjoin`



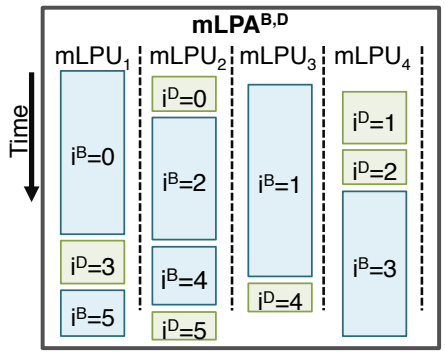
(b) Architecture without LPU sharing



(c) Execution of six loop iterations on (b)



(d) Architecture with LPU sharing



(e) Execution of six loop iterations on (d)

Figure 4.5: **LPU sharing and adaptive resource reallocation** – This example shows sharing and no-sharing architectures for the Database Join (`dbjoin`) kernel, (a) `dbjoin` code showing the five pipeline stages (`A`, `B`, `C`, `D`, `E`), where `B` and `D` contain dynamic-bound inner loops; (b) ElasticFlow architecture with two separate `sLPAs`: sLPA^{B} for stage `B` and sLPA^{D} for stage `D`; (c) Execution of six iterations on the no-sharing architecture; (d) ElasticFlow architecture with LPU sharing, where the `mLPA` consists of a set of `mLPUs` that can be shared between stages `B` and `D`. (e) Execution on the sharing architecture; `sLPU` = single-loop processing unit; `mLPU` = multi-loop processing unit; `sLPA` = single-loop processing array; `mLPA` = multi-loop processing array; `s` = stage ID (i.e., `B` or `D` for this example); i^{B}, i^{D} = loop iteration ID for stage `B` or `D`; `val` = live-in values for the downstream pipeline stages.

an unbalanced workload. Figure 4.5(d) shows an alternative architecture where the sLPAs are fused. The resulting *multi-loop processing array* (mLPA) contains four mLPUs, each of which is capable of executing either loop. An additional parameter, the stage IDs (`s`), must be passed into the mLPA to configure the mLPU to execute the desired loop. As Figure 4.5(e) demonstrates, the mLPUs can be dynamically reallocated for loop `B` or `D` depending on the amount of work, improving resource utilization and reducing pipeline stalls. Compared to sLPUs, an mLPU trades off additional area for increased throughput on unbalanced workloads. These trade-offs are further explored in Section 4.4.6.

4.2.3 Memory Banking

For the applications that access memory in the inner loop, we need to design a high-bandwidth memory system in the LPA that can respond to multiple memory requests from multiple LPUs simultaneously. A naïve implementation is to replicate the memory array and have one copy of the entire memory array in each LPU. Although the replication approach provides sufficient read bandwidth, the memory size would quickly increase with the number of LPUs, resulting in a significant area overhead. Moreover, replicating the memory does not increase the bandwidth for write operations, since each LPU would need to write to all the replicates to maintain coherence. Another option is to use a multi-port memory to handle simultaneous requests from multiple LPUs. However, this is not a scalable approach either since the area and power of a multi-port memory typically increases quadratically with the number of ports [99].

To achieve high memory bandwidth without significantly increasing re-

source usage, we employ a banked memory architecture as shown in Figure 4.3. The banked memory architecture contains a memory bank arbitrator as well as a number of memory banks. The memory bank arbitrator receives memory requests from all the LPUs in the LPA, arbitrates for the available ports of memory banks, and sends memory responses back to the corresponding LPUs. Each memory bank stores a subset of memory content in the address space. The bank arbitrator is responsible for ensuring that the memory requests are mapped to the correct banks, and that each bank receives at most one memory request at any given cycle.

In the case where multiple requests contend for the same bank, the bank arbitrator resolves the conflict by granting memory access to the request with the highest priority while stalling others. We use a fixed priority sorted by the LPU IDs for arbitrating memory requests. For example, when LPUs 1, 2 and 4 all issue memory requests to the same memory bank at a given cycle, the arbitrator will grant access to LPU 1, while temporarily stalling two other LPUs. While the fixed-priority policy always prioritizes the LPU with a smaller ID, it does not necessarily degrade performance owing to the out-of-order nature of the LPA execution. More concretely, once an LPU completes processing the current outer-loop iteration, it will store the result into the ROB and immediately start processing the next iteration, regardless of whether any previous outer-loop iterations are still in flight.

For applications that access multiple arrays in the inner loop, we create a banked memory system for each individual array with its own bank arbitrator. The banked memory system also naturally supports memory write operations, since each write operation only needs to update the corresponding memory lo-

cation in a single memory bank.

Although we limit the scope to on-chip memories in this chapter, we note that with a decoupled memory interface and a dedicated memory controller, it would not be difficult to extend the ElasticFlow architecture to interface with an external memory hierarchy as shown in Figure 4.3.

4.3 ElasticFlow Synthesis

ElasticFlow synthesis maps an irregular loop nest to the architecture proposed in Section 4.2, which requires partitioning the loop nest into multiple stages, identifying inner loop candidates to form the LPAs, and synthesizing these loop bodies into sLPUs and mLPUs. The process considers resource sharing among different inner loops to optimize area usage, synthesizing a high-throughput memory system for the LPUs, scheduling of work onto LPUs to maximize throughput, and buffer sizing to avoid pipeline stalls.

Given a dependence graph for an irregular loop nest that captures both intra-iteration and inter-iteration data and control dependences, our synthesis algorithm first applies dependence analysis [53] to identify all dynamic-bound inner loops and partition each of these inner loops into separate stages. The subgraphs preceding or succeeding each dynamic-bound inner loop will be partitioned into their own stages, resulting in a coarse-grained directed acyclic graph (DAG) composed of stages. Figure 4.5 provides an example, where the kernel is partitioned into five stages (`A`–`E`), and the two dynamic bound inner loops are assigned two separate stages `B` and `D`. A stage containing a dynamic-bound loop will be synthesized as part of an LPA while other stages will be pipelined using

conventional techniques. As such, ElasticFlow supports dependences among different outer loop iterations of all stages except `B` and `D` because `B` and `D` are bound to LPAs.

4.3.1 LPU Allocation

While ElasticFlow allows users to manually configure the LPA, we also provide an LPU allocation framework that allocates an appropriate number of LPUs to meet the throughput requirement of the design. Our LPU allocation framework consists of two main steps — We first profile the loop nests in the original design and predict the number of LPUs needed to meet the throughput target assuming only sLPUs are used. We then reallocate resources from sLPUs to mLPUs to achieve performance improvement by sharing LPUs among different inner loops.

The first step begins by pipelining each of the dynamic-bound inner loops in the original design. For each loop, we automatically extract two parameters from the results: the best achievable initiation interval of inner loop i, denoted as II_i, and the latency in cycles of a single iteration of loop i, denoted as L_i. We also instrument the original source code, compile, and profile the design using typical test vectors to obtain the average-case bound of inner loop i, denoted as B_i. If only sLPUs are used, we will allocate $U_i = [II_i \cdot (B_i - 1) + L_i] \cdot TP$ number of sLPUs for each inner loop i, where TP denotes the throughput requirement of the design in terms of number of outer loop iterations per cycle.

In the second step of our LPU allocation framework, we formulate an integer linear program (ILP) that optimizes the performance by reallocating sLPUs

to mLPUs given the resource usage of the sLPA architecture. Allocating more mLPUs further facilitates load balancing by allowing adaptive assignment of hardware resources for different loops. However, there is an inherent trade-off between performance and area since an mLPU typically consumes more area than an sLPU because it contains the hardware to execute multiple inner loops. The ILP formulation that explores this trade-off is shown in Equations (4.1a)-(4.1e).

Here we classify LPUs into different types depending on which loops share the particular LPU. For the example in Figure 4.5, LPUs are classified into three types: stage `B` only, stage `D` only, and stages `B/D`. The first two are sLPUs, and the latter is an mLPU. Since there are usually only a few inner loops, it is reasonable to enumerate the different types of LPUs. Given K types of LPAs, S_k^j denotes the area of resource j of an LPU in a type-k LPA and can be obtained from synthesizing each type of LPU individually. Given N inner loops, we also classify LPAs into N degrees where a degree-n LPA contains LPUs that can be shared among n loops.

In Equation (4.1a), D_k indicates the degree of a type-k LPA with type-k LPUs, denoted LPA k. A_{total}^j represents the area constraint of resource j and is derived as a user-specified fraction of the number of resources required to synthesize the baseline sLPA architecture. n_k is a nonnegative integer variable that represents the number of LPUs needed for LPA k. r_{ik} is a binary variable that represents whether loop i is bound to LPA k. $T(i)$ denotes the set of LPAs on which loop i can execute. Equation (4.1b) constrains the total area of all the allocated LPUs. Equation (4.1c) prevents over-allocation of LPUs. Equation (4.1d) ensures that each inner loop is allocated to only a single type of LPA. Equation (4.1e) prevents

the allocation of unused LPAs with C as a large constant, and Equation (4.1f) enforces that loops are mapped only to compatible LPAs. The objective of the optimization is to maximize the weighted sum of the total degree of sharing and the total number of LPUs, where α and β are the weights that can be defined by the user. This objective acts as a proxy for performance by balancing the degree of sharing with the number of LPUs.

$$\text{maximize } \alpha \sum_{k=1}^{K} \sum_{i=1}^{N} D_k r_{ik} + \beta \sum_{k=1}^{K} n_k \quad \text{subject to} \tag{4.1a}$$

$$\sum_{k=1}^{K} S_k^j n_k \leq A_{total}^j \ \ \forall j \tag{4.1b}$$

$$\sum_{i=1}^{N} U_i r_{ik} \geq n_k \ \ \forall k \tag{4.1c}$$

$$\sum_{k=1}^{K} r_{ik} = 1 \ \ \forall i \tag{4.1d}$$

$$C \sum_{i=1}^{N} r_{ik} >= n_k \tag{4.1e}$$

$$r_{ik} = 0 \ \ \forall k \notin T(i) \tag{4.1f}$$

4.3.2 Memory Bank Synthesis

For designs that access memory in the inner loop, we synthesize a banked memory system to serve the memory requests from all the LPUs in an LPA. The designer can either specify the banking scheme by overwriting the default banking policy, or use the default cyclic banking scheme as detailed below. Given an array of n elements, we partition the array into M banks in a cyclic fashion: the i^{th} element of the original array will be mapped to the $(i \mod M)^{th}$ bank. We empirically observe that the cyclic partitioning scheme tends to balance the load

of different banks in the common case.

Our synthesis tool allows the user to manually specify the number of memory banks in the memory system. In the case where the bank number is not specified, the tool automatically determines the memory configuration using the following procedure: for an inner loop that is pipelined to an initiation interval II with k memory accesses in a single inner loop iteration, we will have $\frac{kU}{II}$ memory accesses per cycle, where U is the number of LPUs in an LPA. To minimize stalling due to banking conflict, we set the number of banks to be $\frac{kU}{II}$, so that the number of banking conflicts is minimal in common cases. Further increasing the number of banks to provide higher memory bandwidth can potentially continue to reduce banking conflicts, but at the cost of increased hardware complexity. The procedure above achieves a balance between minimizing banking conflicts and reducing design complexity.

4.3.3 Distributor and Collector Synthesis

Each LPA contains a distributor and a collector for assigning incoming inner loop instances to, and gathering results from LPUs, respectively. The distributor contains a *scheduler* which employs a dynamic work distribution policy – when an inner loop instance is available to be executed, the scheduler evaluates the busy/idle states of the LPUs in parallel, and assigns the loop instance to an idle LPU with the smallest LPU ID. Figure 4.6 shows a code snippet in synthesizable C++ that describes the scheduler behavior. Specifically, the scheduler communicates with the upstream and downstream stages using messages through streams. The messages in the streams are type-templatized to account

```
template <typename DT, typename T >
static void Scheduler(
  // strm_out connects the scheduler and the
  // downstream LPUs
  stream<T> (&strm_out)[NUM_LPU],
  // strm_in connects the upstream stage and
  // the scheduler
  stream<DT> &strm_in) {
  // write_succeed indicates whether the
  // current work item has been written
  // to one of the LPUs
  bool write_succeed = false;
  // din: work item read from the upstream stage
  T din;
  // strm_in.read(): blocking read operation
  // operation from strm_in
  din.val = strm_in.read();
  din.id = 0;

  while (true) {
  #pragma pipeline
    write_succeed = false;
    // fixed-priority work distribution
    for (int i = 0; i < NUM_LPU; i++) {
    #pragma unroll
      // strm_out[i].write_nb(din): nonblocking
      // write operation to strm_out[i]
      if (strm_out[i].write_nb(din)) {
        write_succeed = true;
        break;
      }
    }
    // read in the next work item
    if(write_succeed)
      din.val = strm_in.read();
  }
}
```

Figure 4.6: **Code snippet for the scheduler in LPA.**

for various data types in different designs. At each clock cycle, the scheduler attempts to write a work item (i.e., outer loop iteration ID and the corresponding live-in values) into the LPUs using non-blocking write. If any one of the LPU succeeds in receiving the work item, the scheduler will read in a new work item from upstream. Otherwise, the scheduler temporarily stores the current work item and will retry in the next clock cycle.

For mLPAs, the scheduler gives priority to the inner loop with the lowest outer loop iteration ID when multiple inner loops are waiting to be scheduled.

As a result of dynamic loop bound, different inner loop instances may finish execution and exit their LPUs out of order. We synthesize an ROB in the collector to ensure that the results of the LPAs are produced in order. Each LPU stores its results to a location in the ROB in the order of increasing inner loop iteration ID, where the head of ROB stores the smallest loop iteration ID that is being processed. At each cycle, the ROB examines its head entry, and outputs the result if the head contains valid data. If the ROB is full, it applies back pressure to the distributor to prevent further work distribution until additional space frees up.

4.3.4 Buffer Sizing

It is important to suitably size the ROB to maximize the utilization of the LPUs. The distributor will be stalled when a long-latency loop iteration blocks the head of ROB. As a result, the LPUs cannot process new outer loop iterations, and the system becomes underutilized. However, there is no one-size-fits-all design since different applications may have drastically different loop latency patterns. Here we propose a profiling-driven approach to estimating the size of the ROB.

We profile a given application with representative sample datasets to obtain four key parameters for each dynamic-bound inner loop: the maximum latency L_{max}, the minimum latency L_{min}, the average-case latency L_{avg}, and the standard deviation σ of the loop latencies. Assume that there are K LPUs in the LPA, and consider the worst-case scenario where the head of ROB is blocked by

a loop with L_{max}. In the meantime, the other $(K-1)$ LPUs are free to continue executing other inner loop instances. We assume that these inner loop instances have a latency of $(L_{avg} - 3\sigma)$, where the -3σ term accounts for the latency deviation from the mean. In situations where $L_{avg} - 3\sigma < L_{min}$, we use L_{min} instead to avoid over-pessimistic estimations. We define a parameter S using the following equation:

$$S = \frac{L_{max}}{\texttt{max}(L_{avg} - 3\sigma, L_{min})}(K-1) + 1 \tag{4.2}$$

To increase LPU utilization by minimizing pipeline stalling, we require the ROB size to be no less than S. To reduce the logic overhead of ROBs, we round up S to the nearest power of two as the estimated ROB size.

A second consideration is the sizing of *delay lines*, which forward data that do not need to enter an LPA but must nevertheless proceed down the pipeline. A delay line is implemented as a FIFO that connects the stages before and after the LPA, and should be large enough to hold all in-flight data waiting to be consumed with the results from the LPA, or the pipeline will be stalled. The worst-case scenario for the delay line is similar to that of the ROB. Essentially, the delay line entries D should be no fewer than the number of possible in-flight loop instances in the LPA when the LPA is blocked by a loop instance with the maximum latency L_{max}, i.e., $D \geq S + K$. Compared with Equation (4.2), the additional term $+K$ corresponds to the fact that the K LPUs have also received new tasks by the time the long-latency loop instance finishes execution.

4.3.5 Deadlock Avoidance

ElasticFlow communicates live-in and live-out values in bundles between pipeline stages. For the sLPA architecture, artificial deadlock may occur if the number of in-flight outer loop iterations, consisting of those being processed by LPUs and those waiting in ROB, is greater than the total number of entries in the ROB. In this case, LPUs eventually stall because ROB does not have enough available entries to accept new data, and at the same time lacks all the data needed to proceed with reordering. To avoid such deadlock, we design the collector so that the number of in-flight outer loop iterations is limited to be no greater than the number of available entries in the ROB. This guarantees that every in-flight outer loop iteration will find an entry in the ROB once it finishes execution. As a result, the sLPAs are guaranteed to be deadlock free.

If we consider each sLPA as a single compute node, our dataflow architecture contains no cycles and forms a DAG. As proven in [62], such a system cannot deadlock if no inputs are filtered – an input to a node always results in an output. Because each live-in bundle always results in a live-out bundle per outgoing FIFO, the sLPAs are deadlock free and our system encounters no deadlock if it contains sLPAs only.

