Supervised Design-Space Exploration

Hung-Yi Liu

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Supervised Design-Space Exploration

Hung-Yi Liu

Low-cost Very Large Scale Integration (VLSI) electronics have revolutionized daily life and expanded the role of computation in science and engineering. Meanwhile, process-technology scaling has changed VLSI design to an exploration process that strives for the optimal balance among multiple objectives, such as power, performance, and area, i.e. multi-objective Pareto-set optimization. Besides, modern VLSI design has shifted to synthesis-centric methodologies in order to boost the design productivity, which leads to better design quality given limited time and resources. However, current decade-old synthesis-centric design methodologies suffer from: *(i)* long synthesis tool runtime, *(ii)* elusive optimal setting of many synthesis knobs, *(iii)* limitation to one design implementation per synthesis run, and *(iv)* limited capability of digesting only component-level designs as opposed to holistic system-wide synthesis. These challenges make Design Space Exploration (DSE) with synthesis tools a daunting task for both novice and experienced VLSI designers, thus stagnating the development of more powerful (i.e. more complex) computer systems.

To address these challenges, I propose Supervised Design-Space Exploration (SDSE), an abstraction layer between a designer and a synthesis tool, aiming to autonomously supervise synthesis jobs for DSE. For system-level exploration, SDSE can approximate a system Pareto set given limited information: only lightweight component characterization is required, yet the necessary component synthesis jobs are discovered on-the-fly in order to compose the system Pareto set. For component-level exploration, SDSE can approximate a component Pareto set by iteratively refining the approximation with promising knob settings, guided by synthesis-result estimation with machine-learning models. Combined, SDSE has been applied with the three major synthesis stages, namely high-level, logic, and physical synthesis, to the design of heterogeneous accelerator cores as well as high-performance processor cores. In particular, SDSE has been successfully integrated

into the IBM Synthesis Tuning System, yielding 20% better circuit performance than the original system on the design of a 22nm server processor that is currently in production.

Looking ahead, SDSE can be applied to other VLSI designs beyond the accelerator and the programmable cores. Moreover, SDSE opens several research avenues for: *(i)* new development and deployment platforms of synthesis tools, *(ii)* large-scale collaborative design engineering, and *(iii)* new computer-aided design approaches for new classes of systems beyond VLSI chips.

Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to express my sincere appreciation and gratitude toward my PhD advisor, Prof. Luca Carloni. During my PhD journey, Prof. Carloni believed in my talent and potential, inspired me with countless research ideas, supported me to resolve research and life difficulties, and acted as a role model for research and teaching excellence. I look forward to our continued collaboration in the years to come. Besides my advisor, I would like to thank all the other members who served in my thesis defense committee, including Prof. Steven Nowick, Prof. Kenneth Shepard, Prof. Mihalis Yannakakis, and Dr. Matthew Ziegler. In particular, special thanks to Prof. Steven Nowick, who was virtually my secondary advisor, providing me invaluable research feedbacks and writing me strong recommendation letters; special thanks to Prof. Kenneth Shepard, who provided practical comments from a designer's perspective and generously offered design tools as well as computing resource for me to perform experiments; special thanks to Prof. Mihalis Yannakakis, who pioneered the research in multi-objective optimization that led to my first PhD publication, a joint work with one of his prior PhD students; special thanks to Dr. Matthew Ziegler, who acknowledged my internship request, brainstormed and collaborated with me on exiting research projects, and complemented the committee as a researcher from an industrial research lab.

In addition, I would like to thank all my research colleagues, my mentors/managers from the industry, and the members/alumni of the System Level Design (SLD) group at Columbia University. Specifically, I enjoyed the collaboration with the following talented and inspiring researchers: Prof. Nicola Bombieri, Prof. Ilias Diakonikolas, Prof. Franco Fummi, PhD student Jihye Kwon, and Dr. Michele Petracca. Moreover, on sponsored internships I was privileged to be mentored by two top-notch research scientists, Robert Condon (Intel) and Dr. Matthew Ziegler (IBM), and be coached by several empathetic managers, Dr. Yatin Hoskote (Intel), Dr. Christy Tyberg (IBM), and Dr. Victor

Zyuban (IBM). Also, special thanks to Dr. Michael Theobald (D. E. Shaw Research), who offered me a timely Teaching Assistant position at Columbia and has provided influential career advices since then. Furthermore, I would like to thank the fellow PhD students and research scientists in the SLD group. In particular, special thanks to Emilio Cota and Dr. Marcin Szczodrak for their selfless contributions to the group's network and web infrastructure, which was very helpful when I served as a teaching assistant for multiple times. Also, special thanks to Dr. Giuseppe Di Guglielmo, who fearlessly tested, evaluated, and improved my prototype tools.

Needless to say, my family played a central role during my pursuit of the PhD. I am particularly grateful for the supports in any form extended by my parents as well as my parents in law. Without their supports, it would be impossible for me to see the light at the end of the tunnel. Besides, during this special journey I considered myself extremely blessed to have my first kid, Ryan, come to my life and explore the world with me. Since the very first day I was aware of this new life, he has been teaching me how to be a good son, a good father, and a good husband. As for my beloved wife, Melody, I truly do not know how to express my appreciation, gratitude, and apologies for her devotion, caring, and sacrifices. On the other hand, I truly thank Lord for giving my wife as my helper, who has been perfecting me with the Lord's word and exercising with me to become a vessel for the Lord's grace.

Finally, I would like to thank all the brothers and sisters in His body. All their prayers witness to how faithful our Lord is. I also very much thank Lord for giving me this PhD journey, during which I appreciated the fact that I am nothing but He is everything. Glory to Him.

To Lord, My Shepherd.

Chapter 1

Introduction

Design is an ubiquitous concept in every pursuit of knowledge, but has different connotations in different fields. Yet in general, the process of design corresponds to the exploration of a large (often infinite) space for creating complex objects. To facilitate the design process, a clear "taxonomy of design" can help analyze various design paradigms. Hence, I first review a recent design taxonomy and apply it to analyze common paradigms for designing complex computer systems. Then, I point out major challenges for designing *integrated electronic systems* and propose *Supervised Design-Space Exploration* for addressing these challenges.

1.1 What is Design?

Based on 33 proposals in existing literature, Ralph and Wand proposed in 2009 the following definition for design (as a noun): *a specification of an object, manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints* [150]. The definition, illustrated in Figure 1.1, captures the following elements.

- *Object* is the entity being designed, such as a physical artifact, a system, a process, or a policy. The entity is not necessarily a physical object.

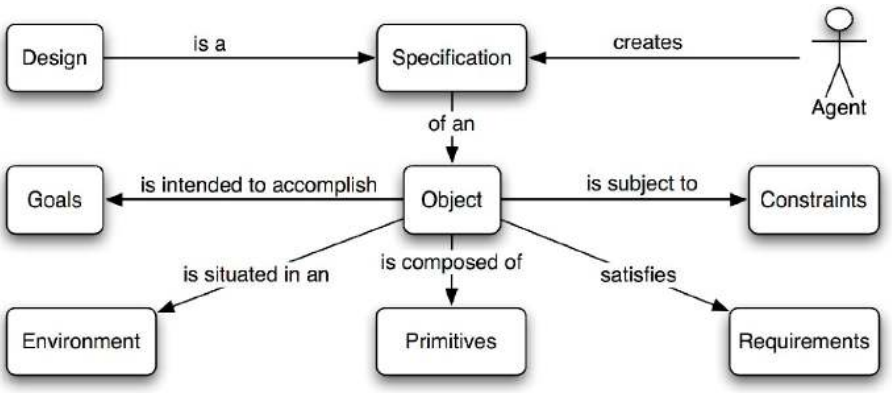


Figure 1.1: A conceptual model of design (source: [150]).

- *Specification* is a description of the design object in terms of its structures, such as the primitives used and their connections.
- *Agent* is the entity or group of entities that specifies the structural properties of the design object, e.g., scientists, engineers, or government officials.
- *Environment* of the *object* is the context in which the object is intended to exist or operate. On the other hand, Environment of the *agent* is the context in which the agent creates the design.
- *Goals* describe the desired impacts of the design object on its environment.
- *Primitives* are the set of elements from which the design object may be composed.
- *Requirements* are structural or behavioral properties that the design object must possess. A structural property is a quality the design object must possess regardless of environmental conditions or stimuli. A behavioral property is a required response to a *given* set of environmental conditions or stimuli.
- *Constraints* are structural or behavioral restrictions on the design object, where "structural" and "behavioral" have the same meaning as for requirements. Even if the design agent had infinite time and resources, a design is still constrained by, e.g., the law of physics.

Applying this definition, Table 1.1 gives two design examples of computer software and hardware systems.

Table 1.1: Examples of Design Elements.