For mLPA architectures, we allocate one ROB for each inner loop sharing a particular mLPA, and limit the number of in-flight outer loop iterations of an inner loop to be no greater than the size of its corresponding ROB. If there is no data dependency between the shared inner loops, the resulting dataflow network forms a DAG, and the result from [62] guarantees no deadlock.

Now we show intuitively that our architecture remains deadlock free even

if there exists data dependency between the shared inner loops. Assume that the mLPA architecture encounters a deadlock caused by two dependent inner loops that share an mLPA. In this situation, the oldest uncommitted outer loop iteration of the producer loop will be held up at the head of the ROB of its mLPA because execution cannot proceed to the corresponding consumer loop, which does not have an LPU to run on. However, the restriction on the number of in-flight outer loop iterations dictates that an mLPU will eventually free up when younger instances of the producer loop finish execution and move to the ROB. The consumer loop will then be able to execute using one of these freed LPUs and consume the producer result. This contradicts with the assumption that the consumer loop has been blocked due to insufficient resources. Hence, we conclude that the mLPA architecture is also deadlock free.

4.4 Experimental Results

Our setup leverages a widely used commercial HLS tool, which uses the LLVM compiler [57] as its front end and compiles a behavioral C/C++ program into Verilog or VHDL targeting Xilinx FPGAs. To our best knowledge, the tool employs modulo scheduling to pipeline a function or a loop nest, where all inner loops must be completely unrolled. Dynamic-bound inner loops will therefore prevent the functions or loop nests from being pipelined. We implement our ElasticFlow synthesis algorithm as a source-to-source transformation in C/C++. The source-to-source transformation first converts the input C/C++ program into LLVM IR, and uses the technique in Section 4.3 to identify dynamic-bound inner loops. The transformation then synthesizes these dynamic-bound inner loops using the commercial HLS tool to obtain the achievable initiation interval

Table 4.1: **Descriptions of ElasticFlow Benchmarks.**

Design	Domains	Description
bgcd	Security	Binary GCD algorithm
cfd	Fluid dynamics	Computational fluid dynamics solver
dr	Image processing	KNN-based digit recognition
newton	Scientific computing	Iterative root finding method
bellman	Graph processing	Single-source shortest path
pagerank	Search engine	Website ranking algorithm
spavg	Scientific computing	Row mean value of sparse matrix
spmv	Scientific computing	Sparse matrix-vector multiplication
dbjoin	Database	Database join operation

and the latency of one inner loop iteration. The average-case loop bound under typical test vectors is obtained using LLVM built-in profiler. The user has the option to overwrite the profiling results with user-specified inputs to generate different architecture configurations. Given the above profiling results, we trigger a customized LLVM pass to place each loop nest into a separate function, and use the dataflow directive provided by the HLS tool to connect these functions as distinct dataflow stages. The live values between stages are passed between the corresponding functions as stream objects provided by the HLS tool. Finally, we convert the optimized IR to low-level C/C++ program using LLVM C/C++ backend, and feed the low-level C/C++ program to the same commercial HLS engine to perform RTL code generation. The generated Verilog RTL design is implemented by Xilinx Vivado 2015.3 targeting a Virtex-7 FPGA device with 5ns target clock period. All timing and area numbers are obtained post place and route. We evaluate ElasticFlow using a variety of real-life applications from search engine, graph processing, database, scientific computing, image processing, and security. Table 4.1 briefly describes these applications. Each application contains one or more dynamic-bound inner loops that the commercial HLS tool cannot effectively pipeline.

4.4.1 Design Space Exploration

We first explore the design space of ElasticFlow architecture by varying LPA configurations from two to eight LPUs and one to eight memory banks. Table 4.2 shows the performance and resource usage comparison between the baseline approach and ElasticFlow. The baseline design generated by the commercial HLS tool is equivalent to the ElasticFlow scheme with one LPU and a single-bank memory. We also vary the number of LPUs for each benchmark to study the performance-area trade-off for ElasticFlow. For designs that access memory in the LPUs, we create a multi-bank memory system for each array, and vary the number of memory banks to study the impact of memory architecture on performance and area. With our synthesis flow, the user either use the default LPU and memory allocation automatically generated by the tool, or manually specify the number of LPUs and the number of memory banks for each individual loop. Not surprisingly, for all designs, ElasticFlow consistently outperforms the baseline in terms of performance. We note that given enough memory bandwidth, increasing the number of LPUs proportionally improves the performance of most designs, where dynamic-bound inner loops bottleneck the throughput of the pipeline without ElasticFlow. In the cases where the designs become memory bound, we observe that by increasing the number of banks in the memory system, we can further improve the performance of these designs. We also observe that for a few benchmarks with 4 or 8 memory banks, the design failed to meet the 5ns timing constraint due to the complex multiplexer network used for bank arbitration. We envision that we could improve the timing of these designs by pipelining the multiplexer network. We also report the relative resource usage in the parentheses in Table 4.2 following the absolute resource count. We observe that the resource usage increases at

Table 4.2: **Performance and Resource Usage Comparison** – `base` is the design generated by the commercial HLS tool that is equivalent to the ElasticFlow scheme with one LPU and a single-bank memory. [`n`] represents the proposed `ElasticFlow` approach with `n` LPUs. [`n, m`] represents the proposed `ElasticFlow` approach with `n` LPUs and `m` memory banks. `CP` = clock period in ns (target `CP` is set to 5ns); `BRAM` = # of block RAMs; `DSP` = # of DSP blocks; `FF` = # of flip-flops; `LUT` = # of lookup tables; `LAT` = latency of the entire design in # of cycles; `Speedup` = speedup of `ElasticFlow` over `base`, in terms of the product of `LAT` and `CP`. Numbers in parentheses show the ratio between the specific configuration and the corresponding baseline design.

Design	CP	BRAM	DSP	FF	LUT	LAT	Speedup
bgcd-base	4.4	2 (1.0)	0 (1.0)	1459 (1.0)	1356 (1.0)	48768	
bgcd-[2]	4.4	2 (1.0)	0 (1.0)	1896 (1.3)	1628 (1.2)	24400	2.0
bgcd-[4]	4.6	2 (1.0)	0 (1.0)	2681 (1.8)	2219 (1.6)	12226	4.0
bgcd-[8]	4.6	2 (1.0)	0 (1.0)	4255 (2.9)	3361 (2.5)	6139	7.9
cfd-base	4.6	3 (1.0)	47 (1.0)	10868 (1.0)	9496 (1.0)	21131	
cfd-[2]	4.6	3 (1.0)	62 (1.3)	13788 (1.3)	11911 (1.3)	11325	1.9
cfd-[4]	4.7	3 (1.0)	92 (2.0)	19543 (1.8)	17022 (1.8)	6139	3.4
cfd-[8]	4.7	3 (1.0)	152 (3.2)	31090 (2.9)	26880 (2.8)	3427	6.2
dr-base	4.4	57 (1.0)	0 (1.0)	774 (1.0)	982 (1.0)	322705	
dr-[2]	4.4	57 (1.0)	0 (1.0)	1135 (1.5)	1222 (1.2)	161363	2.0
dr-[4]	4.6	57 (1.0)	0 (1.0)	1820 (2.4)	1737 (1.8)	80699	4.0
dr-[8]	4.9	57 (1.0)	0 (1.0)	3194 (4.1)	2799 (2.9)	40372	8.0
newton-base	4.4	4 (1.0)	17 (1.0)	6573 (1.0)	6630 (1.0)	47667	
newton-[2]	4.7	4 (1.0)	34 (2.0)	12650 (1.9)	12596 (1.9)	24161	2.0
newton-[4]	4.8	4 (1.0)	68 (4.0)	24651 (3.8)	24488 (3.7)	12454	3.8
newton-[8]	4.9	4 (1.0)	136 (8.0)	48656 (7.4)	48337 (7.3)	8764	5.4
bellman-base	4.3	25 (1.0)	0 (1.0)	1285 (1.0)	1151 (1.0)	6995	
bellman-[2,1]	4.5	25 (1.0)	0 (1.0)	2386 (1.9)	1657 (1.4)	6515	1.1
bellman-[2,2]	4.3	23 (0.9)	0 (1.0)	2514 (2.0)	1878 (1.6)	3694	1.9
bellman-[2,4]	4.5	23 (0.9)	0 (1.0)	2617 (2.0)	2176 (1.9)	3662	1.9
bellman-[2,8]	4.4	35 (1.4)	0 (1.0)	2787 (2.2)	2734 (2.4)	3636	1.9
bellman-[4,1]	4.6	26 (1.0)	0 (1.0)	4237 (3.3)	2682 (2.3)	6517	1.1
bellman-[4,2]	4.5	24 (1.0)	0 (1.0)	4480 (3.5)	3069 (2.7)	3498	2.0
bellman-[4,4]	4.5	24 (1.0)	0 (1.0)	4710 (3.7)	3480 (3.0)	2038	3.4
bellman-[4,8]	4.8	36 (1.4)	0 (1.0)	5547 (4.3)	4317 (3.8)	1974	3.5
bellman-[8,1]	4.9	28 (1.1)	0 (1.0)	7885 (6.1)	4743 (4.1)	6403	1.1
bellman-[8,2]	4.7	26 (1.0)	0 (1.0)	8224 (6.4)	5388 (4.7)	3465	2.0
bellman-[8,4]	4.7	26 (1.0)	0 (1.0)	8775 (6.8)	5985 (5.2)	1966	3.6
bellman-[8,8]	5.0	38 (1.5)	0 (1.0)	10034 (7.8)	7468 (6.5)	1172	6.0

Design	CP	BRAM	DSP	FF	LUT	LAT	Speedup
pagerank-base	4.2	12 (1.0)	0 (1.0)	1337 (1.0)	1074 (1.0)	7612	
pagerank-[2,1]	4.4	12 (1.0)	0 (1.0)	2268 (1.7)	1529 (1.4)	7078	1.1
pagerank-[2,2]	4.5	12 (1.0)	0 (1.0)	2323 (1.7)	1627 (1.5)	4186	1.8
pagerank-[2,4]	4.4	12 (1.0)	0 (1.0)	2361 (1.8)	1763 (1.6)	4187	1.8
pagerank-[2,8]	4.4	20 (1.7)	0 (1.0)	2428 (1.8)	2017 (1.9)	4129	1.8
pagerank-[4,1]	4.5	14 (1.2)	0 (1.0)	3985 (3.0)	2459 (2.3)	7001	1.1
pagerank-[4,2]	4.6	14 (1.2)	0 (1.0)	4102 (3.1)	2632 (2.5)	3899	2.0
pagerank-[4,4]	4.5	14 (1.2)	0 (1.0)	4208 (3.1)	2827 (2.6)	2355	3.2
pagerank-[4,8]	4.6	22 (1.8)	0 (1.0)	4577 (3.4)	3218 (3.0)	2424	3.1
pagerank-[8,1]	4.7	18 (1.5)	0 (1.0)	7394 (5.5)	4329 (4.0)	6736	1.1
pagerank-[8,2]	4.7	18 (1.5)	0 (1.0)	7559 (5.7)	4607 (4.3)	3862	2.0
pagerank-[8,4]	4.6	18 (1.5)	0 (1.0)	7803 (5.8)	4894 (4.6)	2388	3.2
pagerank-[8,8]	5.1	26 (2.2)	0 (1.0)	8398 (6.3)	5653 (5.3)	1508	5.0
spavg-base	4.3	11 (1.0)	0 (1.0)	3014 (1.0)	2775 (1.0)	6657	
spavg-[2,1]	4.2	11 (1.0)	0 (1.0)	3765 (1.2)	3182 (1.1)	6309	1.1
spavg-[2,2]	4.3	11 (1.0)	0 (1.0)	3839 (1.3)	3306 (1.2)	3423	1.9
spavg-[2,4]	4.5	11 (1.0)	0 (1.0)	3900 (1.3)	3451 (1.2)	3417	1.9
spavg-[2,8]	4.6	11 (1.0)	0 (1.0)	3997 (1.3)	3747 (1.4)	3404	2.0
spavg-[4,1]	4.4	11 (1.0)	0 (1.0)	5126 (1.7)	3987 (1.4)	6335	1.1
spavg-[4,2]	4.5	11 (1.0)	0 (1.0)	5256 (1.7)	4195 (1.5)	3339	2.0
spavg-[4,4]	4.4	11 (1.0)	0 (1.0)	5372 (1.8)	4375 (1.6)	1796	3.7
spavg-[4,8]	4.7	11 (1.0)	0 (1.0)	5832 (1.9)	4834 (1.7)	1788	3.7
spavg-[8,1]	4.8	11 (1.0)	0 (1.0)	7823 (2.6)	5613 (2.0)	6293	1.1
spavg-[8,2]	4.8	11 (1.0)	0 (1.0)	8009 (2.7)	5967 (2.2)	3342	2.0
spavg-[8,4]	4.7	11 (1.0)	0 (1.0)	8311 (2.8)	6225 (2.2)	1806	3.7
spavg-[8,8]	5.3	11 (1.0)	0 (1.0)	8938 (3.0)	7040 (2.5)	1040	6.4
spmv-base	4.3	15 (1.0)	3 (1.0)	1337 (1.0)	1166 (1.0)	7487	
spmv-[2,1]	4.6	15 (1.0)	6 (2.0)	2287 (1.7)	1663 (1.4)	7046	1.1
spmv-[2,2]	4.4	15 (1.0)	6 (2.0)	2420 (1.8)	1888 (1.6)	4025	1.9
spmv-[2,4]	4.4	15 (1.0)	6 (2.0)	2566 (1.9)	2171 (1.9)	4062	1.8
spmv-[2,8]	4.5	19 (1.3)	6 (2.0)	2714 (2.0)	2735 (2.3)	4000	1.9
spmv-[4,1]	4.6	16 (1.1)	12 (4.0)	3994 (3.0)	2664 (2.3)	6950	1.1
spmv-[4,2]	4.5	16 (1.1)	12 (4.0)	4246 (3.2)	3071 (2.6)	3821	2.0
spmv-[4,4]	4.6	16 (1.1)	12 (4.0)	4463 (3.3)	3433 (2.9)	2210	3.4
spmv-[4,8]	4.7	20 (1.3)	12 (4.0)	5268 (3.9)	4316 (3.7)	2256	3.3
spmv-[8,1]	4.8	18 (1.2)	24 (8.0)	7354 (5.5)	4693 (4.0)	6688	1.1
spmv-[8,2]	4.7	18 (1.2)	24 (8.0)	7720 (5.8)	5387 (4.6)	3789	2.0
spmv-[8,4]	4.8	18 (1.2)	24 (8.0)	8257 (6.2)	5991 (5.1)	2262	3.3
spmv-[8,8]	6.7	22 (1.5)	24 (8.0)	9408 (7.0)	7529 (6.5)	1373	5.5

Design	CP	BRAM	DSP	FF	LUT	LAT	Speedup
dbjoin-base	4.5	49 (1.0)	0 (1.0)	4250 (1.0)	4607 (1.0)	223240	
dbjoin-[2,1]	4.6	57 (1.2)	0 (1.0)	6519 (1.5)	7266 (1.6)	180752	1.2
dbjoin-[2,2]	4.5	58 (1.2)	0 (1.0)	7477 (1.8)	8435 (1.8)	102774	2.2
dbjoin-[2,4]	4.8	62 (1.3)	0 (1.0)	8101 (1.9)	9284 (2.0)	90638	2.5
dbjoin-[2,8]	5.0	66 (1.3)	0 (1.0)	9631 (2.3)	10896 (2.4)	90606	2.5
dbjoin-[4,1]	4.8	73 (1.5)	0 (1.0)	10945 (2.6)	12611 (2.7)	173006	1.3
dbjoin-[4,2]	4.8	74 (1.5)	0 (1.0)	12226 (2.9)	14354 (3.1)	89720	2.5
dbjoin-[4,4]	4.8	86 (1.8)	0 (1.0)	13255 (3.1)	15301 (3.3)	49300	4.5
dbjoin-[4,8]	7.9	98 (2.0)	0 (1.0)	15128 (3.6)	17719 (3.8)	45688	4.9
dbjoin-[8,1]	5.0	105 (2.1)	0 (1.0)	19840 (4.7)	23514 (5.1)	171502	1.3
dbjoin-[8,2]	5.1	106 (2.2)	0 (1.0)	21695 (5.1)	26698 (5.8)	87622	2.5
dbjoin-[8,4]	5.1	118 (2.4)	0 (1.0)	21214 (5.0)	27299 (5.9)	46622	4.8
dbjoin-[8,8]	5.6	114 (2.3)	0 (1.0)	22167 (5.2)	31271 (6.8)	29134	7.7
kmp-base	4.5	16 (1.0)	0 (1.0)	2072 (1.0)	1752 (1.0)	1024197	
kmp-[2,1]	4.6	16 (1.0)	0 (1.0)	3145 (1.5)	2542 (1.5)	542139	1.9
kmp-[2,2]	4.7	16 (1.0)	0 (1.0)	3204 (1.5)	2646 (1.5)	542113	1.9
kmp-[2,4]	4.5	18 (1.1)	0 (1.0)	3240 (1.6)	2807 (1.6)	542089	1.9
kmp-[2,8]	4.5	18 (1.1)	0 (1.0)	3311 (1.6)	3098 (1.8)	542064	1.9
kmp-[4,1]	4.7	16 (1.0)	0 (1.0)	5142 (2.5)	4157 (2.4)	520292	2.0
kmp-[4,2]	4.8	16 (1.0)	0 (1.0)	5268 (2.5)	4346 (2.5)	294857	3.5
kmp-[4,4]	4.8	18 (1.1)	0 (1.0)	5387 (2.6)	4559 (2.6)	279991	3.7
kmp-[4,8]	4.6	18 (1.1)	0 (1.0)	5701 (2.8)	5037 (2.9)	279991	3.7
kmp-[8,1]	4.9	16 (1.0)	0 (1.0)	9111 (4.4)	7407 (4.2)	552594	1.9
kmp-[8,2]	4.9	16 (1.0)	0 (1.0)	9280 (4.5)	7686 (4.4)	303773	3.4
kmp-[8,4]	4.8	18 (1.1)	0 (1.0)	9490 (4.6)	8035 (4.6)	198119	5.2
kmp-[8,8]	5.7	18 (1.1)	0 (1.0)	10287 (5.0)	8793 (5.0)	198089	5.2

most linearly as the number of LPUs, indicating that the ElasticFlow architecture indeed efficiently utilizes the available hardware resources.

4.4.2 Comparison with EF-Replicate

Here we compare the performance and area between the proposed architecture with memory banking and our previous work that uses memory replication [97], to which we refer as EF-Replicate for the sake of convenience.

For this specific comparison, this chapter and EF-Replicate both use four

Table 4.3: **Comparison with EF-Replicate [97]** – Latency and resource comparsion for six benchmarks that access memory in the inner loop. Four LPUs are used for EF-Replicate and this work. EF-Replicate uses replicated local memroies while this work uses a four-bank memory system.

	EF-Replicate [97]						This Work					
Design	CP	BRAM	DSP	FF	LUT	Latency	CP	BRAM	DSP	FF	LUT	Latency
bellman	4.6	52	0	3930	2545	1972	4.5	24	0	4710	3480	2038
pagerank	4.6	28	0	3836	2394	2364	4.6	14	0	4208	2827	2355
spavg	4.7	25	0	4977	3918	1786	4.4	11	0	5372	4375	1796
spmv	4.6	36	12	3688	2514	2233	4.6	16	12	4463	3433	2210
dbjoin	4.9	242	0	9794	12236	45498	4.8	86	0	13255	15301	49300
kmp	5.0	32	0	4984	4118	279991	4.8	18	0	5387	4559	279991
Ratio	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.41	1.00	1.20	1.23	1.01

LPUs. We also use a 4-bank memory system, while EF-Replicate duplicates local memories four times. We have rerun EF-Replicate on the updated datasets used in this chapter, and the results are shown in Table 4.3. We leave out designs that do not access memory in the LPUs, since there is no difference in final results. Across all six benchmarks that access memory in the inner loops, we achieve almost identical performance with EF-Replicate in terms of clock period and total latency, while significantly reducing the BRAM usage by 60% on average. The flip-flop and LUT overheads are 23% and 20% on average, respectively, which are due to the control and arbitration logic in the memory controller.

To further investigate the performance of the multi-bank memory system across different numbers of LPU and memory banks, we provide a more detailed study of `spmv` under various real-life input vectors. Table 4.4 summaries the performance of `spmv` in terms of execution latency normalized to the reference design where each LPU has its local replicated memory. The test vectors are real-life sparse matrices taken from the University of Florida Sparse Matrix Collection [32] covering application domains spanning electromagnetism, social network, fluid dynamics, robotics, and combinatorics. We observe that by having sufficient number of memory banks in the memory system, the performance degradation can be reduced to within 10% of the replication approach for all the test vectors. The memory system achieves comparable performance with the replication approach when the number of memory banks is no smaller than the number of LPUs. This is because each LPU is pipelined to $II = 1$ and sends out at most one memory request per cycle. As a result, an LPA with n LPUs will initiate at most n memory requests per cycle. Thus we need at least n memory banks to satisfy the memory bandwidth requirement.

Table 4.4: **Performance comparisons of memory banking and memory replication with real-life test vectors from various domains** – qc324 (electromagnetics), Journals (network), ex2 (fluid dynamics), rbsb480 (robotics), and CAG_mat364 (combinatorics). Performance is reported as execution latency normalized to the replication method.

Design	Replication	1 Bank	2 Banks	4 Banks	8 Banks
		2 LPUs			
qc324	1.00	0.56	1.00	1.00	1.00
Journals	1.00	0.55	0.99	1.00	1.00
ex2	1.00	0.56	0.99	1.00	1.00
rbsb480	1.00	0.58	0.99	1.00	1.00
CAG_mat364	1.00	0.59	1.00	0.99	1.00
		4 LPUs			
qc324	1.00	0.29	0.57	0.99	1.00
Journals	1.00	0.29	0.54	0.97	0.98
ex2	1.00	0.30	0.59	0.97	0.99
rbsb480	1.00	0.30	0.57	0.98	1.00
CAG_mat364	1.00	0.31	0.59	1.00	0.99
		8 LPUs			
qc324	1.00	0.16	0.30	0.58	0.97
Journals	1.00	0.18	0.33	0.59	0.98
ex2	1.00	0.17	0.30	0.55	0.94
rbsb480	1.00	0.17	0.32	0.57	0.93
CAG_mat364	1.00	0.22	0.41	0.71	0.92

4.4.3 Comparison with Aggressive Unrolling

An alternative approach to pipelining the outer loop is to fully unroll all inner loops using the worst-case loop bound as the unroll factor. Obviously, such a complete unrolling approach can lead to good performance, but would incur significant area overhead. In addition, we argue that such unrolling approach is unsafe or impractical for two reasons. Firstly, the worst-case loop bound simply does not exist for many applications. For example, in many iterative numerical applications, the loop bound of the dynamic loops varies vastly for different input datasets and the precision requirement. The user has little prior knowledge of the loop bound, thus unrolling the dynamic-bound loop to a user-specified

degree may violate the correctness of the program. Secondly, even if the user indeed knows the true worst-case loop bound, the unroll approach may still be impractical because the unrolled design is too large to be efficiently implemented. For example, in many sparse matrix-related applications, the worst-case loop bound will be proportional to the dimension of the matrix. Such inner loops will be too expensive to unroll for large matrices.

Here we show that ElasticFlow can achieve similar performance as the aggressive unrolling approach even for applications with small datasets that can be unrolled with reasonable amount of resource. Table 4.5 shows the results for two benchmarks `dbjoin` and `spmv`. Here we carefully select datasets with reasonably small worst-case loop bounds so that the unrolling approach can be practically implemented. Based on profiling results of the specific datasets, the unrolling factor for `dbjoin` is chosen to be 120, and `spmv` is unrolled by 100 times. According to Table 4.5, ElasticFlow achieves similar performance with aggressive unrolling, but requires significantly less resource usage (by 3-4x). This is because the average-case inner loop count is much smaller than the worst-case inner loop count in these applications. As a result, the aggressive unroll approach, whose resources are over-provisioned for the worse case, does not significantly improve the average-case performance.

4.4.4 Reorder Buffer Sizing

Figure 4.7 demonstrates the performance impact of ROB sizing as well as the effectiveness of the ROB size estimation scheme described in Section 4.3.4. To simplify the ROB logic, we restrict the buffer size to be a power of two. As

Table 4.5: **Elasticflow vs. Aggressive Unrolling Comparison** – The unroll* approach applies a user-specified worst-case unroll factor based on the profiling results of the worst-case loop bounds (120 for `dbjoin` and 100 for `spmv`), which may be unsafe if the actual loop bound exceeds the user-specified unroll factor.

Design		LAT	CP	FF	LUT
dbjoin	Unroll*	386	4.5	66250	25055
	ElasticFlow	389	4.8	13255	15301
spmv	Unroll*	365	4.4	19211	5008
	ElasticFlow	372	4.5	4295	3289

illustrated by the figure, our profiling-based approach for estimating the buffer size can reasonably predict the optimal ROB sizes that are free from stalling. For `bgcd` and `cfd`, the estimation matches exactly with the optimal ROB size. For `dbjoin` and `spavg`, the estimation provides a reasonable upper bound of the buffer size, which still ensures that the LPAs can achieve the best performance.

4.4.5 LPU Allocation

To validate the efficacy of the LPU allocation algorithm, we compare the user-specified throughput target with the measured throughput from designs generated from the ILP. Here we define the throughput target as five times the throughput of the baseline designs (designs with one LPU and a single-bank memory) in Table 4.2. Note that in practice, the reference throughput target can be specified based on the requirement of the specific design, and we choose to use these specific set of throughput numbers simply as an example to verify the ILP. The inputs to the ILP are the reference throughput target and loop profiling result.

Table 4.6 shows the LPU allocation results. For clear illustration, we translate

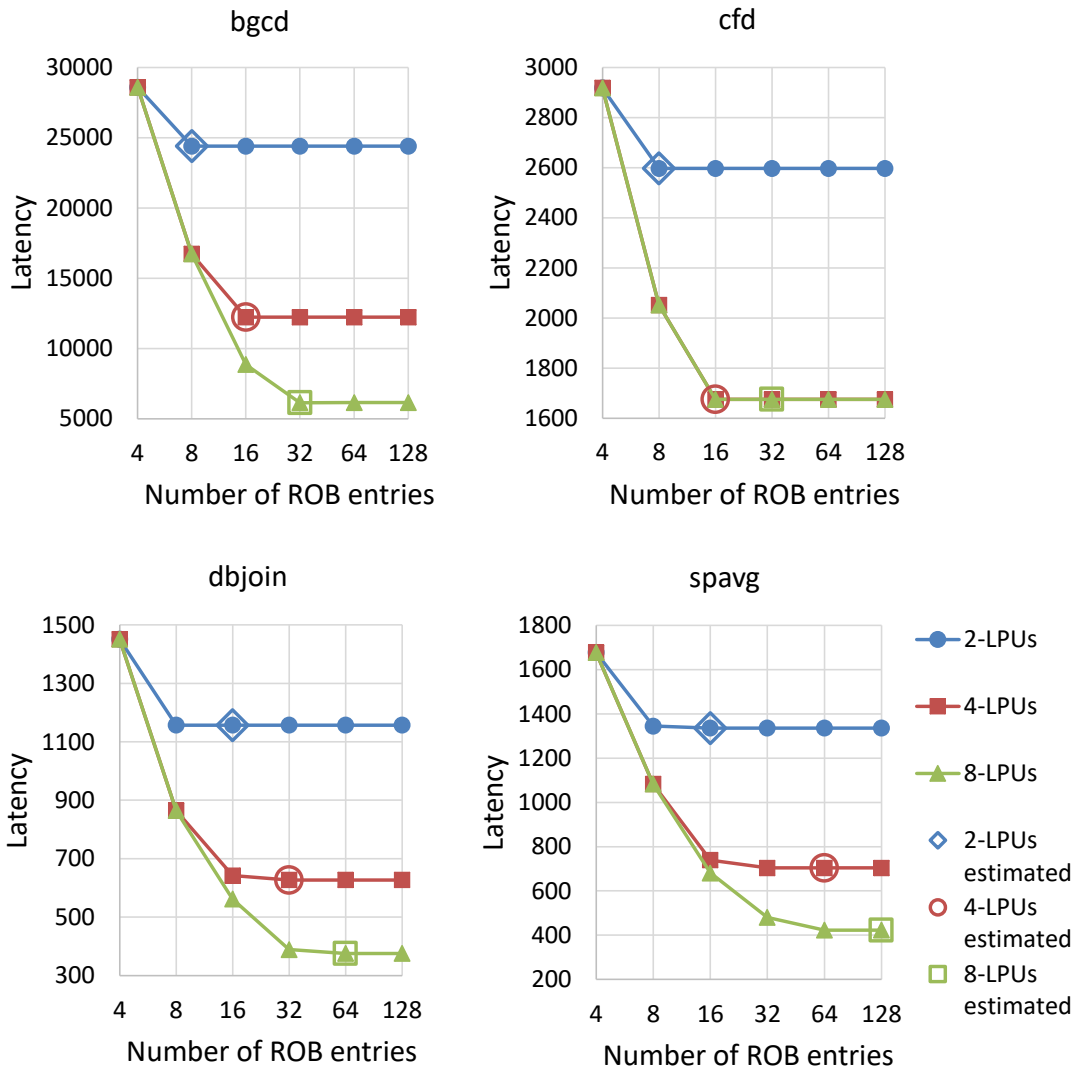


Figure 4.7: **Performance impact of ROB size and the estimated ROB size using Equation** (4.2)**.**

the throughput values into latency numbers as shown in Table 4.6. We calculate the relative latency difference between the measured latency and the target latency. We also list the predicted LPU numbers from the ILP, and the overall synthesis time including the profiling time to run the commercial HLS tool and the time to solve the ILP. We observe that the measured latency of designs generated from the LPU allocation algorithm closely matches the latency target, with

Table 4.6: **Validation of LPU allocation formulation** – We target achiving 5x speedup over the baseline designs. `Baseline Latency` = Latency of baseline design in number of cycles with one LPU and no memory banking; `Target Latency` = The latency target in number of cycles, defined as 1/5 of `Baseline latency`; `Measured Latency` = Measured latency from ILP-generated designs; `Diff` = Relative difference between maesured latency and target latency; `Syntheis Time` = Runtime of HLS tool + ILP.

Design	Baseline Latency	Target Latency	Measured Latency	Diff (%)	Predicted LPU#	Synthesis Time (s)
bgcd	48768	9754	9793	0.4%	5	28.9
cfd	21131	4226	4289	1.5%	6	43.7
dr	322705	64541	61595	-4.6%	5	56.5
newton	47667	9533	9718	1.9%	5	27.7
bellman	6995	1399	1416	1.2%	5	37.8
pagerank	7612	1522	1598	5.0%	5	34.5
spavg	6657	1331	1266	-4.9%	5	36.6
spmv	7487	1497	1622	8.3%	5	38.9
dbjoin	223240	44648	45498	1.9%	4	128.2
kmp	1024197	204839	222773	8.8%	5	42.1

an 8.8% deviation in the worst case. This shows the accuracy of our LPU allocation algorithm. The small deviations from the target latency are mostly due to the memory banking contentions that are not currently captured by the ILP formulation.

We also observe that for all the designs considered here, the run time of the ILP is all within one second using a state-of-the-art linear programming solver [24], making the runtime overhead of ILP negligible compared to the much longer CAD tool runtime.

Table 4.7: **LPU Resource Sharing** – Latency reduction and resource overheads for `cfd` and `dbjoin`. `cfd-A`: 8 mLPUs vs. 8 sLPUs; `cfd-B`: 16 mLPUs vs. 16 sLPUs; `dbjoin-A`: 7 mLPUs vs. 8 sLPUs; `dbjoin-B`: 14 mLPUs vs. 16 sLPUs.

Design	LAT Reduction	LUT Overhead	FF Overhead
cfd-A	34.7%	9.3%	3.3%
cfd-B	31.5%	8.9%	3.5%
dbjoin-A	21.3%	9.7%	-10.5%
dbjoin-B	21.6%	9.8%	-12.7%

4.4.6 LPU Sharing

We evaluate the effectiveness of our LPU sharing technique with two representative designs which contain more than one inner loop. `dbjoin` comprises two pointer-chasing inner loops that can share the same array of LPUs as shown in Figure 4.5(a); `cfd` contains two dynamic-bound inner loops that perform similar iterative fluid dynamics computations. As detailed in Section 4.3.1, we enforce an area constraint such that the total area of the mLPA designs are similar to that of the corresponding sLPA designs.

Table 4.7 demonstrates the latency reduction and resource usage of LPU sharing. For each benchmark, we provide two design points where the comparable sLPA design contains 8 or 16 LPUs. In addition, we are able to have equal numbers of mLPUs for `cfd`, since its two inner loops are structurally similar. For `dbjoin`, we allocate seven mLPUs (`dbjoin-A`) and 14 mLPUs (`dbjoin-B`), respectively. As shown in Table 4.7, using m-LPA can further improve the performance by 21%–34% with similar area, showing the effectiveness of the LPU sharing technique. For `dbjoin`, the mLPA designs require fewer FFs as we observe that many flip-flops are mapped to shift register LUTs (i.e., SRLs).