Object Type	Symbolic Script [150]	Physical Artifact
Object	a software system	a hardware system
Specification	software programs	circuit layouts
Agent	programmers	engineers
Goals	support management of customer information	support multimedia processing
Environment	personal computers with specific operating systems	printed circuit boards with specific I/O protocols
Requirements	identify customers with certain characteristics; query latency	encode and decode certain multimedia standards; processing throughput
Primitives	instructions in a programming language	transistors with a fabrication technology
Constraints	CPU speed and memory size	gate and wire delay

A clear taxonomy of "design" can help organize, share, and reuse design knowledge [150]. Furthermore, understanding the elements of design would be useful in determining the issues and information relevant to the process of design and in planning this process [150]. Based on the taxonomy [150], I examine complex computer system design in the next section and for each design element present common approaches to addressing the system design complexity.

1.2 Coping with Complex Computer System Design

Computing is becoming ubiquitous [79], penetrating personal devices such as smart phones, smart watches, and eye glasses. The ubiquitous computing is enabled by powerful computer systems, which have been becoming more complex than ever. For example, Linux kernel 3.10 has 15,803,499 lines of code in its 2013 release, increasing by 90X with respect to its release in 1994 (176,250 lines of code) [110]. The Intel Core i7 processor has 2.6B transistors in 2014, increasing by 473X with respect to its Pentium Pro processor in 1995 (5.5M transistors) [90]. The trend of transistor density

doubling every two years is known as Moore's Law [115].

To cope with the design complexity of complex computer systems, many paradigm shifts have taken place by efficiently handling the aforementioned design elements. For example:

- Specification: to raise the level of abstraction for more productive design capture, software programming languages have evolved from machine/assembly code to various high-level languages, such as Java and Python, supporting high-level concepts like variables, objects, complex arithmetic/control expressions, and threads. For synchronous digital circuit design, the abstraction has also been raised from schematics to Register Transfer Level (RTL) hardware-description languages, such as Verilog and VHDL, which can describe sequential elements and combinational circuits. Sequential elements synchronize the circuit's operation with clock edges and stores the temporal operation results. On the other hand, combinational logic performs all the logic functions in the circuit using proper connection of logic gates.

- Agent: aiming for high design productivity, Computer-Aided Design (CAD) tools are widely adopted by human designers to automate (parts of) the design process, which leads to better design quality given limited design time and resources. In large software designs that typically involve multiple programmers, build-automation and version-control tools are essential for the development, maintenance, and integration of the entire software. In circuit design, logic synthesis tools are the standard for the automatic generation of gate-level netlists from RTL descriptions.

- Environment: to reduce the gap between the object and the agent environments, software developed for a guest machine can now run on a different host machine through virtualization techniques [170]. For hardware design, emulation and simulation techniques also become common practices, such as hardware prototyping using Field-Programmable Gate Array (FPGA) [9] and logic verification using simulation programs [78].

- Goals: "design" as a verb is to explore a design space for achieving a design object that exhibits multiple design objectives. The exploration process is illustrated in Figure 1.2, where two-dimensional design and objective spaces are used as examples. The axes of the design space represent the controllable *knobs* (aka variables or parameters) to manipulate the object within a feasible region, which is confined by the design requirements and constraints. For instance,

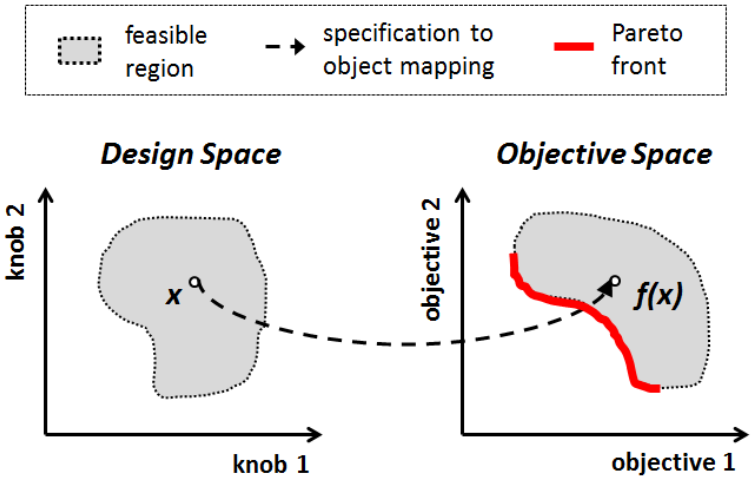


Figure 1.2: Design-space exploration. The function f for specification-to-object mapping corresponds to the construction of the design object. Suppose that the objectives are design cost and performance in terms of latency (both to be minimized). The red Pareto front features the optimized objects.

the knobs of a software compiler are its compilation flags. On the other hand, the objective axes stand for the qualities of the achievable objects. In many scientific and engineering fields, the objectives are *conflicting* [23], e.g., the development cost, memory footprint, and query efficiency of database design in computer science. Therefore, from an optimization perspective, the "best" design objects are the ones that cannot improve in any objective without sacrificing at least another. For instance, the database systems that are either "most query efficient with largest memory usage" or "least memory hungry with longest query latency" are equally best in a multi-objective setting. These design objects are considered *Pareto-optimal* (or Pareto-efficient) and are called *Pareto points* [23]. The collection of Pareto points are called *Pareto set* (or Pareto front, or Pareto curve) [23]. In Section 8.3, I survey more computer-science applications of Pareto-set discovery.

- Primitives: to avoid reinventing the wheel, standard libraries are common features of high-level programming languages. The libraries are collections of frequent and generic routines that were optimized beforehand for later reuse, such as sorting on arbitrary data types. Object-oriented programming also promotes the design of standard template libraries using

abstract data types [95]. For circuit design, standard cells and Intellectual Property (IP) blocks are finer- and coarser-grained design primitives, respectively: a standard cell is a group of wired transistors that provides a basic Boolean function or a storage function; an IP block is a functional unit that is synthesizable (aka soft IP) or already defined in a transistor-layout format (hard IP) [155]. Both standard cells and IP blocks are instrumental in the success of the fabless business model in the semiconductor industry [128].

- Requirements: to verify a specification with critical design requirements, such as safety requirements, formal verification methods have been applied to both software and hardware design in industrial settings. Formal methods, such as model checking [38], guarantee the completeness (i.e., stimuli independent) and are able to return counterexamples for fixing the specification. Another approach for tackling design requirements is based on correct-by-construction methodologies. Software examples for *algorithm* design are the problem-solving paradigms "divide and conquer" and "dynamic programming", where the solution to the main problem can be constructed from the solutions to its subproblems [44], A hardware example for *electronic system* design is the "globally asynchronous locally synchronous" architecture for composing locally-clocked IP components via asynchronous communication protocols, while preserving semantics and avoiding deadlocks [146].

- Constraints: the limits of computing have been continuously pushed with new platforms, such as networked, distributed, or parallel computer systems. Each computer unit in these systems has also been continuously improved with new technologies. In the context of computer hardware design, the next section introduces major technology breakthroughs and presents new challenges.

1.3 Challenges of Electronic System Level Design

Low-cost Very Large Scale Integration (VLSI) electronics have revolutionized daily life and expanded the role of computation in science and engineering [100]. The revolution was supported by three main semiconductor areas: *(i)* device and manufacturing, *(ii)* circuits and architectures, and *(iii)* CAD methodologies and tools. For the first two areas, the past decade has witnessed some technology breakthroughs *reaching production*, e.g., new manufacturing technologies such

as FinFET [46; 88] and 3D integration [47; 152], and new computer architectures such as multi-core processors [97; 126; 127; 141; 183] and heterogeneous System-on-Chip (SoC) [37; 49; 103; 193]. In contrast, there have not been major CAD breakthroughs making an industrial impact comparable to such innovations of the past as circuit simulation, automatic place and route, RTL description languages, and logic synthesis. These CAD technologies were already mature in the 90s. In order to continue sustaining the progress of the semiconductor industry in the face of growing system-design complexity, CAD researchers must rely on design reuse to address the complexity [34; 155] and raise the level of abstraction above RTL to enable Electronic System Level (ESL) design [15; 76].

The heterogeneous SoC, which features a mix set of programmable cores and task-specialized accelerators, has become the dominant computer architecture thanks to its energy efficiency[1]. The SoC structure is composed of heterogeneous components, each responsible for either computation, communication, or data cache. This modular structure addresses the ESL design complexity by parameterizing, exploring, and integrating primitive components. These components can be pre-designed, pre-verified, and pre-characterized as IP blocks, aiming for broad reuse in various systems (or subsystems) for maximum design productivity, i.e. component-based design [34; 155]. In order to achieve a 10X design-productivity improvement, ITRS predicted that by 2020 a stationary or consumer SoC would consist of 90% reused components (assuming a constant tool effort) [91]. In industry, the 2001–2013 CAGR of semiconductor-IP revenue is indeed a good 13% [65]. Preferably, the design of SoC component is specified in high-level languages (i.e. soft IP's), so that they can be implemented with automatic synthesis tools to serve various ESL requirements.