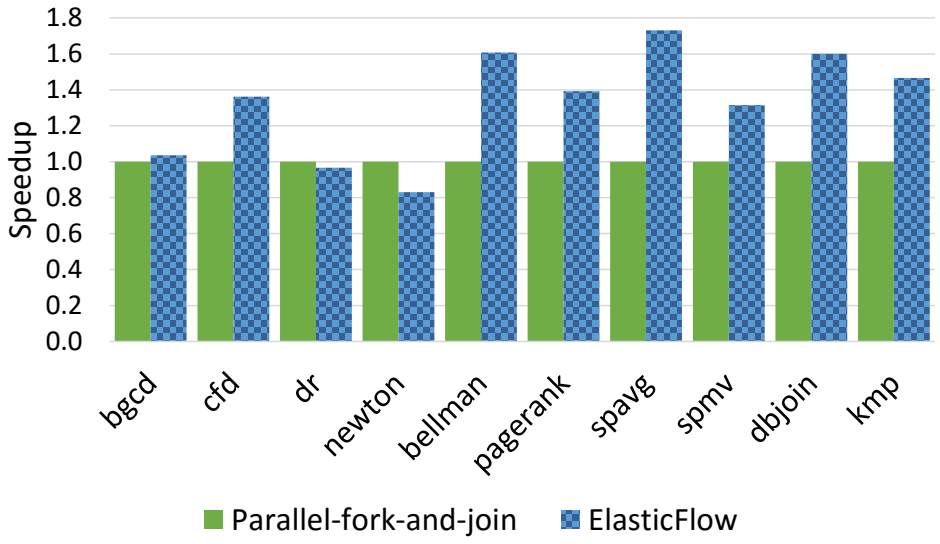


Figure 4.8: **Performance comparison between ElasticFlow and parallel-fork-and-join** – `ElasticFlow` is our approach; `Parallel-fork-and-join` implements the CGPA architecture [63] using sLPAs. We use eight LPUs for all the designs.

4.4.7 Comparison with CGPA

Related to our work, the Coarse-Grained Pipelined Accelerators (CGPA) architecture [63] also exploits decoupled pipeline parallelism, which enables outer loop pipelining using a hardware structure called parallel-fork-and-join. The parallel-fork distributes inner loop instances based on a predetermined mapping between outer loop iteration IDs and the available loop processing units. The parallel-join collects outputs only after all loop processing units have produced valid results. The parallel-fork-and-join architecture implements a generic cache system with a fixed number of ports, which may become the performance bottleneck if the memory bandwidth requirement of the application exceeds the maximum bandwidth of the cache system. We note that this parallel-fork-and-join scheme can be implemented using sLPAs with a special distributor-collector scheme. In the distributor-collector scheme, the distribu-

tor in the sLPA uses the static scheduling policy illustrated in Figure 4.4(b), and the collector delays the outputs until all LPUs have finished execution. With parallel-join, ROB is not used although we still need to allocate sufficient amount of buffers in the collector to temporarily hold the results from LPUs.

Figure 4.8 shows the performance comparison of ElasticFlow and the parallel-fork-and-join architecture. All the designs have eight LPUs, and the parallel-fork-and-join architecture uses a four-port cache. The number of memory banks in ElasticFlow is determined using the technique in Section 4.3.2. We observe that ElasticFlow can achieve up to 1.7x performance improvement compared to the alternative approach that resembles CGPA. We also observe that the two approaches have similar resource usage. The source of performance improvement of ElasticFlow over the parallel-fork-and-join architecture is twofold: Firstly, the dynamic scheduling and LPU sharing techniques in ElasticFlow enable better load balancing and resource utilization of the LPUs. Secondly, while the parallel-fork-and-join architecture is constrained by a limited, fixed number of memory ports, ElasticFlow employs a flexible multi-bank memory system where the number of memory banks can be configured to accommodate the memory bandwidth requirement of the application. In `dr` and `newton`, the performance of ElasticFlow is slightly worse than the parallel-fork-and-join architecture because the work of these two benchmarks are relatively balanced, so that dynamic scheduling does little help further balancing the load, while the overhead of dynamic scheduling leads to longer latency.

4.5 Related Work

Loop pipelining is typically enabled by modulo scheduling [84], a software pipelining technique for extracting instruction-level parallelism across loop iterations, and is an important optimization implemented in various academic and commercial HLS tools, including Vivado HLS [20], Altera SDK for OpenCL [26], and LegUp [15]. Recent advances in pipeline flushing [29], multithreading [95], and runtime dependency analysis [1] aim to achieve even higher-performance designs, along with techniques to optimize area by reducing the usage of registers [111], LUTs [94, 113], and memory ports [9]. Despite these optimizations, existing approaches mostly focus on pipelining simple loops and are ineffective for more complex loop nests seen in many programs.

Like simple loop pipelining, nested loop pipelining was first introduced in the software domain [83] and later extended to hardware synthesis [79]. Recent optimizations in HLS employ polyhedral analysis to enable automatic parallelization, streaming, and data reuse of regular nested loops with affine data access patterns [80]. Such compiler analysis performs polyhedral code transformation to optimize the design for performance and area based on the program pattern. However, polyhedral analysis is performed at compile time and is mostly useful for regular loop nests with static bounds. Lattuada and Ferrandi propose a technique to parallelize irregular loop nests by unrolling the outer loop and vectorizing certain instructions [58]. However, this technique requires that the inner loop bound does not depend on the outer loop iteration, making it inapplicable to our benchmarks.

The CGPA framework generates coarse-grained pipelines for a loop nest by

partitioning it into parallel and non-parallel sections [63]. CGPA employs replicated data-level parallelism to create multiple identical copies of the parallel section and applies decoupled pipeline parallelism to separate the parallel and sequential sections with a set of FIFOs. We can view ElasticFlow as a generalization of the CGPA framework which achieves additional performance improvement by enabling out-of-order execution and dynamic scheduling of inner loop instances. In addition, ElasticFlow realizes better resource efficiency and potentially higher throughput by optimizing the allocation and sharing of LPUs with the mLPA architecture. We also study buffer sizing for both the reorder buffer and the delay line, and propose a runtime policy that guarantees the absence of deadlock.

Kocberber et al. propose Widx, a reconfigurable accelerator which uses decoupled pipeline architecture for hash indexing in database systems [55]. In Widx, a hashing unit distributes work to a parallel array of walker units, with the results combined in an output unit. Our work is similar to Widx in that we also use decoupled dataflow pipelines containing a distributor, parallel processing units, and a collector. We distinguish our work from Widx in the following aspects: (1) Widx makes use of general-purpose RISC cores for accelerating database applications, while we focus on generating application-customized pipelines, which can potentially achieve higher performance and energy efficiency. We have validated our approach on a set of applications from multiple different domains. (2) We propose profiling-driven synthesis techniques for specifying various architectural parameters including number of LPUs, memory bank configuration, reorder buffer size, and LPU sharing. (3) While Widx focuses on in-order dispatch, our dynamic scheduling policy enables out-of-order dispatch. According to our experimental result in Figure 4.8, where the

parallel-fork-and-join scheme implements in-order dispatch, out-of-order dispatch results in a higher performance.

4.6 Conclusions

We presented ElasticFlow, a novel hardware architecture and associated synthesis techniques, which can efficiently address the pipelining of irregular loops nests containing dynamic-bound inner loops. ElasticFlow generates a dataflow pipeline architecture containing arrays of loop processing units, on which multiple instances of inner loops can be executed concurrently. We further study the complications of enabling out-of-order loop execution to improve throughput, and propose adaptive resource sharing schemes that enable the reallocation of loop execution units during runtime in response to workload imbalance for improved resource efficiency. Experimental results over a variety of real-life benchmarks show that ElasticFlow is able to achieve substantial performance improvement over a best-in-class commercial HLS tool.

CHAPTER 5

ASSIST: ARCHITECTURAL SYNTHESIS SYSTEM FOR INSTRUCTION SET TARGETS

Traditionally, architectural design for microprocessors is a highly manual process, which usually requires designers to specify the detailed cycle-level behavior at register-transfer level. However, two increasingly popular trends in computer architecture research motivate the need for automatic synthesis from a higher level of abstraction for ISA-defined architectures. Firstly, there is an increasing popularity in domain-specific processors with customized ISAs that deliver superior performance than the general-purpose processors while still maintaining software programmability. Recent representative examples in image processing and machine learning include the Pixel visual core [115], the Tensor Processing Unit [51] and Microsoft's Catapult project [81]. Secondly, The advent of free and open ISAs and their corresponding software and hardware ecosystems open up opportunities for fast deployment of low-cost and customizable microprocessors that are specifically tailored for the targeted domains such as embedded processors and Internet-of-things devices [5, 7].

This wave of new applications brought by domain-specific processors and open ISAs poses significant challenges on the EDA tools that assist the design of such processors. On one hand, different application domains have drastically different performance, power, and area requirements. There is no one-size-fits-all microarchitecture that satisfies the need for all relevant application domains. On the other hand, these emerging applications are usually from highly competitive domains that require fast time-to-market with limited budget for design, verification and fabrication.

High-level synthesis (HLS) is a well-studied technique to synthesize hardware designs from behavioral descriptions in high-level languages such as C/C++. Current HLS tools are designed for fixed-function circuits. While it is possible to describe an ISA and synthesize the corresponding hardware using HLS, the QoR of the synthesized designs is significantly worse than their manually-optimized counterparts (Section 5.1).

In this chapter, we address the problem of synthesizing high-quality, in-order, pipelined processors from an instruction set specification. We name the collection of our techniques as ASSIST, standing for Architectural Synthesis System for Instruction Set Targets.

We summarize the major techniques and contributions as follows:

1. We propose a set of micro-operations (micro-ops) defined in Python for designers to specify the instruction set in an intuitive fashion. Compared to generic Python code, restricting the input scope to pre-defined micro-ops facilitates the synthesis of efficient hardware logic for the processor designs.
2. We propose a set of synthesis techniques to generate high-quality datapath and control logic from user-defined instruction sets with optimized pipeline schedule and performance-optimized forwarding and hazard resolution logic.
3. Using the proposed synthesis flow, we explore the design space of processor microarchitectures with different pipeline schedules across three different technologies. Examples include the fully automatic synthesis of more than 50 different pipeline schedules for the RISC-V 32I ISA, many of which appear on the Pareto-optimal frontier of cycle time, execution time,

Table 5.1: **Summary of the MIPS processor design generated from the HLS tool.**

Initiation interval	4
Pipeline depth	5
Number of pipeline FSM stages	7
Number of 32-bit multiplier	2
Number of 32-bit adder/substractor	6

and area tradeoff curves.

4. We present the case study of using ASSIST to synthesize a cryptographic instruction set extension from the instruction set specification and demonstrate the QoR improvement over the base processor.

5.1 Drawbacks of Current High-Level Synthesis Tools

As a motivational example, we study the synthesis of a simple MIPS processor from a C specification using a state-of-the-art commercial HLS tool [107], which compiles an input program into a control data flow graph (CDFG), then synthesizes the datapath and the finite state machine (FSM) from the CDFG. We use the HLS tool to synthesize the benchmark `MIPS` in the HLS benchmark suite CHStone [46] into RTL, and observe the QoR reported from the HLS tool. Figure 5.1 shows the code snippet for the MIPS benchmark, which uses a `while` loop with the pipeline pragma to describe the functionality the MIPS processor pipeline.

Table 5.1 summarizes the key results from the HLS synthesis report, and we make the following observations:

```
1  while (pc != 0) {
2    #pragma HLS pipeline
3    ins = imem[IADDR (pc)];
4    pc = pc + 4;
5    op = ins >> 26;
6    switch (op) {
7      case R:
8          funct = ins & 0x3f;
9          shamt = (ins >> 6) & 0x1f;
10         rd = (ins >> 11) & 0x1f;
11         rt = (ins >> 16) & 0x1f;
12         rs = (ins >> 21) & 0x1f;
13         switch (funct) {
14             case ADDU:
15                 reg[rd] = reg[rs] + reg[rt];
16                 break;
17             case SUBU:
18                 reg[rd] = reg[rs] - reg[rt];
19                 break;
20             /* additional funct omitted */
21         }
22         break;
23     case J:
24         tgtadr = ins & 0x3ffffff;
25         pc = tgtadr << 2;
26         break;
27     case JAL:
28         tgtadr = ins & 0x3ffffff;
29         reg[31] = pc;
30         pc = tgtadr << 2;
31         break;
32     default:
33         address = ins & 0xffff;
34         rt = (ins >> 16) & 0x1f;
35         rs = (ins >> 21) & 0x1f;
36         switch (op) {
37             case ADDIU:
38                 reg[rt] = reg[rs] + address;
39                 break;
40             case LUI:
41                 reg[rt] = address << 16;
42                 break;
43             /* additional op omitted */
44             default:
45             pc = 0; /* error */
46             break;
47         }
48       break;
49     }
50 }
```

Figure 5.1: **Code snippet of the MIPS benchmark in the CHStone benchmark suite.**

1. **Sub-optimal initiation interval**. Although we target a fully pipelined implementation, the HLS synthesized design only achieved an initiation interval of four due to the loop-carried dependency on the program counter `pc`. This dependency is typically resolved through speculative execution in manual processor design. However, the current commercial HLS tools do not support such speculation techniques.

2. **Inefficient datapath implementation**. Different instructions specified using case statements are parsed and defined in different basic blocks of the CDFG. Current HLS tools have insufficient support for exploring the intricate tradeoff between reducing the number of functional units and increasing the complexity of multiplexer networks. Consequently, HLS tools in many cases tend to allocate duplicated resources even though the operations can share the same resource. For instance, the HLS tool allocates two 32-bit physical multipliers for the signed and unsigned multiplication operations, respectively, while only one multiplier is needed in a resource-efficient implementation.

3. **Complex FSM control logic**. The HLS tool generates a 7-state FSM to control the pipeline execution, leading to additional resource usage for handling state transitions. Compared to an optimized processor implementation without explicit control FSM, the FSM logic in the HLS generated design incurs additional resource usage and increases cycle time.

ASSIST aims to overcome the aforementioned drawbacks by proposing domain-specific HLS techniques targeting ISA-defined programmable architectures. Specifically, ASSIST (1) guarantees a fully-pipelined datapath and resolves hazards through forwarding, stalling, and speculation, (2) facilitates

resource sharing using instruction composition based on micro-ops, and (3) avoids centralized FSM by synthesizing pipeline control logic distributed across pipeline stages.

5.2 Assumptions and Limitations

Realistic processors incorporate sophisticated micro-architectural mechanisms to maximize the performance. While ASSIST attempts to include the beneficial mechanisms for synthesizing high-quality designs, the current version of ASSIST has the following assumptions and limitations:

1. **Single-issue pipeline**. The pipeline issues at most one instruction per clock cycle. This is not a fundamental limit in ASSIST, and we can extend the ASSIST framework to support superscalar execution by adding multiple execution units and a dispatch unit at the instruction issue stage.
2. **Not-taken branch prediction**. We assume a simple branch prediction scheme where conditional branches are always predicted to be not taken. More complex branch prediction schemes can be incorporated by creating dedicated branch prediction unit.
3. **No floating-point support**. The current framework only supports fixed-point data format and operators. Extensions to floating-point arithmetic would require definitions of floating-point registers and floating-point arithmetic units.
4. **Latency-insensitive interface to cache/memory**. The pipeline assumes a variable-latency interface to memory. The interface can be further sim-

plified if the instruction and data memories present a fixed-latency interface. In addition, memory input delays can be specified within the ASSIST framework to model the behavior of the memory interface.

5. **RISC instructions**. ASSIST currently supports RISC-like ISA specifications, which require no variable-cycle instructions, no complex memory addressing mode, etc. This design decision is made primarily to simplify the synthesis flow, which is not a fundamental limitation and can be potentially extended by incorporating additional logic to handle more complex instructions.
6. **In-order pipeline execution**. Instructions are executed in their original order of the program. Support for out-of-order execution requires extensive modifications to the datapath and control logic, including adding the dispatch and issue logic and logic for the instruction retire stage.

5.3 ASSIST Techniques

Figure 5.2 provides a high-level view of the ASSIST synthesis flow, which consists of two main components. The architectural synthesis engine takes an instruction set specification and a proposed pipeline schedule (i.e., the number of pipeline stages and the locations of the pipeline registers), and synthesizes the corresponding datapath and control logic of the processor in the form of Chisel HDL. The generated Chisel HDL is then simulated with a set of benchmarks to evaluate the cycle-level performance, and implemented using a user-supplied technology to obtain the implementation results. The second component of ASSIST is an autotuning engine based on OpenTuner [4], which uses the cy-

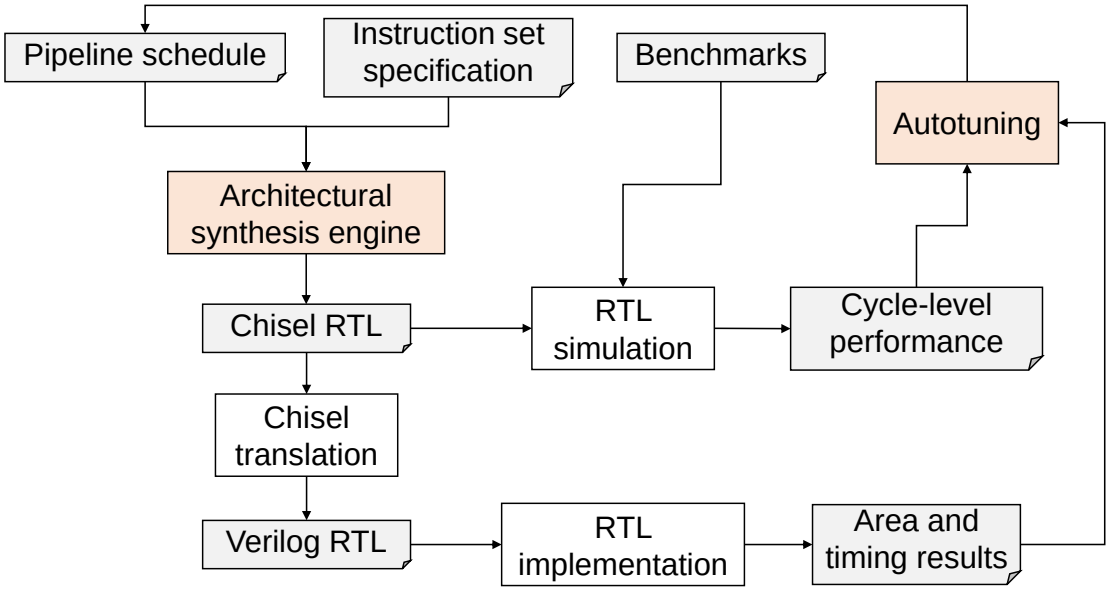


Figure 5.2: **The overall flow of ASSIST.**

cle count and implementation results as feedback, and automatically optimizes microarchitecture-level parameters such as the pipeline schedule.

5.3.1 Instruction Set Specification

With ASSIST, the designer specifies each instruction in the instruction set by describing the instruction name, encoding and the execution semantics. In particular, the execution semantics of each instruction is defined by composing a subset of pre-defined micro-ops. Table 5.2 lists the pre-defined micro-ops in ASSIST. Figure 5.3 shows the specifications of three instructions in the RISC-V 32I ISA: `add`, `beq`, and `lb`, and a custom `simd_add` operation. The instruction specifications start with `s.create_inst`, which registers the declared instruction with its assembly name and its encoding. The users then use predefined functions such as arithmetic operations (e.g., `s.add`), comparisons (e.g.,

Table 5.2: **List of micro-ops used in the ISA specification.**

Function	Description
create_inst	Create a new instruction with assembly name and encoding
inc_pc	Increment PC to the next word
update_pc	Update the value of PC with address
update_pc_with_pred	Update PC if pred is true; otherwise do inc_pc
add	Addition
sub	Substraction
xor	Bitwise Exclusive OR
slt	Set true if less than
sltu	Set true if less than (unsigned operands)
or	Bitwise OR
and	Bitwise AND
sll	Logical left shift
srl	Logical right shift
sra	Arithmetic right shift
bit_range	Select bit slice of certain range
bit_concat	Concatenate two bit slices
select	Select true or false branch based on condition
compare_eq	True if operands are equal
compare_ne	True if operands are not equal
compare_lt_signed	True if first operand less than second (signed)
compare_lt_unsigned	True if first operand less than second (unsigned)
compare_ge_signed	True if first operand greater than or equal to second (signed)
compare_ge_unsigned	True if first operand greater than or equal to second (unsigned)
compare_gt_signed	True if first operand greater second (signed)
compare_gt_unsigned	True if first operand greater second (unsigned)
mem_read	Read certain bytes from memory
mem_write	Write certain bytes to memory
assign	Assign operand to another value

`s.compare_eq`), PC control (e.g., `s.update_pc_with_pred`), and memory operations (e.g., `s.mem_read`).

The use of pre-defined micro-ops simplifies the process of resource sharing in ASSIST. Specifically, since different dynamic instructions share the ALU in a time multiplexing fashion, the ALU-related micro-ops instantiated in different instructions can be naturally shared. Another benefit of the use of micro-ops is that it creates a clear datapath-control split. Compute type micro-ops such as `add` and `bit_range` are intuitively mapped to the datapath, while control type

```
def execute_add(s):
    s.create_inst('ADD', '0000000xxxxxxxxxx000xxxxx0110011')
    tmp = s.add(s.rs1, s.rs2)
    s.assign(tmp, s.rd)
    s.inc_pc()

def execute_beq(s):
    s.create_inst('BEQ', 'xxxxxxxxxxxxxxxxx000xxxxx1100011')
    tmp = s.compare_ne(s.rs1, s.rs2)
    s.update_pc_with_pred(tmp, s.pc, s.imm_b)

def execute_lb(s):
    s.create_inst('LB', 'xxxxxxxxxxxxxxxxx000xxxxx0000011')
    addr = s.add(s.rs1, s.imm_i)
    s.mem_read(addr, 1, s.rd, SIGNED)
    s.inc_pc()

def execute_simd_add(s):
    s.create_inst('SIMD_ADD', '0000001xxxxxxxxxx010xxxxx1101011')

    def kernel():
        rs1_l = s.bit_range(s.rs1, 15, 0)
        rs1_h = s.bit_range(s.rs1, 31, 16)
        rs2_l = s.bit_range(s.rs2, 15, 0)
        rs2_h = s.bit_range(s.rs2, 31, 16)
        sum_l = s.add(rs1_l, rs2_l)
        sum_h = s.add(rs1_h, rs2_h)
        sum_val = s.bit_concat(sum_h, sum_l)
        return sum_val

    sum_val = s.compute_kernel(kernel, s.rs1, s.rs2)
    s.assign(sum_val, s.rd)
    s.inc_pc()
```

Figure 5.3: **Examples of instruction definitions in ASSIST.**

micro-ops such as `update_pc` are assigned to be implemented in the control logic.

In addition to instruction semantics, the user is also responsible for specifying the register fields and the definitions for immediate constants. Figure 5.4 shows the register and immediate constant specifications for the RISC-V 32I ISA.

```
# define register access fields
s.rs1 = s.RegAddr('rs1', ['range', 19, 15])
s.rs2 = s.RegAddr('rs2', ['range', 24, 20])
s.rd  = s.RegAddr('rd', ['range', 11, 7])

# define immediate encodings
s.imm_i = s.Imm('imm_i', [['fill', 20, 'inst[31]'],
                ['range', 31, 20]])
s.imm_s = s.Imm('imm_s', [['fill', 20, 'inst[31]'],
                ['range', 31, 25], ['range', 11, 7]])
s.imm_u = s.Imm('imm_u', [['range', 31, 12], ['fill', 12, 0]])
s.imm_b = s.Imm('imm_b', [['fill', 19, 'inst[31]'],
                ['fill', 1, 'inst[31]'], ['fill', 1, 'inst[7]'],
                ['range', 30, 25], ['range', 11, 8], ['fill', 1, 0]])
s.imm_j = s.Imm('imm_j', [['fill', 11, 'inst[31]'],
                ['fill', 1, 'inst[31]'], ['range', 19, 12],
                ['fill', 1, 'inst[20]'], ['range', 30, 21],
                ['fill', 1, 0]])
s.imm_z = s.Imm('imm_z', [['fill', 27, 0], ['range', 19, 15]])
```

Figure 5.4: **Examples of register field and immediate constant definitions in ASSIST.**

5.3.2 Datapath Synthesis

Starting from the instruction set specification, ASSIST first synthesizes the basic components of a datapath separately. The synthesized datapath components are then connected with optional pipeline registers between them. ASSIST abstracts the datapath of a typical in-order processor pipeline into seven component groups, each of which is associated with a template that will be instantiated given a concrete instruction set specification during the synthesis step.

PC update and instruction fetch This datapath component calculates the next PC value based on the different modes of branch and jump instructions inferred from the instruction set specification, specifically from the `update_pc` functions. The PC value is then used to fetch the next instruction from the instruction memory.

Instruction decode The instruction decoder generates decoded register addresses and immediate fields from the instruction encoding based on the input instruction set specification.

Register read This module instantiates the register file, and reads out the register values based on the register source addresses from the decoder. In addition, the forwarding path leading to the register file is handled in this module by introducing a multiplexer after the register output, which selects either the data from the register or the forwarding path selected by the forwarding control logic.

Branch condition and target This group of modules compute the predication whether a branch is taken, and calculate the corresponding branch target based on user's instruction set specification.

ALU operand select The ALU operand select logic chooses the correct operand sources for each instruction. In addition, it handles the forwarding path from ALU output or writeback module to the inputs of the ALU with a multiplexer at the output of the operand select module.

ALU logic This is the ALU logic for executing the various arithmetic operations defined in the instruction set specification. The common micro-ops across different instructions are shared in the ALU.

Memory request and register writeback This group of modules handle memory requests and responses assuming a latency-insensitive interface. The writeback multiplexer selects the correct source for writing back to the register file.

These seven groups of datapath components constitute a graph with data dependencies between certain stages defined by the dataflow in the processor pipeline. A *pipeline schedule* in ASSIST is a list specifying the pipeline stages of each of the seven component groups. For example, a pipeline schedule of `[1111111]` is a pipeline with all components in stage one, which is effectively a combinational datapath. Another example of a pipeline schedule, `[1234567]`, is a fully pipelined structure, with each component group occupying a separate pipeline stage. The specific pipeline schedule for a given instruction set can be either specified by the user, or explored automatically with autotuning, which will be discussed in Section 5.3.5. Given a proposed pipeline schedule, ASSIST analyzes the use-define relation between the different datapath components, and inserts pipeline registers to relay data across pipeline boundaries.

Depending on the pipeline schedule, there are two possible sources (i.e., at the ALU output or after the writeback multiplexer) and two possible destinations (i.e., after the register file read or after the ALU operand select multiplexers) for the forwarding paths. These four individual forwarding paths are constructed whenever the corresponding source-destination pairs are in different pipeline stages. The corresponding control signals are automatically generated in the control logic.

5.3.3 Control Logic Generation

Given the datapath and the corresponding pipeline schedule, the pipeline control logic is responsible for maximizing the pipeline throughput while ensuring correctness. For data hazards, the control logic enables data forwarding whenever possible, and stalls the pipeline if necessary to ensure correct execution. For control hazards, the control logic issues pipeline squash signal when a branch is mis-predicted, and resumes the execution from the correct branch target.

Forwarding control logic For each of the four possible forwarding paths in the datapath, ASSIST generates a control signal that enables the forwarding path if the register writeback address matches that of the data consuming instruction(s).

Data hazard stall logic This logic stalls the pipeline by inserting bubbles into the pipeline when a data hazard cannot be solved by forwarding. Examples of scenarios where stalling is needed are a slow memory response preventing the register read operation to happen, or reading from a register location whose value has not been generated by the ALU. In those scenarios, the data hazard signal will be asserted, which stalls all the in-flight instructions before the data-producing instruction until the data hazard has been resolved.

Pipeline squash logic For branches that are mistakenly predicted, the pipeline squash logic is asserted to squash all instructions in the mis-predicted path.

5.3.4 Pipelining in ASSIST

Pipelining is the main optimization technique in ASSIST to optimize the QoR of the synthesized processors. Here we discuss the pipelining opportunities and the corresponding design space created by the design points with various pipeline schedules.

Pipelining within a datapath module Some of the datapath modules such as the ALU module may need to be pipelined to achieve the desired clock frequency. ASSIST supports pipelining within a datapath module by enabling retiming in modern synthesis tools. With cycle time targeting retiming in the synthesis tool, the locations of the registers within a combinational module are automatically optimized to achieve the optimal clock frequency [60]. ASSIST automatically inserts a number of registers guided by the given pipeline schedule to the target combinational module, and uses retiming to relocate the registers into the combinational module.

Pipelining between datapath modules Pipeline registers can also be inserted at the boundaries between the datapath modules. In this process, the group of datapath modules formulates a directed acyclic graph (DAG), where each datapath module (which may be pipelined internally) is a node in the DAG. The edges in the DAG represent data dependencies between the datapath modules that are inferred from the functionality of each module. For example, the instruction decode logic depends on the instruction fetch logic, thus an edge exists from `instruction fetch` to `instruction decode`. Similarly, an edge exists between `Memory request/register writeback` and `ALU`

`logic` since the memory address or the register writeback value needs to be computed by the ALU before the memory request or writeback operation.

ASSIST allows the user to manually specify the pipeline schedule. Alternatively, ASSIST also supports an autotuning-based approach to automatically search for the optimized pipeline schedule for part of or the entire datapath. For instance, the user can choose to fix the schedule of the pipeline front end, and use ASSIST to automatically explore the different pipeline schedules of the ALU logic that implements the compute portion of various simple and complex instructions.

5.3.5 Autotuning of Pipeline Schedule

In addition to allowing the designers to manually specify the pipeline schedule, ASSIST supports automatic exploration of optimized pipeline schedules. In the traditional HLS tools, scheduling is typically performed using crude static timing estimates, where the delay of the datapath operators are pre-characterized and the datapath delay is derived by summing up the operation delays along the critical path. In ASSIST, we leverage the accurate timing and area information from downstream tools and use an autotuning based approach to search through the design space of various pipeline schedules. This choice is due to (1) the synthesized processor pipeline contains complex control logic that complicates static timing analysis, (2) multiple supported technology targets and generations make timing pre-characterization expensive and inflexible for new technology targets, and (3) many important QoR metrics such as execution time and area that are significantly impacted by scheduling cannot be accurately de-

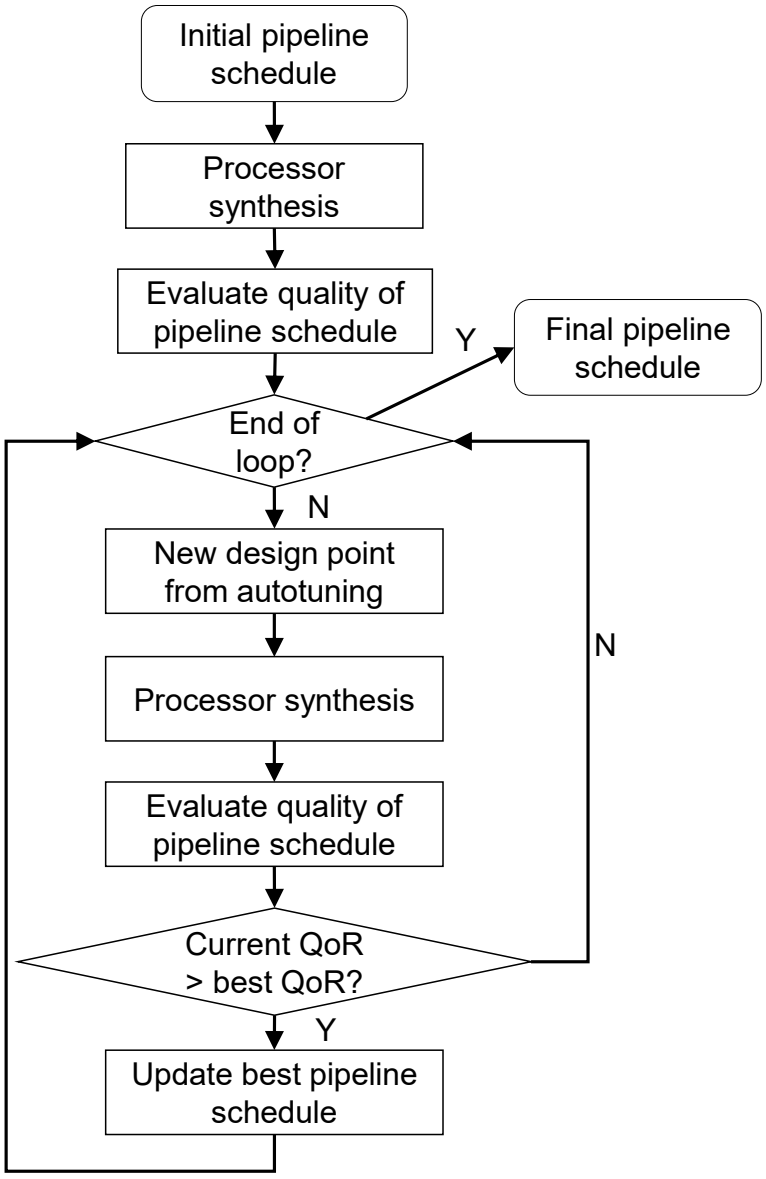


Figure 5.5: **High-level flow chart of the optimization steps in ASSIST.**

termined statically.