The synthesis of RTL code from C-like languages (such as SystemC [4]) is known as High-Level Synthesis (HLS) [48]. Empirically, when designers move from RTL to high-level specifications, the code density can be reduced by 7X–10X [42], and therefore the specifications are easier to develop, debug, and maintain. Given a high-level specification with user-specified *knob settings* (see footnote [2]), HLS can automatically generate RTL implementations with different micro-architectures,

[1] Section 2.1 provides more background for the rise of heterogeneous architectures.

[2] A *knob setting* is a complete configuration of synthesis knobs onto a design specification. In other words, a knob setting corresponds to a *synthesis scenario* or a *synthesis script*, which is conceptually depicted as the point x in Figure 1.2. Throughout this dissertation, I use "knob setting" and "synthesis scenario" interchangeably.

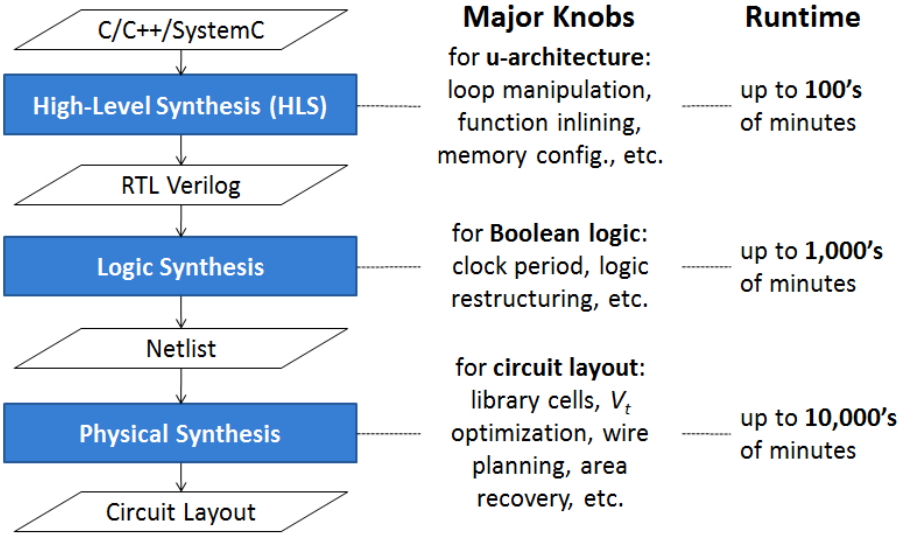


Figure 1.3: A synthesis-centric design flow.

each exhibiting unique trade-offs among multiple design objectives[3]. Moreover, by adopting high-level languages, the co-design of software and hardware for embedded systems can start earlier in the design cycle [175] using virtual platforms [111]. In industry, the 2001–2013 CAGR of ESL-tool revenue (11%) is already ranked at the top among all the front-end VLSI-design tools, which include logic synthesis, RTL simulation, formal verification, timing analysis, etc [65].

Overall, in order to realize component-based ESL design, powerful automatic synthesis tools are essential, because the creation and maintenance of reusable components are estimated to be 2–5X more difficult than their creation for one-time use [91]. A synthesis-centric design flow includes three synthesis stages: HLS, logic synthesis, and physical synthesis (Figure 1.3). HLS translates a high-level design specification to an RTL implementation, which is further synthesized to a gate-level netlist by logic synthesis, and finally placed and routed by physical synthesis. The goals of the three synthesis stages are different: *(i)* HLS optimizes RTL implementations with various micro-architectural knobs, such as loop manipulation, function inlining, memory configuration, and many others, *(ii)* logic synthesis optimizes the area/power of a netlist by restructuring Boolean logic and mapping it to standard cells, under a clock-period constraint, and *(iii)* physical synthesis

[3] The trade-offs are conceptually captured by the feasible region of the objective space in Figure 1.2. HLS (or "synthesis" in general) is represented by the mapping function f in the figure.

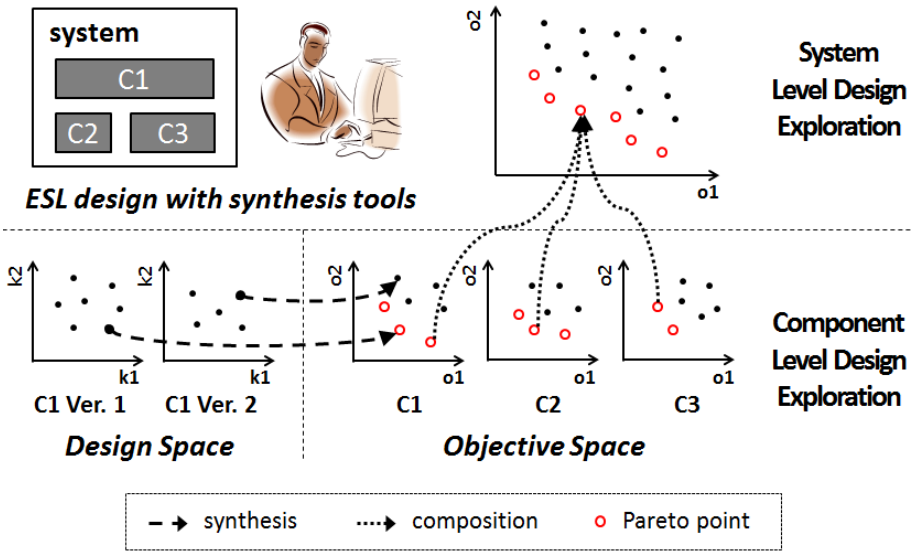


Figure 1.4: Challenges of ESL design with synthesis tools.

determines the layout of the netlist using various optimization techniques, such as gate/wire sizing, buffer insertion/sizing, V_t swapping, routing layer/track assignment, and many others. Although modern synthesis tools are powerful thanks to these many synthesis knobs, the runtime of a synthesis job is significant, ranging from hours to days on modern workstations.

Across all the synthesis stages, the synthesis-driven design methodology faces several big challenges as described below. Figure 1.4 illustrates these challenges using a simple example of a system design that is composed of three components (C1, C2, and C3).

- **Design-Space Exploration (DSE)** (represented by the Pareto points in Figure 1.4). Classical DSE optimizes one single objective given others as constraints, such as the area minimization with clock-period constraints that is used in logic synthesis. However, modern VLSI design requires to explore a *multi-objective* design space, such as performance vs. energy [164] and other objectives such as area [52], power [119], manufacturability [36], and reliability [102]. To this end, multi-objective DSE should search for a Pareto set, as opposed to a single Pareto point. Unfortunately, existing synthesis tools can only return one implementation per synthesis run and that implementation is not even guaranteed to be a Pareto point. Therefore, the search of the Pareto set requires extensive synthesis runs, each taking significant runtime.

- **System-Level Composition** (represented by the composition arcs in Figure 1.4). The design of heterogeneous SoCs relies on the effective reuse and integration of optimal component implementations. As holistic synthesis is infeasible (due to prohibitively high requirements in terms of tool runtime and memory occupation), efficient system-composition tools are desirable for SoC optimization. The composition is done by first running component-level synthesis and then combining proper component implementations that form the system-level Pareto set. The ideal composition tool should be able to identify the critical components that can have the most contributions to the system-level Pareto set. Moreover, considering the long synthesis-tool runtime, this tool should be adaptive in spending component synthesis jobs (top-down), without the need to fully characterize each component-level Pareto set (bottom-up).

- **Synthesis-Tool Usage** (represented by the synthesis arcs in Figure 1.4). The elusive optimal setting of synthesis knobs and the long runtime of synthesis jobs make synthesis tools hard to use. The synthesis Quality of Results (QoR) is often design specific; there is no universal recipe of knob settings for achieving the best QoR. Although the example of Figure 1.4 shows only two knobs, the actual number of knob settings for HLS grows exponentially with the specification complexity; for integrated logic and physical synthesis, the number can reach $O(2^p)$ with the number of knobs $p \in [100, 400]$. The sheer number of synthesis scenarios (i.e. knob settings) together with the long runtime required by each synthesis scenario prohibits exhaustive synthesis in practice.