ASSIST utilizes the autotuning framework OpenTuner [4] to automatically search through the design space of pipeline schedules. Figure 5.5 illustrates the autotuning process in ASSIST. OpenTuner is an iterative meta-heuristic based autotuning framework, which maintains an ensemble of search techniques such

as simulated annealing and differential evolution. At each iteration, OpenTuner uses the multi-armed bandit (MAB) algorithm to select one of the search techniques. The selected search technique will be used to propose a new design point in the subspace for exploration.

Specifically, OpenTuner treats each search technique as an arm in the MAB formulation, and measures its reward using the area under curve mechanism [39]. OpenTuner balances exploitation (choosing the best-known arm to obtain the highest expected reward), and exploration (selecting an infrequently used arm to gain more information about its reward distribution) during the search process. This is achieved by ranking each arm with a score that reflects the arm's average reward and the frequency that the arm has been chosen so far. At each iteration, the arm (i.e., search technique) with the highest score will be selected to propose a new design point.

ASSIST supports three optimization modes. In the single-objective mode, the single target (e.g., cycle time) of each proposed design point is fed into OpenTuner in order to predict the next design point to explore. In the multi-objective mode, we use a heuristically-determined weighted sum of the optimization objectives to measure the quality of a design point. Finally, ASSIST also supports a constrained optimization target, where design points that fail the user-specified constraints are immediately discarded during the search process. The design points that satisfy user's constraints are subsequently used in the autotuning process.

While ASSIST does not directly explore the different possible memory architectures, the memory interface is exposed as input and output ports in ASSIST. During the autotuning process, the user can specify various input/output tim-

ing constraints to model the impact of the memory interface on the synthesized pipeline. These timing constraints are taken into account during the autotuning process such that the synthesized processor pipelines are optimized for the specific set of timing constraints.

5.4 Experimental Results

In this section, we present synthesis and implementation results of ASSIST. We begin by showing the design space of the processor microarchitecture in ASSIST using the RISC-V 32I ISA as an example. We then study the effectiveness of the autotuning technique using cycle time constrained performance optimization, where we show that the autotuning algorithm can find an optimized set of design points with a small number of test runs. Finally, we discuss results on using ASSIST to synthesize processors with custom instruction extensions and domain-specific processors such as a cryptographic processor.

5.4.1 Base Processor Design Space Exploration

We use the ASSIST synthesis flow to exhaustively generate the processor implementations of all possible schedules targeting the RISC-V 32I ISA. For the single-issue, in-order processor structures without pipelining within a datapath module, ASSIST groups the datapath components into seven modules. This leads to a total of 64 design points that are of different cross-component pipeline schedules, which are combinatorially enumerated by deciding whether to insert pipeline registers between each of the seven datapath component modules.

Table 5.3: **Top three designs for cycle time, area, and runtime targeting Virtex-7 FPGA** — We show the best designs in the design space for different targets. `Top-3`: the top-3 designs for the corresponding target; `Base-1`: manual one-stage design; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `CT`: final clock period in nanosecond after implementation; `LUT`: number of lookup tables used; `FF`: number of flip-flops used; `RT`: total runtime in millisecond over seven kernels.

Target	Schedule	CT	LUT	FF	RT
	Base-1	6.61	4892	2350	4.54
	Base-2	6.62	4571	2460	5.20
	Base-5	5.31	4701	2794	4.84
Cycle time	[1223444]	*4.84*	4885	2891	5.32
	[1123455]	*4.86*	4967	3102	6.69
	[1112333]	*4.86*	5013	2747	4.86
LUT	[1112334]	4.95	*4295*	2915	5.16
	[1111111]	6.50	*4300*	2350	4.47
	[1234456]	5.18	*4304*	3198	6.42
FF	[1111111]	6.50	4300	*2350*	4.47
	[1222222]	7.09	4811	*2481*	5.57
	[1122222]	6.89	4818	*2506*	5.41
Execution time	[1112222]	5.40	4850	2539	*4.24*
	[1112223]	5.25	4465	2707	*4.27*
	[1111111]	6.50	4300	2350	*4.47*

Each of the 64 design points are fully populated with forwarding paths whenever possible to maximize cycle-level performance.

We study the QoRs of the 64 design points under various FPGA and ASIC technology targets. For FPGA, we target a Xilinx Virtex-7 FPGA implemented with Vivado version 2017.1. For ASIC, we target 90nm and 32nm standard cell libraries, both implemented with Synopsys Design Compiler and IC Compiler II version 2016.12. To measure the total execution time of each design, we use the cycle count obtained from seven representative kernels and the corresponding cycle time to calculate the total execution time for finishing these benchmark

Table 5.4: **Top three designs for cycle time, area, and runtime targeting a 90nm ASIC technology library** — We show the best designs in the design space for different targets. `Top-3`: the top-3 designs for the corresponding target; `Base-1`: manual one-stage design; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `CT`: final clock period in nanosecond after implementation; `Area`: final design area in μm^2; `RT`: total runtime in millisecond over seven kernels.

Target	Schedule	CT	Area	RT
	Base-1	1.19	250629	0.818
	Base-2	0.57	243207	0.448
	Base-5	0.55	231026	0.501
Cycle time	[1234566]	*0.53*	288279	0.782
	[1223345]	*0.54*	245590	0.616
	[1223334]	*0.55*	242746	0.502
Area	[1111112]	0.58	*224577*	0.399
	[1223444]	0.61	*225291*	0.670
	[1222223]	0.60	*227934*	0.471
Execution time	[1111112]	0.58	224577	*0.399*
	[1111123]	0.58	239328	*0.416*
	[1111222]	0.62	243053	*0.426*

kernels. Tables 5.3 to 5.5 shows the top-3 designs in the design space for cycle time, area and execution time targeting the different technologies, while Figures 5.6 to 5.8 shows the design points in the area versus cycle time and area versus execution time curves. Table 5.6 further details all the five-stage variants from ASSIST targeting a 90nm ASIC technology library. We observe that there is significant variation in QoR even with fixed number of pipeline stages, motivating the need for automatic design space exploration of the pipeline schedule.

We compare the designs generated from ASSIST against three manually optimized designs from the Sodor Processor Collection [19] with 1, 3 and 5 pipeline stages. From Tables 5.3 to 5.5, we observe that across different technologies, the top-3 designs in the design space consistently outperforms the baseline designs for different targets. This shows that the designs synthesized from ASSIST

Table 5.5: **Top three designs for cycle time, area, and runtime targeting a 32nm ASIC technology library** — We show the best designs in the design space for different targets. `Top-3`: the top-3 designs for the corresponding target; `Base-1`: manual one-stage design; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `CT`: final clock period in nanosecond after implementation; `Area`: final design area in μm^2; `RT`: total runtime in millisecond over seven kernels.

Target	Schedule	CT	Area	RT
	Base-1	0.91	47908	0.625
	Base-2	0.34	52202	0.267
	Base-5	0.32	51706	0.292
Cycle time	[1112233]	*0.33*	54530	0.330
	[1223344]	*0.33*	56320	0.362
	[1223334]	*0.33*	53956	0.301
Area	[1111111]	0.90	*48964*	0.618
	[1222222]	0.36	*51692*	0.283
	[1122222]	0.38	*51874*	0.298
Execution time	[1111222]	0.36	53283	*0.247*
	[1111122]	0.38	53587	*0.261*
	[1112222]	0.34	52281	*0.267*

Table 5.6: **The five-stage pipeline variants from ASSIST targeting a 90nm ASIC technology library** — Execution time is measured as the runtime for finishing seven kernels.

	Cycle time (ns)	Area (μm^2)	Execution time (ms)
ASSIST five-stage designs	0.54	245590	0.62
	0.55	256126	0.56
	0.55	250426	0.50
	0.56	269950	0.67
	0.56	281748	0.77
	0.57	268992	0.68
	0.57	263528	0.52
	0.57	262058	0.65
	0.58	257719	0.66
	0.59	260898	0.67
	0.62	261721	0.68
	0.62	261181	0.82
	0.64	243676	0.67
	0.64	264084	0.67
	0.66	241202	0.91
Manual five-stage	0.55	231026	0.50

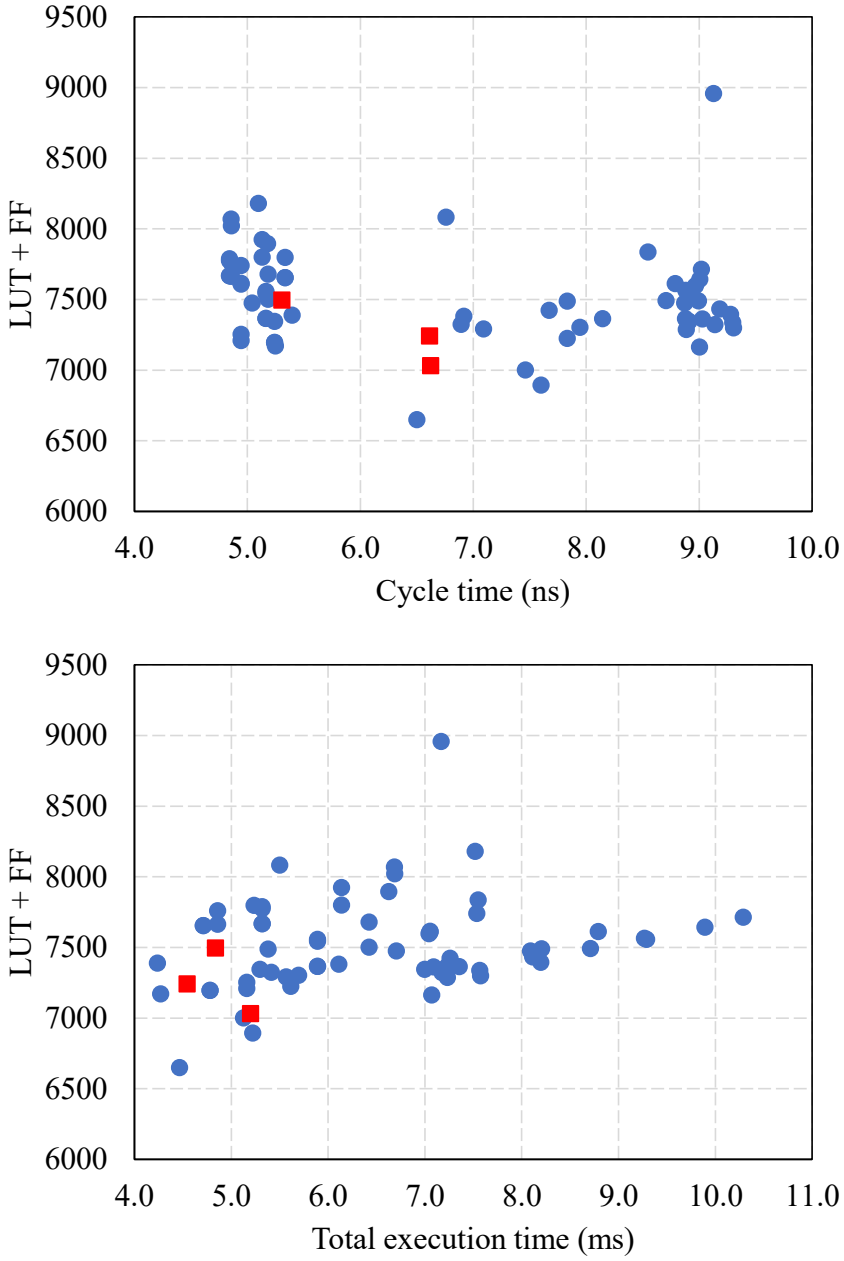


Figure 5.6: **Design space of synthesized pipeline processors targeting the Virtex-7 FPGA —** blue dots are designs automatically generated from our flow with one to seven pipeline stages. Red dots are the three manually optimized designs from [19] with one, two and five stages.

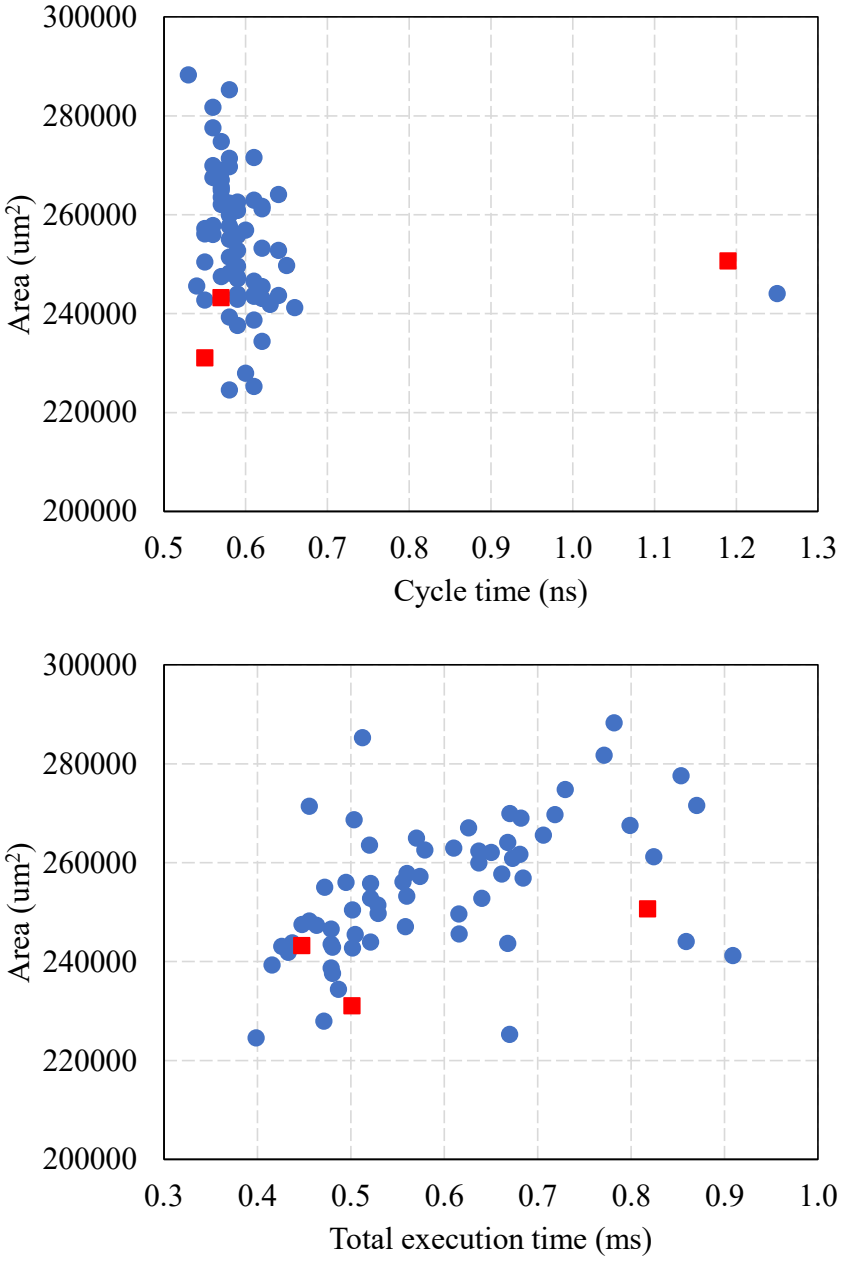


Figure 5.7: **Design space of synthesized pipeline processors targeting a 90nm ASIC technology library —** blue dots are designs automatically generated from our flow with one to seven pipeline stages. Red dots are the three manually optimized designs from [19] with one, two and five stages.

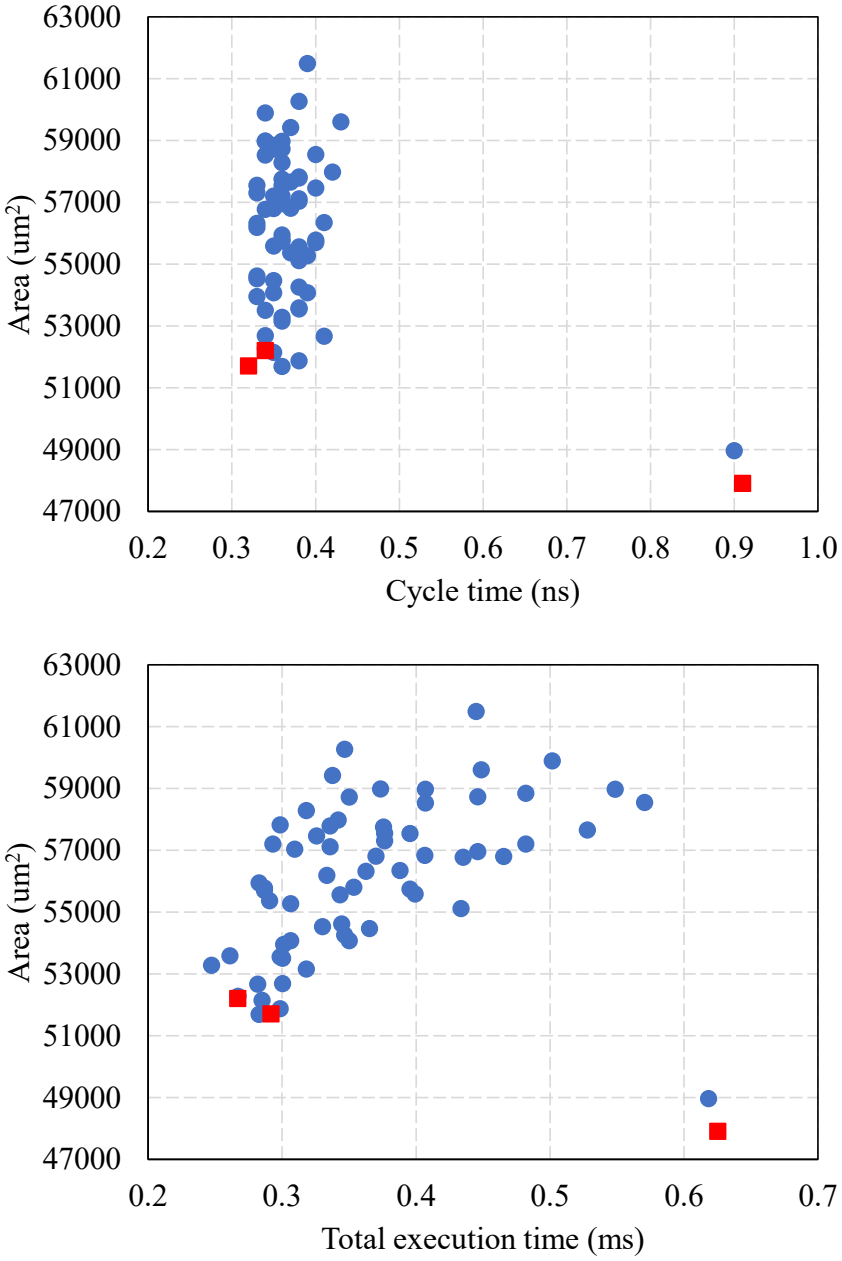


Figure 5.8: **Design space of synthesized pipeline processors targeting a 32nm ASIC technology library —** blue dots are designs automatically generated from our flow with one to seven pipeline stages. Red dots are the three manually optimized designs from [19] with one, two and five stages.

Table 5.7: **Optimization results using the autotuning algorithm targeting Virtex-7 FPGA** — We optimize the runtime under clock time constraint. For each clock time constraint, we run the autotuning flow for 100 times, each with a budget of 16 implementation runs, and record the best-found result and its occurrence. `T`: clock time constraint during implementation; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `Occ.`: occurrence of the design found during the 100 iterations; `CT`: final clock period in nanosecond after implementation; `RT`: total runtime in millisecond over seven kernels; `LUT`: number of lookup tables used; `FF`: number of flip-flops used.

	Schedule	Occ.	CT	RT	LUT	FF
	Base-2		6.62	5.202	4571	2460
	Base-5		5.31	4.838	4701	2794
	[1112222]	41	5.40	4.237	4850	2539
T=7.0	[1111111]	17	6.50	4.466	4300	2350
	[1112223]	11	5.25	4.271	4465	2707
	[1112233]	24	4.86	4.859	4915	2750
T=5.0	[1112333]	14	4.86	4.859	5013	2747
	[1112334]	11	4.95	5.160	4295	2915

manual designs.

5.4.2 Cycle Time Constrained Performance Optimization

The autotuning framework in ASSIST can effectively navigate through the design space by using intelligent search algorithms in OpenTuner [4]. We study the effectiveness of the autotuning technique in ASSIST targeting cycle time constrained performance optimization. Tables 5.7 to 5.9 show the results for different technologies under two different cycle time constraints. We measure the performance as the total execution time for finishing seven representative kernels. To minimize the effect of randomness during the autotuning process, we repeat the experiments for 100 times for each optimization task. For each experiment, we allocate a budget of exploring 16 design points for the auto-

Table 5.8: **Optimization results using the autotuning algorithm targeting a 90nm ASIC technology library** — We optimize the runtime under clock time constraint. For each clock time constraint, we run the autotuning flow for 100 times, each with a budget of 16 implementation runs, and record the best-found result and its occurrence. `T`: clock time constraint during implementation; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `Occ.`: occurrence of the design found during the 100 iterations; `CT`: final clock period in nanosecond after implementation; `RT`: total runtime in millisecond over seven kernels; `Area`: final design area in μm^2.

	Schedule	Occ.	CT	RT	Area
	Base-2		0.57	0.448	243207
	Base-5		0.55	0.501	231026
T=0.60	[1111112]	37	0.58	0.399	224577
	[1222333]	19	0.57	0.448	247493
	[1122222]	9	0.58	0.455	271427
T=0.56	[1233444]	24	0.56	0.495	256040
	[1233345]	17	0.55	0.502	250426
	[1223334]	16	0.55	0.502	242746

tuning algorithm. The 100 experiment runs are independently conducted. The `Occurrence` column in the tables represents the number of occurrences that the specific design point was found during the autotuning process across 100 runs. We observe that the autotuning algorithm is able to consistently obtain design points with superior performance than the baseline designs under various cycle time constraints. This shows that the autotuning algorithm is effective in search through the complex design space.

5.4.3 Integration of Instruction and Data Memories

While ASSIST does not directly synthesize the instruction and data memories, we design ASSIST such that the generated processor pipelines can be easily integrated with various types of memories and caches. We present a case study of

Table 5.9: **Optimization results using the autotuning algorithm targeting a 32nm ASIC technology library** — We optimize the runtime under clock time constraint. For each clock time constraint, we run the autotuning flow for 100 times, each with a budget of 16 implementation runs, and record the best-found result and its occurrence. `T`: clock time constraint during implementation; `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `Occ.`: occurrence of the design found during the 100 iterations; `CT`: final clock period in nanosecond after implementation; `RT`: total runtime in millisecond over seven kernels; `Area`: final design area in μm^2.

	Schedule	Occ.	CT	RT	Area
	Base-2		0.34	0.267	52201.7
	Base-5		0.32	0.292	51705.9
	[1111222]	39	0.36	0.247	53282.8
T=0.40	[1111122]	13	0.38	0.261	53586.5
	[1112222]	10	0.34	0.267	52281.0
	[1223334]	35	0.33	0.301	53955.5
T=0.33	[1112233]	15	0.33	0.330	54530.4
	[1234445]	9	0.33	0.333	56192.8

integrating ASSIST pipelines with scratchpad memories as instruction and data memories on Virtex-7 FPGAs. Although it is possible to integrate complete instruction and data caches to ASSIST, here we focus on demonstrating the adaptability of the autotuning algorithm to external memory interfaces. Scratchpad memory is a relatively simple memory structure that facilitates QoR analysis. In addition, asynchronous read operations can be easily supported in scratchpad memories, which is required by the current ASSIST synthesis flow. We design the scratchpad memories as RTL templates, which are incorporated to the processor pipelines during the optimization process.

Table 5.10 shows the QoR of the top-3 designs targeting cycle time minimization. We design the instruction scratchpad template to support asynchronous read operations. Such asynchronous read, synchronous write memories are mapped to distributed RAMs on the FPGA. The data memories in this exper-

Table 5.10: **Optimization results using the autotuning algorithm for designs with scratchpad memories targeting Virtex-7 FPGA** — We optimize the designs for cycle time, where the autotuning flow has a budget of 16 implementation runs. `Base-2`: manual two-stage design; `Base-5`: manual five-stage design; `CT`: achieved clock period in nanosecond after implementation; `RT`: total runtime in millisecond over seven kernels; `LUT`: number of lookup tables used; `FF`: number of flip-flops used; `BRAM`: number of block RAMs used.

Schedule	CT	RT	LUT	FF	BRAM
Base-2	6.90	5.422	4412	2462	1
Base-5	6.41	5.840	4798	2800	1
[1223333]	6.39	5.123	5106	2685	1
[1123333]	6.39	5.127	5283	2700	1
[1234444]	6.42	4.664	5114	2542	1

iment use synchronous read, and are mapped to block RAMs on the FPGA. The instruction and data scratchpads are set to 64 words and 128 words for this specific experiment, respectively.

We observe that ASSIST generated designs achieve better cycle time than the manually-optimized designs at the cost of small increase in resource usage. This is primarily due to the fact that ASSIST can effectively explore the design space of different pipeline schedules through autotuning. Since each autotuning iteration implements a design that combines the processor pipeline and the scratchpad memories, the impact of the memories on the QoR is accurately considered during the optimization process. In Table 5.10, the design with the minimal cycle time from ASSIST only uses a three-stage pipeline. We observe from the timing report that the ASSIST generated design enables the EDA implementation tool to aggressively optimize the combinational logic in the pipeline backend, thus compensating for the fewer pipeline stages and matching the cycle time of the baseline design.

```
1 for (i = 1, rcon = 1; i < 14; ++i) {
2     aes_subBytes(buf);
3     aes_shiftRows(buf);
4     aes_mixColumns(buf);
5     if ( i & 1 )
6         aes_addRoundKey( buf, &ctx->key[16]);
7     else {
8         aes_expandEncKey(ctx->key, &rcon);
9         aes_addRoundKey(buf, ctx->key);
10    }
11 }
```

Figure 5.9: **The main loop of the AES encryption algorithm.**

5.4.4 Custom Instruction Extension

One application of ASSIST is to synthesize processors with custom instruction extensions. This technique is especially useful for compute kernels that are not efficiently executed on the base processor, such as bit-level operations.

The Advanced Encryption Standard (AES) is a widely-used encryption technique for electronic data, which is composed of four major stages that are interleaved and iteratively executed as shown in Figure 5.9. Among the four major stages, the `mixColumns` step is the most time consuming, which constitutes around 50% of the total runtime for AES encryption. The `mixColumns` function is specified is Figure 5.12, where each iteration of the loop operates on four bytes by mixing them using XOR, logic AND, and shift operations.

We observe that each iteration of the `mixColumns` loop operates on a single 32-bit word, which can be potentially extracted as a single instruction as illustrated in Figure 5.11. In Figure 5.11, the entire loop body is implemented with one assembly instruction called `mix_column`.

The advantage of incorporating the entire `mixColumns` loop body instruc-

```
1 uint8_t rj_xtime(uint8_t x) {
2     return (x & 0x80) ? ((x << 1) ^ 0x1b) : (x << 1);
3 }
4
5 void aes_mixColumns(uint8_t *buf) {
6     register uint8_t i, a, b, c, d, e;
7     mix : for (i = 0; i < 16; i += 4) {
8         a = buf[i];
9         b = buf[i + 1];
10        c = buf[i + 2];
11        d = buf[i + 3];
12        e = a ^ b ^ c ^ d;
13        buf[i] ^= e ^ rj_xtime(a^b);
14        buf[i+1] ^= e ^ rj_xtime(b^c);
15        buf[i+2] ^= e ^ rj_xtime(c^d);
16        buf[i+3] ^= e ^ rj_xtime(d^a);
17 }
```

Figure 5.10: **Code snippet of the original `mixColumns` function in AES.**

```
1 void aes_mixColumns(unsigned int *buf) {
2     register uint8_t i;
3     for (i = 0; i < 4; i++) {
4         asm volatile (
5             "mix_column   %[z], %[x], %[y]"
6             : [z] "=r" (buf[i])
7             : [x] "r" (buf[i]), [y] "r" (buf[i]));
8 }}
```

Figure 5.11: **An alternative implementation of the `mixColumns` function using custom instruction `mix_column`.**

tion is the reduction of instruction count. The original loop body of the `mixColumns` has a total number of 50 assembly instructions, which is reduced to one single instruction by implementing the `mix_column` instruction.

In our flow, the designer only needs to define the `execute_mix_column` as in Figure 5.12, which is structurally very similar to the original C implementation of the `mixColumns` loop body. The ALU functionality and the corresponding decoding logic are automatically synthesized from our synthesis technique.

In Table 5.11, we compare the performance and resource usage of the AES

Table 5.11: **Performance and resource usage comparison between the the base processor and the custom processor with `mix_column` instruction extension targeting Xilinx Vertex 7 device —** `Cycle time` = cycle time in `ns`; `LUT` = number of lookup tables; `FF` = number of flip flops; `# Cycle` = number of cycles to complete AES encoding task; `Execution time` = execution time in microseconds.

	AES-original	AES-custom
Cycle time (ns)	4.78	4.99 (+4.2%)
LUT	4760	5308 (+11.5%)
FF	2912	2913 (+0.03%)
# Cycle	11248	5759 (-48.8%)
Execution time (μs)	53.8	28.7 (-46.6%)

```
def execute_mix_column(s):
    s.create_inst('MIX_COLUMN', '0000001xxxxxxxxxx001xxxxx1101011)

    def kernel():
        a = s.bit_range(s.rs1, 7, 0)
        b = s.bit_range(s.rs1, 15, 8)
        c = s.bit_range(s.rs1, 23, 16)
        d = s.bit_range(s.rs1, 31, 24)
        e = s.xor(s.xor(s.xor(a, b), c), d)
        ab = s.xor(a, b)
        bc = s.xor(b, c)
        cd = s.xor(c, d)
        da = s.xor(d, a)
        cond_ab = s.compare_gt_unsigned(
                    s.and(ab, s.const('0x80', 8)),
                    s.const('0', 1))
        cond_bc = s.compare_gt_unsigned(
                    s.and(bc, s.const('0x80', 8)),
                    s.const('0', 1))
        cond_cd = s.compare_gt_unsigned(
                    s.and(cd, s.const('0x80', 8)),
                    s.const('0', 1))
        cond_da = s.compare_gt_unsigned(
                    s.and(da, s.const('0x80', 8)),
                    s.const('0', 1))

        rj_ab_0 = s.bit_range(s.shift(ab, s.const('1', 1)), 7, 0)
        rj_bc_0 = s.bit_range(s.shift(bc, s.const('1', 1)), 7, 0)
        rj_cd_0 = s.bit_range(s.shift(cd, s.const('1', 1)), 7, 0)
        rj_da_0 = s.bit_range(s.shift(da, s.const('1', 1)), 7, 0)
        rj_ab_1 = s.xor(rj_ab_0, s.const('0x1b', 8))
        rj_bc_1 = s.xor(rj_bc_0, s.const('0x1b', 8))
        rj_cd_1 = s.xor(rj_cd_0, s.const('0x1b', 8))
        rj_da_1 = s.xor(rj_da_0, s.const('0x1b', 8))
        rj_ab = s.select(cond_ab, rj_ab_1, rj_ab_0)
        rj_bc = s.select(cond_bc, rj_bc_1, rj_bc_0)
        rj_cd = s.select(cond_cd, rj_cd_1, rj_cd_0)
        rj_da = s.select(cond_da, rj_da_1, rj_da_0)

        buf0 = s.xor(a, s.xor(e, rj_ab))
        buf1 = s.xor(b, s.xor(e, rj_bc))
        buf2 = s.xor(c, s.xor(e, rj_cd))
        buf3 = s.xor(d, s.xor(e, rj_da))

        res = s.bit_concat(buf3, s.bit_concat(
                            buf2, s.bit_concat(buf1, buf0)))

        return res

    res = s.compute_kernel(kernel, s.rs1, s.rs2)
    s.assign(res, s.rd)
    s.inc_pc()
```

Figure 5.12: **ISA definition of the custom `mix_column` instruction.**

encryption algorithm running on the RISC-V 32I base processor and the RISC-V 32I processor with the `mix_column` instruction extension targeting a Xilinx Virtex 7 FPGA. We observe that at the cost of 4.2% increase in cycle time and 11.5% increase in LUT usage, the processor with instruction extension reduces the total cycle count by 48.8%, leading to a 46.6% reduction in total execution time for each encryption task. To achieve such a performance improvement, the only tasks for the designer are specifying the `mix_column` instruction at the instruction level, changing the source code with the inline assembly instructions, and updating the RISC-V compiler with the encoding of the new `mix_column` instruction.

5.4.5 Cryptographic Instruction Set Extension

As a case study, we use the synthesis flow to design and generate a cryptographic ISA extension for the RISC-V 32I base ISA. We use the Cryptonite processor ISA [13] as our reference design. The Cryptonite processor uses a domain-specific ISA that can execute a number of standard cryptographic algorithms such as AES, DES, MD5 and SHA. We implement a 32-bit version of the cryptographic ISA extension with the following custom cryptographic instructions:

- `swap(x,y)`: take two half words, `x` and `y`, swap their relative positions. i.e., $f(x, y) = y \mid x$.
- `upper(x,y)`: mix the 4^{th} and 2^{nd} bytes of `x` and `y` in a certain way. i.e., $f(x, y) = x_3 \mid x_1 \mid y_3 \mid y_1$.