- **Synthesis-Aware Specification Refinement** (represented by the design space derived from different versions of a component specification (C1 Ver. 1 and C1 Ver. 2) in Figure 1.4). These versions satisfy the same functional requirements, but differ in the detailed descriptions, e.g. using different algorithms for sorting. The exploration of synthesizable specifications has long been done manually and is not addressed automatically by CAD tools. Very often, a restructure of the specification can change the synthesis-explorable design space, thus yielding different synthesis QoR. However, component-based design requires extensive reuse of high-quality synthesizable components across product cycles, design teams, and system companies. Therefore, as SoC designs are becoming more complex, the tool support for synthesis-aware

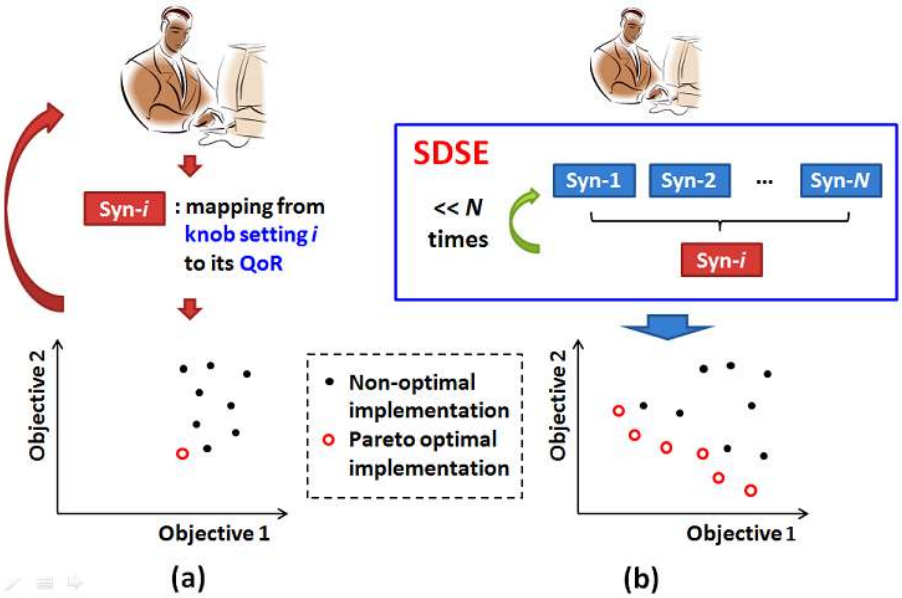


Figure 1.5: (a) Traditional DSE is an ad hoc process to search one (or few) optimal design implementation. (b) Supervised DSE (SDSE) is an abstraction layer that returns an approximate Pareto set by autonomously supervising synthesis jobs.

specification refinement is becoming more and more desirable.

1.4 Thesis Contributions

In order to address the aforementioned challenges, I propose **Supervised Design-Space Exploration (SDSE)**: *SDSE is an abstraction layer between a designer and a synthesis tool, aiming to autonomously supervise synthesis jobs for design-space exploration. The system Pareto set is approximated by combing the required component implementations, which are achieved by running a minimum number of synthesis jobs. The synthesis jobs are adapted to the most influential system components with the most effective knob settings for synthesizing them.*

Specifically, my thesis makes the following contributions:

- I introduce a new abstraction layer for autonomous DSE with synthesis tools. Figure 1.5 illustrates the traditional and the supervised DSE approaches. The traditional approach requires *(i)* human analysis on synthesis QoR and *(ii)* human decision among a huge number of candidates to select the next synthesis scenario [123]. The manual efforts required by the

traditional approach limit the exploration result to one (or few) optimal implementation. In contrast, SDSE offloads the analysis and decision making to an autonomous process, such that the designer can focus on extra design optimizations beyond synthesis exploration. Moreover, SDSE aims at not only minimizing the number of synthesis jobs for DSE but also returning an approximate Pareto set. The Pareto set brings the following benefits: *(i)* at the stage of early design exploration, when the preferences of design objectives are unclear or subjective to change, the Pareto set provides a global view of equally good design options, thus limiting the risk of being trapped into local optima, *(ii)* in a design/implementation flow, the Pareto set can be refined at the next level(s) of abstraction without the need to start from scratch, and *(iii)* for final design optimization, the Pareto set features a portfolio of optimal implementations that can be reused in the future across design projects and design teams/houses.

- I propose two novel compositional frameworks that can minimize the number of component synthesis jobs for approximating the system-level Pareto set (to optimize the composition arcs of Figure 1.4). Together with the frameworks, I propose novel algorithms for identifying the most influential system components, so that most of the synthesis jobs are adapted to those components for the maximum return of investment. The system Pareto set is approximated by combining the component synthesis results. I have applied SDSE for system-level exploration with commercial high-level and logic synthesis tools.

- I present two enhanced iterative-refinement frameworks that can minimize the number of synthesis jobs for component-level DSE using advanced learning methods (to optimize the synthesis arcs of Figure 1.4). The learning methods can better estimate synthesis QoR for unknown synthesis scenarios, thereby avoiding unnecessary synthesis jobs and adapting instead to the Pareto-promising scenarios. I have applied SDSE for component-level exploration with industrial high-level and physical synthesis tools, and have successfully integrated SDSE research into the IBM Synthesis Tuning System for production processor design.

- I consider SDSE extensible and general, seeing great potential for applications beyond synthesis and electronic system design. SDSE is extensible with code-refactoring techniques which introduce "virtual" knobs that are not natively offered by the synthesis tools (to optimize the

Table 1.2: Thesis Contributions to The Design Elements [150].

Design Element	Contributions	Chapter
Specification	a new abstraction layer between designers and synthesis tools;	3–6
	a code-refactoring methodology	7
Agent	automatic setting of synthesis knobs;	3–6
	automatic selection of component implementations;	3–4
	automatic suggestion of code-refactoring options	7
Goals	discovery of post-synthesis Pareto set	3–7
Environment	learning methods as synthesis surrogates for design evaluation	5–6
Requirements	adaptive component synthesis while satisfying system performance requirements	3–4
Primitives	synthesis tools as query oracles	3–7
Constraints	automation of large-scale design-space exploration	3–7

component specification of Figure 1.4). In particular, I propose a pattern-based refactoring methodology to further improve the DSE results in terms of both HLS QoR and tool runtime. Moreover, SDSE could be generalized *(i)* to create a platform for synthesis knob/library deployment/testing on real customer designs, *(ii)* to promote more collaboration between system architects and component designers, and *(iii)* to explore designs beyond VLSI chips, for applications which also require extensive CAD-tool invocations with different knob settings.

Table 1.2 summarizes my contributions to the design elements that were defined in Section 1.1.

1.5 Outline of The Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, I describe the background for the rise of heterogeneous SoCs, existing CAD tools for SoC design, and the need for a new generation

of CAD tools. I also present an overview of the related works on the topic of "multi-objective DSE with CAD tools" in Chapter 2. Afterwards, the dissertation is partitioned into three parts before Chapter 9 draws conclusions:

- **Part I: System-Level Design-Space Exploration** describes two SDSE frameworks for compositional system exploration. In particular, this part presents "synthesis planning for exploring ESL design" using HLS (Chapter 3) and "compositional approximation for SoC characterization" using logic synthesis (Chapter 4).
- **Part II: Component-Level Design-Space Exploration** describes two SDSE frameworks for iterative component exploration. In particular, this part presents "knob-setting optimization for high-level synthesis" (Chapter 5) and "parameter tuning for physical synthesis" (Chapter 6).
- **Part III: Extending Supervised Design-Space Exploration** describes a pattern-based code-refactoring methodology to enhance the HLS DSE results (Chapter 7) and sketches several future research directions based on SDSE (Chapter 8).

Finally, in Appendix A I describe four prototype SDSE tools that I develop:

- *Synthesis Planning for Exploring ESL Design* (SPEED): a prototype tool that implements SDSE with HLS for system-level DSE. The SPEED methodology and its companion algorithms are presented in Chapter 3.
- *Compositional Approximation of Pareto Sets* (CAPS): a prototype tool that implements SDSE with logic synthesis for system-level DSE. The CAPS algorithm and its companion heuristic modifications are presented in Chapter 4.
- *Learning-based Exploration with Intelligent Sampling, Transductive Experimental Design, and Refinement* (LEISTER): a prototype tool that implements SDSE with HLS for component-level DSE. LEISTER's underlying algorithms are presented in Chapter 5.
- *Enhanced gSpan* (EgSpan): a prototype tool for enhanced DSE with pattern-based code-refactoring. The code-refactoring methodology and the EgSpan algorithm are presented in Chapter 7.

Chapter 2

Background

Semiconductor chips have been the major driving force for the advancements of diverse computer systems, including personal computers, network servers, government/bank mainframes, and supercomputers. The semiconductor performance and cost continuously improve, making possible the global expansion of information technology, which is estimated to ascribe up to 40 percent of the global productivity growth in the period of 1993–2013 [17]. The improvements have, however, become increasingly difficult ever since early 2000. This chapter gives a brief introduction to the rise of heterogeneous System-on-Chip (SoC) and its design complexity, the existing Computer-Aided-Design (CAD) tools for addressing the design complexity, and the need for a new generation of CAD tools. After the introduction section, an overview of the related works on "design-space exploration with CAD tools" is presented to highlight the novelty of SDSE.

2.1 Design Complexity of SoC with CAD Tools

Since mid 80s, the application-level uniprocessor performance and the semiconductor device speed have increased by over 3,000X and 100X, respectively [87]; meanwhile, the fabrication cost per semiconductor gate has decreased linearly up to the 28nm-technology node [89]. The improvements were driven by a triplet of actions: *(i)* scale the process technology down, *(ii)* scale the supply voltage down, and *(iii)* scale the clock frequency up. Technology scaling shrinks the feature size (e.g. the size of the transistor gate), such that more transistors can be crammed onto a single die at the next technology node. Ideally, *power density* should remain constant with technology

scaling *and* voltage scaling (aka Dennard scaling [57]). However, voltage scaling slowed down due to excessive leakage current [56]; meanwhile, frequency was scaled faster than dictated by technology scaling [87]. Since technology scaling continued, processor design soon became power-limited and the frequency scaling practically stopped in the early 2000.

Given the power constraint, processor design resorted to *homogeneous* multi-core architectures by adding more processor cores to each die. Parallel processing allows each core to be more energy efficient by having a lower peak performance (due to the lower frequency), whereas multiple cores can increase the overall computing throughput performance. Even better, for maximum power saving, each core can dynamically adjust its operating voltage and frequency according to the currently-expected workloads, a technique known as Dynamic Voltage and Frequency Scaling. However, parallelism alone hits diminishing returns at both the low-energy and high-performance extremes [87]. That is, sacrificing large performance factors returns small energy savings, and vice versa. Moreover, other factors affecting the effectiveness of parallelism exist: *(i)* the parallelizable portions of an application can be limited by its nature (aka Amdahl's law) [7], *(ii)* a parallel algorithm may require higher time complexity than its sequential counterparts [116], and *(iii)* with advanced technology, cache-memory power consumption can be comparable to the power spent on computation [118]. As a result, homogeneous parallelism alone can not promise a long-term performance scaling similar to the one allowed by technology scaling for four decades.

2.1.1 The Rise of Heterogeneous Architectures

Given the limitations of technology scaling and homogeneous parallelism, specialized hardware components (aka *accelerators*) emerge as energy-efficient solutions to work together with general-purpose processor cores. Accelerators are application-specific integrated circuits (ASICs) customized for dedicated tasks with improved computation and memory-access efficiencies. Graphic Processing Units (GPUs) can be regarded as a programmable form of accelerators [132]. A *heterogeneous* System-on-Chip (SoC) integrates general-purpose processor cores, GPU cores, and accelerator cores, in addition to other components for on-chip communication and data caching [21]. In fact, in the 2011–2013 period, the energy efficiency of commercial "processor + GPU" SoCs based on 22nm–32nm technologies is still below 10 MOPS/mW [49; 139; 192], while the efficiency of dedicated hardware designs based on 65nm–0.13um technologies can already reach the 100–1,000 MOPS/mW

range for applications such as multimedia [101], biomedicine [12], artificial intelligence [129], and computer vision [138; 179].

Despite the energy advantage, SoC design is facing demanding time-to-market schedules as well as ever increasing design costs. Typically, consumer processors or desktop SoCs follow a 12–18 month product cycle [6; 90], which is even shorter for mobile SoCs [148; 156]. Within the stringent cycles, the design complexity of a consumer SoC ranging from 47M transistors (Intel 45nm Atom, 2008) to 2B transistors (Apple 20nm A8, 2014) needs to be carefully addressed by usually teams of 10's–100's of engineers [184]. Converted into design costs, the sheer design complexity corresponds to high Non-Recurring-Engineering (NRE) investments. For instance, a 180M-gate chip costs $80M–$90M with a 28nm technology, and a 240M-gate chip would cost $150M–$160M with a 20nm technology [89; 94; 168]. Even worse, an over 300M-gate 16/14nm chip is projected to cost more than $310M [89; 94; 168]. Moreover, the fabrication of an SoC from the photomask generation to the chip packaging can take 6–8 weeks [184]. With all these economic factors considered, the first-pass success is critical for commercial SoC design.

2.1.2 ASIC Design Flow

In order to address the enormous design complexity, Very Large Scale Integration (VLSI) design methodologies have progressed over 40+ years, from full-custom (manual) design to computer-aided design (CAD) with increasingly higher levels of abstraction for design capture. Fig. 2.1 depicts the chronological evolution across four decades when chip design has progressed from *(i)* schematic models to *(ii)* logic-gate (aka standard-cell) models, *(iii)* Register Transfer Level (RTL) models described by hardware description languages such as VHDL or Verilog, and finally *(iv)* Transaction Level Modeling (TLM) using high-level languages such as SystemC. By raising the level of abstraction, VLSI designs are captured in coarser granularity. This makes the number of design primitives remain manageable, thus increasing design productivity to handle the growing design scale. The multi-decade evolution is a joint result of the tight collaboration between industry (including worldwide design houses, CAD-tool vendors, and semiconductor foundries) and academia (including dedicated publication venues of 10+ active IEEE and/or ACM journals and conferences). From an economic perspective, the established ecosystem that makes building VLSI design successful is a reason why replacing the semiconductor technology (e.g. with quantum computing) for

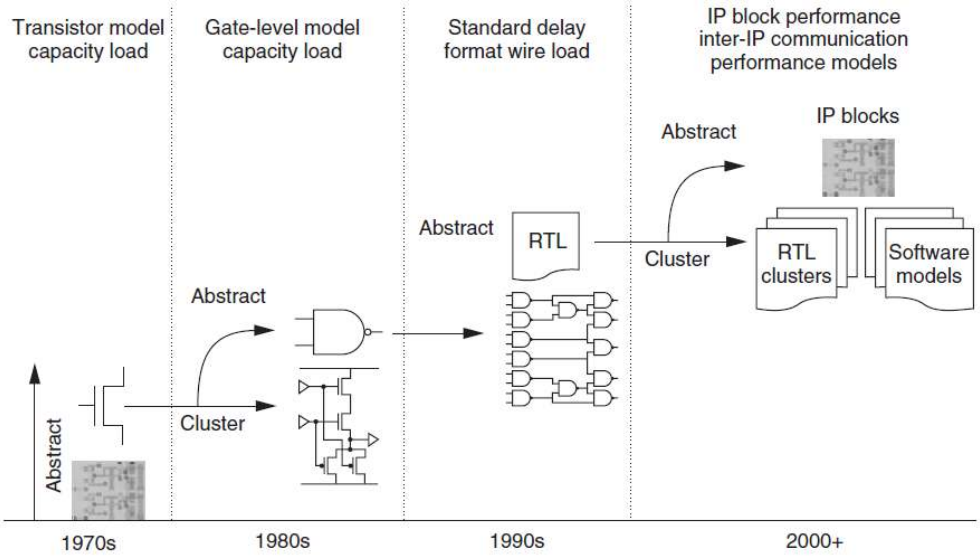


Figure 2.1: The evolution of design abstraction for VLSI design (source: [157]).

resolving the power problem can be too expansive in the foreseeable future [87].

Fig. 2.2 shows a common ASIC design flow with CAD tools starting from high-level specification[1]. The flow is divided into two parts: *(i)* front-end design and *(ii)* back-end design. Regarding synthesis and verification, the front-end and back-end design focus on the *functional* and *physical* aspects, respectively. While the mainstream front-end design starts with a hand-crafted RTL specification, High-Level Synthesis (HLS) is becoming popular as explained in Chapter 1. The flow proceeds as follows.

- First, the high-level specification is synthesized by HLS tools into a synthesizable RTL specification. This needs to be verified that is behaviorally correct via simulation tools (e.g. virtual platforms [111]) or formal approaches (e.g. sequential equivalence checking [124]). The high-level design emphasizes micro-architectural optimization and its interaction with software.

- Afterwards, the RTL specification is synthesized by Logic Synthesis tools to a netlist of standard cells which implements the design with logic gates (such as AND and OR), sequential elements (such as registers and memories), and the connection among them. Also, the netlist needs to be verified via logic simulation or equivalence checking. In addition, Static Timing

[1] More details about an ASIC design flow can be found in [184]. For instance, Fig. 2.2 omits scan-logic insertion for circuit testing, which per se is a research field in CAD [108].