- `lower(x,y)`: mix the 3^{rd} and 1^{st} bytes of `x` and `y` in a certain way. i.e., $f(x, y) = x_2 \mid x_0 \mid y_2 \mid y_0$.
- `rbl(x,imm)`: a special shift operation. i.e., $f(x, imm) = x_h \ll imm \mid x_l \ll (imm + 8)$.
- `rbr(x,imm)`: another special shift operation. i.e., $f(x, imm) = x_h \ll imm \mid x_l \gg (imm + 8)$.
- `fold(x,y)`: reorder the higher half words and lower half words of x and y with XOR operations. i.e., $f(x, y) = x_h \oplus y_l \mid x_l \oplus y_h \oplus y_l$.
- `ifold(x,y)`: another way of reordering the higher high words and lower half words of x and y with XOR operations. i.e., $f(x, y) = x_l \oplus y_h \mid y_h \oplus y_l$.

Compared to the original paper [13] where the authors designed the entire cryptographic processor by hand, we were able to implement the instruction set as an extension to the RISC-V base ISA using the ASSIST synthesis flow. Specifically, implementing the seven cryptographic instructions only required around 100 lines of Python code. The synthesis flow then takes the user-specified instruction set, and synthesize a range of processors with different schedules. The autotuning backend automatically returns the optimized implementation for user-specified optimization objective.

Table 5.12 shows the timing, performance, and resource usage of the synthesized cryptographic processor when compared to the base RISC-V processor, where we use the autotuning algorithm to optimize for cycle time. We estimate the cycle count of the cryptographic processor by counting the number of dynamic instructions. We observe that the additional cost due to the instruction extensions leads to a moderate 8.4% increase in cycle time with negligible resource

Table 5.12: **Timing and resource usage comparison between the the base processor and the processor with cryptographic extensions targeting Xilinx Vertex 7 device —** `Cycle time` = cycle time in `ns`; `Cycle count` = estimated number of cycles to execute a 128-bit AES encryption task; `Execution time` = runtime for a 128-bit AES encryption task; `LUT` = number of lookup tables; `FF` = number of flip flops.

	RISC-V base	RISC-V + Crypto
Cycle time (ns)	4.78	5.18 (+8.4%)
Dynamic instruction count	9929	984 (-90.1%)
Execution time (μs)	47.4	5.1 (9.3X faster)
LUT	4760	5460 (+14.7%)
FF	2912	2686 (-7.8%)

overhead. The cryptographic processor executes a 128-bit AES encryption task 9.3X faster than the baseline processor without cryptographic extensions, showing the performance benefit of domain-specific instruction extensions that are automatically synthesized from the ASSIST flow.

5.5 Related Work

Existing efforts on programmable processor generation typically employ a template-based approach, where the designer instantiates from a set of predefined microarchitectural configurations. Representative examples include Xtensa [42] and FabScalar [18]. These techniques usually fix the pipeline schedule/structure which do not adapt well to different technology targets. These techniques usually have fixed pipeline schedules which do not necessarily adapt well to different technology targets. In addition, designers need to manually explore the design space to find an optimized design point satisfying the QoR target.

Another line of research aims at enabling processor synthesis from architectural description languages (ADLs). The idea is to design ADLs specifically for describing processor pipelines while providing a convenient interface for designers to specify the desired processor architecture. Examples in this category includes LISA [89], EXPRESSION [75], and T-spec/T-piper [77]. These synthesis approaches also require designers to manually specify the datapath and pipeline schedules, and usually do not provide automatic optimization techniques for finding optimized design points. Processor synthesis using Bluespec [6] follows a similar approach where designers describe the processor using guarded atomic action constructs. While this provides a productive environment for designers to specify their design, manual datapath construction is still required to synthesize a processor pipeline.

Mokhov, et al. propose techniques that synthesize processor instruction sets from high-level ISA specifications [76], where designers can quickly experiment with different ISAs using their tool. However, they focus on supporting ISA extensions, without providing techniques to automatically pipeline the synthesized datapath. Huang, et al. present a technique to generate back-end compilers together with synthesized processor pipelines [50]. Their approach requires a new compiler for each processor architecture because it relies on the compiler to statically resolve data and control hazards, which usually results in conservative designs with sub-optimal QoR. Ralf Dreesen proposes an automatic flow to generate simple processor pipelines from ISA descriptions [34]. However, the proposed technique lacks support for resource sharing between similar instructions, and uses precomputed component delay models for pipeline scheduling. The generated designs are approximately 3X larger than the manually optimized counterparts, and have relative low design frequency at 680 MHz using

a 65nm ASIC technology library.

5.6 Conclusions and Future Directions

This chapter presents ASSIST, a processor synthesis framework that automatically synthesizes single-issue, in-order, pipelined processors from an instruction set specification. ASSIST explores the design space of different pipeline schedules using an autotuning-based method. We demonstrate the QoR of ASSIST by exploring the design space of RISC-V 32I processors, and discuss the synthesis of a cryptographic instruction set extension to the base processor. Further directions include the extension to various micro-architectural features such as various branch prediction mechanisms, superscalar execution and the incorporation of various memory systems. Additional optimization targets such as energy efficiency can also be explored by integrating power analysis into the autotuning flow. In addition, machine learning based techniques can potentially be used for fast QoR estimation that alleviates the runtime overhead of the implementation tools.

BIBLIOGRAPHY

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2013.

[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL Combinational Benchmark Suite. *International Workshop on Logic & Synthesis (IWLS)*, Jun 2015.

[3] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(5):806–819, May 2016.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Aug 2014.

[5] Marnix Arnold and Henk Corporaal. Designing Domain-Specific Processors. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Apr 2001.

[6] Arvind, Rishiyur S Nikhil, Daniel L Rosenband, and Nirav Dave. High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2004.

[7] Krste Asanović and David A Patterson. Instruction Sets Should be Free: the Case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, Aug 2014.

[8] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma. Always-On Vision Processing Unit for Mobile Applications. *IEEE Micro*, 35(2):56–66, Mar 2015.

[9] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 3(3), Sep 2010.

[10] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 60413. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[11] Sanjeev P Bhavnani, Jagat Narula, and Partho P Sengupta. Mobile Technology and the Digitization of Healthcare. *European heart journal*, 37(18):1428–1438, Feb 2016.

[12] Shekhar Borkar and Andrew A Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[13] Rainer Buchty, Nevin Heintze, and Dino Oliva. Cryptonite–A Programmable Crypto Processor Architecture for High-Bandwidth Applications. *International Conference on Architecture of Computing Systems*, Mar 2004.

[14] Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2003.

[15] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2011.

[16] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart Cities in Europe. *Journal of urban technology*, 18(2):65–82, Apr 2011.

[17] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. *Design Automation Conf. (DAC)*, May 2013.

[18] Niket K Choudhary, Salil V Wadhavkar, Tanmay A Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H Dwiel, Sandeep Navada, Hashem H Najafabadi, and Eric Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[19] Christopher Celio. The Sodor Processor Collection. `https://github.com/ucb-bar/riscv-sodor`.

[20] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang.

High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Apr 2011.

[21] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-Rich Architectures: Opportunities and Progresses. *Design Automation Conf. (DAC)*, May 2014.

[22] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Apr 2011.

[23] Jason Cong and Kirill Minkovich. Optimality Study of Logic Synthesis for LUT-Based FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):230–239, Feb 2007.

[24] CPLEX. High-Performance Software for Mathematical Programming and Optimization. *IBM*, 2005.

[25] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing, 1999.

[26] Tomasz S Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P Singh, and Stephen D Brown. OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Jul 2012.

[27] Steve Dai, Gai Liu, and Zhiru Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[28] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. Enabling Adaptive Loop Pipelining in High-Level Synthesis. *Asilomar Conference on Signals, Systems, and Computers*, Oct 2017.

[29] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2014.

[30] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[31] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar 2018.

[32] Timothy A Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, Nov 2011.

[33] Giovanni De Micheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 10(1):63–73, Jan 1991.

[34] Ralf Dreesen. Generating Interlocked Instruction Pipelines from Specifications of Instruction Sets. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2012.

[35] Nicole B Ellison, Charles Steinfield, and Cliff Lampe. The Benefits of Facebook Friends: Social Capital and College Students' Use of Online Social Network Sites. *Journal of Computer-Mediated Communication*, 12(4):1143–1168, Jul 2007.

[36] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[37] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2012.

[38] Amir H Farrahi and Majid Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11), Nov 1994.

[39] Álvaro Fialho, Raymond Ros, Marc Schoenauer, and Michele Sebag. Comparison-Based Adaptive Strategy Selection with Bandits in Differential Evolution. *International Conference on Parallel Problem Solving from Nature*, Sep 2010.

[40] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, Aug 2004.

[41] Charles J Geyer. Practical Markov Chain Monte Carlo. *Statistical Science*, pages 473–483, 1992.

[42] Ricardo E Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, Mar 2000.

[43] Michael Gschwind, H Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell. *IEEE Micro*, Mar 2006.

[44] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[45] Jie Han and Michael Orshansky. Approximate Computing: An Emerging Paradigm for Energy-Efficient Design. *European Test Symposium (ETS)*, May 2013.

[46] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, May 2008.

[47] W Keith Hastings. Monte Carlo Sampling Methods using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[48] Jason Hsu. *Multiple Comparisons: Theory and Methods*. Chapman and Hall/CRC, 1996.

[49] Yu Hu, Victor Shih, Rupak Majumdar, and Lei He. FPGA Area Reduction by Multi-Output Function Based Sequential Resynthesis. *Design Automation Conf. (DAC)*, Jun 2008.

[50] I-J Huang and Alvin M Despain. High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers. *Design Automation Conf. (DAC)*, Jul 1992.

[51] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[52] William J Kaufmann and Larry L Smarr. *Supercomputing and the Transformation of Science*. WH Freeman & Co., 1992.

[53] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[54] W James Kent, Charles W Sugnet, Terrence S Furey, Krishna M Roskin, Tom H Pringle, Alan M Zahler, and David Haussler. The Human Genome Browser at UCSC. *Genome research*, 12(6):996–1006, Jun 2002.

[55] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. *Int'l Symp. on Microarchitecture (MICRO)*, pages 468–479, Dec 2013.

[56] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3):46–54, Mar 2011.

[57] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Int'l Symp. on Code Generation and Optimization (CGO)*, Mar 2004.

[58] Marco Lattuada and Fabrizio Ferrandi. Exploiting Outer Loops Vectorization in High Level Synthesis. *Architecture of Computing Systems (ARCS)*, pages 31–42, Mar 2015.

[59] Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Larry G Roberts, and Stephen Wolff. A Brief History of the Internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, Oct 2009.

[60] Charles E Leiserson and James B Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6(1-6):5–35, Jun 1991.

[61] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards Fully Autonomous Driving: Systems and Algorithms. *Intelligent Vehicles Symposium (IV)*, Jun 2011.

[62] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D Chamberlain. Deadlock Avoidance for Streaming Computations with Filtering. *Int'l Symp. on Parallelism in Algorithms and Architectures (SPAA)*, Jun 2010.

[63] Feng Liu, Soumyadeep Ghosh, Nick P Johnson, and David I August. CGPA: Coarse-Grained Pipelined Accelerators. *Design Automation Conf. (DAC)*, Jun 2014.

[64] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 36(11):1817–1830, Nov 2017.

[65] Gai Liu, Ye Tao, Mingxing Tan, and Zhiru Zhang. CASA: Correlation-Aware Speculative Adders. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Aug 2014.

[66] Gai Liu and Zhiru Zhang. A Reconfigurable Analog Substrate for Highly Efficient Maximum Flow Computation. *Design Automation Conf. (DAC)*, Jun 2015.

[67] Gai Liu and Zhiru Zhang. A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[68] Gai Liu and Zhiru Zhang. Statistically Certified Approximate Logic Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2017.

[69] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2013.

[70] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. Multi-Level Ap-

proximate Logic Synthesis under General Error Constraints. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2014.

[71] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[72] A Mishchenko and RK Brayton. Scalable Logic Synthesis using a Simple Circuit Structure. *International Workshop on Logic & Synthesis (IWLS)*, Jul 2006.

[73] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. *Design Automation Conf. (DAC)*, Jul 2006.

[74] Alan Mishchenko, Satrajit Chatterjee, and Robert K Brayton. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):240–253, Feb 2007.

[75] Prabhat Mishra, Aviral Shrivastava, and Nikil Dutt. Architecture Description Language (ADL)-Driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 11(3):626–658, Jun 2004.

[76] Andrey Mokhov, Alexei Iliasov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev, and Alexander Romanovsky. Synthesis of Processor Instruction Sets from High-Level ISA Specifications. *IEEE Transactions on Computers*, 63(6):1552–1566, Jun 2014.

[77] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. Automatic Pipelining from Transactional Datapath Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454, Mar 2011.

[78] Peichen Pan, Arvind K Karandikar, and CL Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 17(6):489–498, Jun 1998.

[79] Darin Petkov, Randolph Harr, and Saman Amarasinghe. Efficient Pipelining of Nested Loops: Unroll-and-Squash. *Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Apr 2001.

[80] Louis-Noël Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-Based Data Reuse Optimization for Configurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2013.

[81] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. *Int'l Symp. on Computer Architecture (ISCA)*, Oct 2014.

[82] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[83] J Ramanujam. Optimal Software Pipelining of Nested Loops. *Int'l Parallel Processing Symp. (IPPS)*, Apr 1994.

[84] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 1994.

[85] Louis B Rosenberg. The Use of Virtual Fixtures as Perceptual Overlays to Enhance Operator Performance in Remote Environments. Technical report, Stanford University, Sep 1992.

[86] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[87] Robert R Schaller. Moore's Law: Past, Present and Future. *IEEE spectrum*, 34(6):52–59, Jun 1997.

[88] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2013.

[89] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. Architecture Implementation using the Machine Description Language LISA. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, Jan 2002.

[90] Juliet Popper Shaffer. Multiple Hypothesis Testing. *Annual Review of Psychology*, 46(1):561–584, Feb 1995.

[91] Doochul Shin and Sandeep K Gupta. Approximate Logic Synthesis for Error Tolerant Applications. *Design, Automation, and Test in Europe (DATE)*, Mar 2010.

[92] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[93] Sanbao Su, Yi Wu, and Weikang Qian. Efficient Batch Statistical Error Estimation for Iterative Multi-Level Approximate Logic Synthesis. *Design Automation Conf. (DAC)*, Jun 2018.

[94] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[95] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2014.

[96] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2015.

[97] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2015.

[98] Don Tapscott. *Blueprint to the Digital Economy: Creating Wealth in the Era of E-Business*. McGraw-Hill, Inc., 1999.

[99] Y Tatsumi and HJ Mattausch. Fast Quadratic Increase of Multiport-Storage-Cell Area with Port Number. *Electronics Letters*, Dec 1999.

[100] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. AxNN: Energy-Efficient Neuromorphic Systems using Approximate Computing. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Aug 2014.

[101] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. *Design, Automation, and Test in Europe (DATE)*, Mar 2013.

[102] Andrew J Viterbi. The Evolution of Digital Wireless Technology from Space Exploration to Personal Communication Services. *IEEE Transactions on Vehicular Technology*, 43(3):638–644, Aug 1994.

[103] Ben James Winer, Donald R Brown, and Kenneth M Michels. *Statistical Principles in Experimental Design*. McGraw-Hill, 1971.

[104] Felix Winterstein, Samuel Bayliss, and George A Constantinides. High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS. *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2013.

[105] Yi Wu and Weikang Qian. An Efficient Method for Multi-Level Approximate Logic Synthesis under Error Rate Constraint. *Design Automation Conf. (DAC)*, Jun 2016.

[106] Yi Wu, Chuyu Shen, Yi Jia, and Weikang Qian. Approximate Logic Synthesis for FPGA by Wire Removal and Local Function Change. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, Jan 2017.

[107] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, 2017.

[108] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[109] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks*. Microelectronics Center of North Carolina (MCNC), 1991.

[110] Wenlong Yang, Lingli Wang, and Alan Mishchenko. Lazy Man's Logic Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2012.

[111] Zhiru Zhang and Bin Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2013.

[112] Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, and Zhiru Zhang. Improving High-Level Synthesis with Decoupled Data Structure Optimization. *Design Automation Conf. (DAC)*, Jun 2016.

[113] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2015.

[114] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[115] Qiuling Zhu, Ofer Shacham, Albert Meixner, Jason Rupert Redgrave, Daniel Frederic Finchelstein, David Patterson, Neeti Desai, Donald Stark, Edward T Chang, William R Mark, et al. Architecture for High Performance, Power Efficient, Programmable Image Processing, May 2018. US Patent 9,965,824.