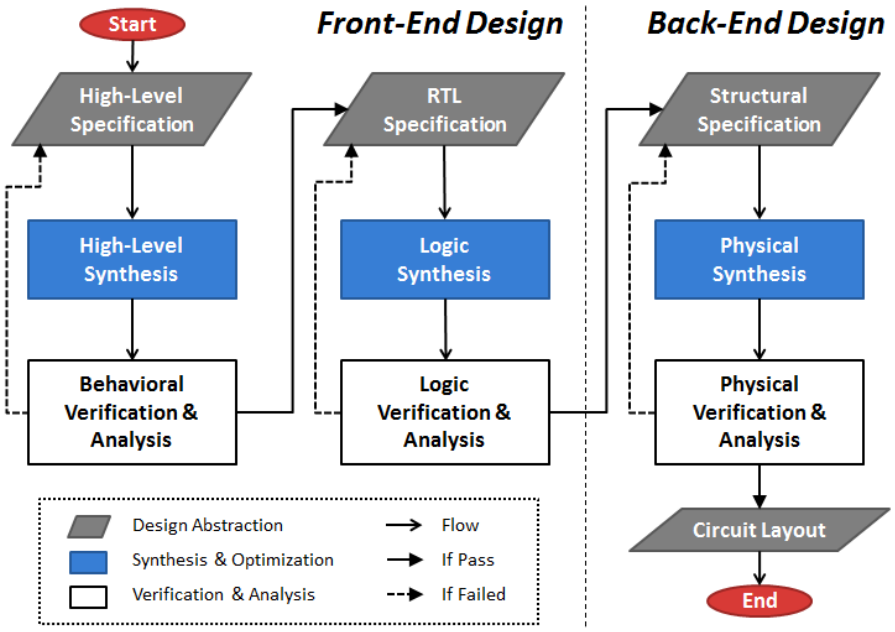


Figure 2.2: An ASIC design flow.

Analysis (STA) is performed on the netlist for assessing its performance (i.e. the minimum clock period) based on primitive interconnect-load models.

- The verified netlist (i.e. structural specification) kicks off the back-end design by running Physical Synthesis tools, which generate the circuit layout of the design. If the layout passes physical verification and analysis (including STA with refined interconnect models, design-rule checking, logic-versus-schematic checking, and many others), the circuit can be signed off for fabrication (aka *design closure*).

Nowadays, this flow is not only common to ASIC design but also increasingly adopted for high-performance processor design, in order to cope with its sheer complexity [196].

Each synthesis step in the ASIC design flow involves multiple computation-intensive tasks [52; 166]:

- Major HLS tasks include: *(i)* allocating resources (e.g. adders, multipliers, etc.) for implementing target behaviors, *(ii)* scheduling operators (e.g. additions, multiplications, etc.) across multiple clock cycles for optimizing the latency of the design and the utilization of resources, and *(iii)* binding operators to resource instances and binding variables to register

instances to minimize implementation costs such as area and power.

- Major logic-synthesis tasks include: *(i)* minimizing a multi-level Boolean network by factoring with common sub-expressions and sharing minterms, *(ii)* mapping the Boolean network with technology-dependent standard cells to minimize implementation costs, and *(iii)* iteratively refining the synthesis results.

- Major physical-synthesis tasks include: *(i)* floor-planning large design blocks to minimize their interconnect wire length and to construct robust power-supply networks, *(ii)* placing standard cells in rows to minimize the total wire length and to avoid routing congestion, *(iii)* synthesizing clock trees to distribute clock signals to registers with minimized clock skews, *(iv)* routing the placed standard cells via multiple metal layers while honoring design rules such as wire width, wire spacing, etc., and *(v)* restructuring the netlist, sizing standard cells, and/or inserting buffer cells to optimize the circuit performance/power/area, if necessary.

2.1.3 Next Generation CAD Tools

Although the CAD-tool-based design flow has been instrumental in the commercial success of VLSI design, new challenges arise around the existing tools with respect to Quality of Results (QoR), tool runtime, design exploration, and user-tool interface.

- Most of the tasks in the three synthesis steps are computationally intractable (at least NP-hard) [52; 166]. Because of this intractability plus the sheer size of VLSI design, practical (commercial) synthesis tools are based on heuristics. However, they still require significant CPU time per synthesis run, which can range from hour(s) to 10+ days to deliver acceptable QoR.

- Traditionally, these synthesis tools were designed to perform performance-constrained *area* or *power* optimization [52; 166], i.e., *single*-objective optimization. However, other objectives due to aggressive technology scaling, such as design manufacturability [36; 189] and design reliability [32; 102], are also becoming important when the synthesis results are evaluated. Even with advanced synthesis features that can optimize these additional objectives during synthesis, each synthesis run still only delivers *one* implementation, as opposed to a Pareto set of implementations. In order to discover the Pareto set, "synthesis exploration" with extensive

knob settings as well as "QoR analyses" with human decision making are inevitable. In fact, analysis reports of the size of 1–2TB are common for the design of a 22nm 10B+ transistor chip [173], showing the complexity of manual design exploration with CAD tools.

Moreover, the Pareto set is necessary to provide useful alternative designs when *(i)* the designer does not (yet) have clear preferences over the multiple objectives (e.g. early in the design cycle), *(ii)* the objectives are not well-defined (e.g. they are subjective to the preferences of multiple designers) or hard to be quantified (e.g. the true cost function is not available [63]), and *(iii)* the Pareto points are intermediate products for subsequent explorations in a design flow (e.g. an area-efficient circuit placement may be unroutable).

- To achieve design closure faster, the steps in the design flow require customized settings and sequences that are both tailored toward certain design objectives. In addition, due to the lack of good predictive cost functions, the execution of the design flow cannot be accomplished by a fixed set of discrete steps, but instead requires incremental cost analyses [157]. Put together, these requirements call for a new generation of CAD tools that provide: *(i)* appropriate abstractions, *(ii)* simplified and standardized user-tool interfaces, *(iii)* intuitive design environments, and *(iv)* scalable design methodologies [24].

All of these challenges reflect a perspective on the future of CAD research and, correspondingly, of the Electronic Design Automation (EDA) industry: "*Very few of these challenges require drastic innovation in the key algorithms underlying the EDA tools. Instead they all point to the design flows and the design environment through which the designers interact with design tools and to the scale of problems that need to be solved.*" [173] Nevertheless, in order to scale withe the problem size, innovative scalable algorithms are still required at the design flow/methodology level [24].

Making contributions toward these perspectives, I propose Supervised Design-Space Exploration (SDSE), a novel abstraction layer which offloads manual design exploration processes in a design flow to autonomous and scalable processes. Thus, SDSE shortens and simplifies the learning curve of mastering synthesis tools. I have developed prototypes to apply SDSE to all the three synthesis steps for the design of accelerator cores as well as processor cores, which all demand scalable exploration algorithms.

2.2 Overview of Related Works

The majority of existing works on "DSE with CAD tools" treat a design as a whole object. Very few of them take advantages of the SoC structure by *compositionally* exploring the system design. A compositional approach combines the component implementations that form the system Pareto set. The compositionality is revealed by satisfying system-level design requirements (e.g. system throughputs) based on the guarantee of component-level requirements (e.g. component latencies) in a way that is correct by construction.

This section first overviews the major existing works that can be applied to the SoC *component* design and then switches to the works that explore the SoC *system* design compositionally. Detailed comparisons with existing works are distributed into Chapters 3 to 7.

2.2.1 Component-Level Design-Space Exploration

Existing works for component-level DSE with CAD tools can be categorized into four types:

- First of all, *local-search* heuristics are the most popular choices, since they can flexibly encode and perturb knob settings in search for the Pareto-optimal ones. Typical heuristics include: *(i)* Genetic Algorithm based approaches [11] (e.g., Strength Pareto Evolutionary Algorithm [66] and Non-dominated Sorting Genetic Algorithm [114; 177]) and *(ii)* Simulated Annealing based approaches [177] (e.g., Adaptive Simulated Annealing [159]). These local-search heuristics, however, invoke a considerable number of trial synthesis jobs in each search iteration, yet a large portion of the synthesis results is dropped as the algorithm proceeds. Consequently, this type of approaches is *not scalable* when working with long-runtime synthesis tools for large-scale DSE. In contrast, SDSE targets the application with synthesis tools, which demand scalable exploration algorithms.

- To avoid the intensive computation required by synthesis tools, the second type of approaches aims at *predicting* synthesis QoR, instead of invoking actual synthesis jobs to acquire the QoR. For example, fuzzy systems or learning models are embedded into a Genetic Algorithm framework to predict the QoR [11; 161; 43]. Similar approaches using statistical inference [18; 82; 120] or learning models [121; 133; 134; 186; 197] are employed within an *iterative-refinement*

framework. In general, the iterative-refinement framework is shown to be superior to local-search-based frameworks [121; 134; 186; 197]. However, all the iterative-refinement works have been applied to *processor simulators*, as opposed to ASIC synthesis tools. Therefore, the effectiveness of "learning-based iterative refinement" for synthesis-driven DSE remains unclear. In contrast, SDSE features two *enhanced* iterative-refinement frameworks using advanced training, sampling, and learning methods. The effectiveness of SDSE is validated by *(i)* accelerator-core design with commercial HLS tools and *(ii)* processor-core design with industrial physical-synthesis tools.

- Alternatively, a third type of approaches applies *pre-pruning* of the design space in order to reduce the search space. For instance, parameter-dependency graphs are utilized so that Pareto points are first generated locally based on dependent parameters [77]. This preprocessing is followed by exhaustive combination of the local Pareto points [77] (or accelerated by local search [136]). Parameter-clustering [160] also belongs to this type of approaches. Although these works can reduce the size of search space, the final exploration results depend on the correctness of parameter dependencies, which in practice is not easy to obtain a-priori. This type of approaches still requires substantial design expertise and is therefore *restricted* to a small set of applications. In contrast, SDSE does not require the design-space-reduction pre-processes to be scalable (Chapters 3, 4, and 5), but can still be augmented with design-space-reduction techniques (Chapter 6).

- The fourth type of approaches generates Pareto points based on *well-defined "knob-setting to QoR" mapping* functions. For instance, to uniformly sample a QoR space, a multi-objective optimization problem can be transformed into a sequence of single-objective problems via weighted sums of the objectives [169]. However, this technique works only for convex problem formulations; unfortunately, synthesis-derived Pareto sets can be non-convex [113]. Another algorithm for generating Pareto points has been presented for custom-instruction selection [20]. In this application domain, the mapping function from knob settings to QoR is well defined and polynomial-time computable. Still other similar works for custom circuit design are based on differentiable closed-form mapping functions [14; 200]. Nevertheless, the mapping functions of commercial synthesis tools are not publicly available. As a result, these

techniques are *not generally applicable* to synthesis-driven DSE, which is the target of SDSE.

2.2.2 System-Level Design-Space Exploration

The prior works summarized in the previous subsection explore their target design as a whole object. The holistic synthesis of an entire SoC, however, requires prohibitively-high computational efforts such as CPU time and memory occupation. Alternatively, there are works (also for processor simulation) [20; 66; 75; 85; 167] that take advantages of the design hierarchy: they generate Pareto points in a *bottom-up* fashion by first fully characterizing all the SoC components and then exhaustively combining them. Hence, even in a compositional setting, these existing works are still considered exhaustive. In contrast, I present two compositional SDSE frameworks for High-Level and Logic Synthesis, respectively, which explore the system design space by dynamically identifying the components that have the most impacts on the system-level Pareto set. Empirically, SDSE's *top-down* exploration approach can approximate the Pareto set with an order of magnitude fewer synthesis jobs (or tool runtime) than the exhaustive approach.

Part I

System-Level Design-Space Exploration

Chapter 3

Synthesis Planning for Exploring ESL Design

The growing SoC complexity has motivated the adoption of soft-IP components and high-level synthesis (HLS) tools, as introduced in Section 1.3. From an industrial viewpoint [106], the ideal *component-based design paradigm* still requires major progress on two levels: (i) at the component-level, in the design and cost/performance modeling of an IP component to increase its reusability across potential SoC designs and to enable architecture exploration; (ii) at the system-level, in the automatic integration of IP components to find an optimal implementation of a given SoC.

Contributions. In this chapter, we propose a HLS-driven design methodology for compositional system-level design exploration that consists of two main steps. The first step is the development of libraries of synthesizable components which are both designed in high-level languages and characterized through HLS to maximize their reusability. In the second step, from the specification of a given system we construct a *system-level Pareto front* where each point represents a distinct implementation composed of optimal component instances, each of which is implemented in RTL by synthesizing its high-level specification.

Our methodology aims at minimizing the total number of component synthesis to achieve a rich set of target system implementations. In particular, we propose:

- a concise component-library format, which preserves the freedom of system designers to select a wide range of component implementation alternatives to fit the specific design context of a

system;

- an algorithm that, for a given system design context, prunes the design space of a component to minimize the set of its implementation candidates for integration into the system.
- an algorithm for system-level design space exploration which performs *HLS-planning* before invoking actual synthesis on the components. Thus, all the planned synthesis tasks can be run *in parallel* to minimize the total exploration time.
- a CAD tool called SPEED that implements these two algorithms. SPEED stands for "Synthesis Planning for Exploring ESL Design". A prototype implementation of SPEED is described in Appendix A.1

3.1 Problem Description

Model of Computation. We adopt timed marked graph (TMG), a subclass of Petri nets (PN) [125], as the main model of computation. PN-based models allow compositional system-level performance analysis [178]. The simplicity of TMG allows us to perform rapid *static* analysis of interesting classes of systems without the need for simulation. Still, we also discuss extensions of TMG to model more general systems.

Definition 1. *A PN is a bipartite graph defined as a 5-tuple* (P, T, F, w, M_0)*, where* P *is a set of* m *places,* T *is a set of* n *transitions,* $F : (P \times T) \cup (T \times P)$ *is a set of arcs,* $w : F \to \mathbb{N}^+$ *is an arc weighting function, and* $M_0 \in \mathbb{N}^m$ *is the initial marking (i.e., the number of tokens at each* $p \in P$*) of the net.*

Let $\bullet t = \{p|(p,t) \in F\}$ denote the set of input places of a transition t and $t\bullet = \{p|(t,p) \in F\}$ the set of output places. Let $\bullet p = \{t|(t,p) \in F\}$ denote the set of input transitions of a place p and $p\bullet = \{t|(p,t) \in F\}$ the set of output transitions. When a transition t fires, $w(p,t)$ tokens are removed from each $p \in \bullet t$ and $w(t,p)$ tokens are added to each $p \in t\bullet$. As a *firing rule* we assume that t fires as soon as it is enabled, i.e. each $p \in \bullet t$ has at least $w(p,t)$ tokens. A PN is *live* iff no transition can be disabled after any marking reachable from M_0. A PN is *safe* iff the number of tokens at each place does not exceed one for any marking reachable from M_0. An example live and safe PN with four transitions and five places is shown in Fig. 3.1(a).

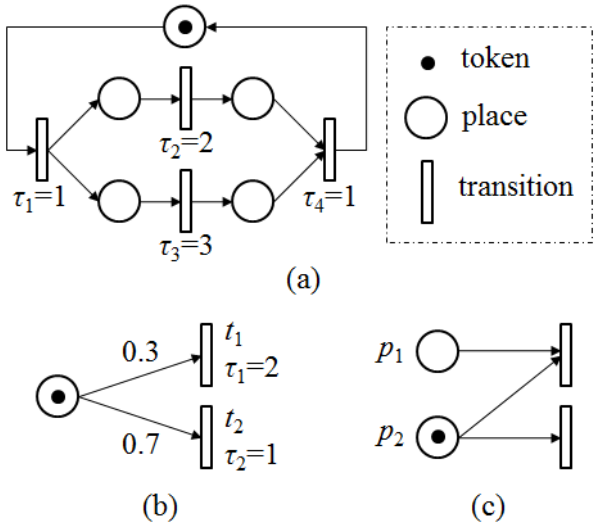


Figure 3.1: (a) A timed marked graph; (b) A stochastic timed free-choice net where the token goes to t_1 (t_2) with a 30% (70%) probability; (c) A net that is not free-choice because $\bullet(p_2\bullet) = \{p_1, p_2\} \neq \{p_2\}$.

Definition 2. *A TMG is a PN with* $w : F \to 1$ *and* $\forall p \in P$, $|\bullet p| = |p \bullet| = 1$, *extended with a transition firing-delay vector* $\tau \in \mathbb{R}^n$, *where* τ_i *denotes the duration time which the* i*-th transition takes for a firing.*

Fig. 3.1(a) also shows a simple TMG which models two concurrent data flows. The *minimum cycle time* [151] of a strongly connected TMG is:

$$c = \max_{k \in K} \{D_k / N_k\} \tag{3.1}$$

where K is the set of cycles in the TMG, D_k is the sum of transition firing-delays in cycle $k \in K$, and N_k is the number of tokens in the cycle. For example, the minimum cycle time of the TMG in Fig. 3.1(a) is $\max\{4/1, 5/1\} = 5$.

Since every place of a TMG has exactly one input and one output transition, TMG cannot model decisions/conflicts in a dynamic system. We then introduce another subclass of PN, free-choice net (FC) [125], to model decisions by non-deterministic token movement from a place.

Definition 3. *An FC is a PN with* $w : F \to 1$ *and* $\forall p \in P$, $|p \bullet| > 1 \implies \bullet(p\bullet) = \{p\}$.

If $|p\bullet| > 1$, the tokens at p can go to any $t \in p\bullet$. The flexibility of FC enables modeling of data dependencies, which commonly exist in stream-computing systems [163].

Definition 4. *A stochastic timed FC is an FC extended with:* (i) *a routing rate associated with each arc* $(p,t) \in F$ *if* $|p\bullet| > 1$ *such that, for each such* p*, the sum of the routing rates equals 1, and* (ii) *a transition firing-delay vector* $\tau \in \mathbb{R}^n$.

Fig. 3.1(b) shows an example of stochastic timed FC. A live and safe stochastic timed FC can be transformed to a behaviorally-equivalent TMG, whose minimum cycle time corresponds to the *average* cycle time of the original FC [31]. Therefore, although for simplicity in this chapter we focus on TMGs, *our algorithmic findings are applicable to live and safe stochastic timed FC after performing a proper graph transformation* [31].

Effective Latency and Throughput. First, the *effective latency* λ of a component is the product of its clock cycle count and clock period. Specifically, the effective latency corresponds to the total elapsed time for a component to read an input token, process the token, and generate an output token. Second, given a system consisting of a set of components, we use a TMG to model the data flow of a system by representing each component with a transition t_i whose firing delay τ_i is equal to its effective latency. The maximum sustainable *effective throughput* θ of the system is defined as the reciprocal of the minimum cycle time of its TMG if this is strongly connected and as the minimum θ among its strongly-connected components, otherwise. In the sequel, we refer to "maximum sustainable effective throughput" simply as "effective throughput". The TMG of Fig. 3.1(a) models a system of 4 components with a θ of $1/5 = 0.2$.

The proposed methodology focuses on the design exploration of an SoC (or an SoC subsystem) that is realized by combining components, often in a pipeline fashion, which are specified with a high-level language. This fits well, for instance, a graphics or multimedia subsystem which is specified as a pipeline of SystemC modules communicating through TLM channels. We express the result of the system-level design exploration as a Pareto front which captures the system implementation trade-offs between the effective throughput θ and the *implementation cost* α.

Ratio Distance. To quantify the quality of the Pareto front in characterizing the system-level design space we adopt the concept of *ratio distance* ($\mathcal{RD}$) [113]. Let the implementation cost and the effective throughput of a system-design point d on the Pareto front be d_α and d_θ, respectively. Intuitively, the ratio distance captures the greater ratio gap between the two points in terms of the objectives α and θ.

Table 3.1: Some Typical High-Level Synthesis Knobs and Their Settings

Knob	Setting
Loop Manipulation	Breaking, Unrolling, Pipelining
State Insertion	Number of States
Memory Configuration	Register Array, Embedded Memory
Clock Cycle Time	Nano-seconds

Definition 5. *Given Pareto-optimal system-design points d and d' with $d'_\alpha \geq d_\alpha$ and $d'_\theta \geq d_\theta$,* $\mathcal{RD}(d, d') = \max\{d'_\alpha/d_\alpha - 1, d'_\theta/d_\theta - 1\}$.

Finally, we give a formal definition of the *Compositional System-Level Design Exploration Problem*, where we refer to synthesis tools as *oracles* and to "running the tools" as *"querying the oracles"*.

Problem 1. *Given a HLS oracle, a target granularity $\delta > 0$, and a TMG model of a system with its components designed in high-level languages, construct a system-level Pareto front w.r.t. α versus θ such that the maximum $\mathcal{RD}$ of two consecutive points on the front is less than δ, by querying the HLS oracle on individual components as few times as possible.*

3.2 An Exploration-and-Reuse Design Methodology

The main difficulties of solving Problem 1 are: *(i)* at the *component* level, to identify the HLS knobs and their settings to query the oracle for optimal component instances, and *(ii)* at the *system* level, to identify critical components to query and to reuse the queries whenever possible for efficient Pareto front construction.

Knob-Settings. Table 3.1 lists a few typical HLS knobs and their settings. A knob may be applied multiple times with different settings, e.g., different loops in a design can be manipulated in different ways. Besides, some settings may involve multiple parameters, e.g., the initiation intervals and the number of stages for loop pipelining can vary. A *knob-setting (KS)* is a particular setting of *all* the knob applications in a design. Hence a KS corresponds to the core content of a HLS script.

Example. To illustrate the application of different KSs on a component, we designed a DCT as a synthesizable SystemC component of an MPEG2 encoder. The DCT consists of nested loops

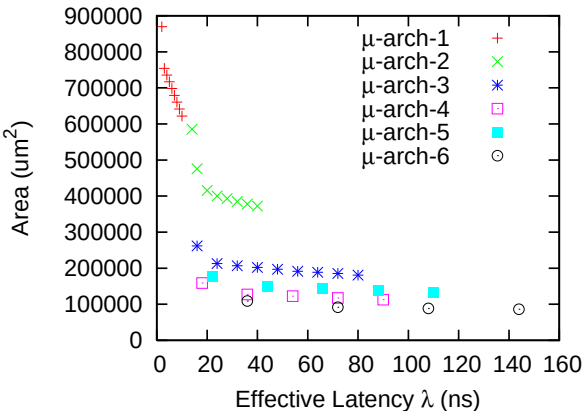


Figure 3.2: HLS-driven implementation for a Discrete Cosine Transform.

of integer multiplications. Fig. 3.2 shows a characterization of the DCT implementation in terms of area vs. effective-latency trade-offs which is obtained by running HLS with the application of loop manipulations to the outermost loop (all inner loops are unrolled). Specifically, μ-arch-1 is the result of completely unrolling the loop to perform DCT in a single clock cycle; μ-arch-2 and μ-arch-3 are the results of partially unrolling the loop to perform DCT in 2 and 4 clock cycles, respectively; and μ-arch-4 to μ-arch-6 are the results of pipelining the loop with different initiation intervals and pipeline stages. For each μ-architecture, we sample its clock period to find the two extreme effective latencies, λ_{min} and λ_{max}, between which the DCT can be synthesized.

In this example, we see *intra*-micro-architecture trade-offs explored by adjusting the clock period. There are also *inter*-micro-architecture trade-offs. For example, μ-arch-1 can achieve shorter effective latencies by parallelizing all the multiplications at the cost of greater area occupation due to fully-duplicated multipliers. To distinguish the latter trade-offs, we denote two KSs as *distinct* when they differ by at least one setting among all the knobs, except the technology-dependent knobs such as the clock period. For instance, there are 6 distinct KSs in Fig. 3.2.

The Proposed Methodology. Fig. 3.3 illustrates the two main steps of our approach to solve Problem 1.

First, feasible KSs of the component are determined based on the *component designer*'s knowledge. Example *infeasible* KSs include (but not restricted to): *(i)* arbitrary state insertion that causes clock-cycle-count violations for real-time applications, *(ii)* loop unrolling that requires single-cycle parallel data access that cannot be supported by limited memory read/write ports, and *(iii)* not

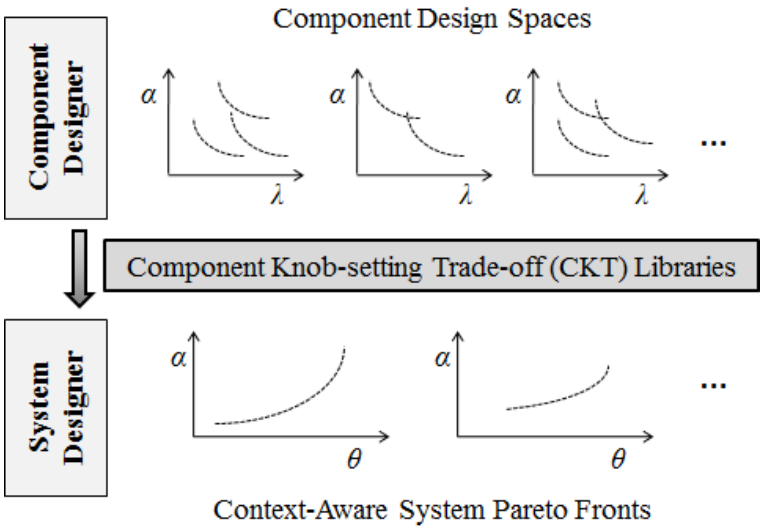


Figure 3.3: The proposed design methodology.

inlining functions that involve both sequential and combinational paths. After determining the feasible KSs, the component designer should also find the extreme-λ: this step requires only a one-time effort and its results can be reused by different system designers.

Then, at the time of component integration, the *system designer* has the freedom to pre-select a subset of the feasible KSs for each component based on the specific context of the given system design. Hence, different pre-selection schemes will result in *context-aware system Pareto fronts* as shown in Fig. 3.3[1]. For example, in addition to typical costs like area and power, which have been characterized in the component libraries, if system designers want to avoid the higher mask/testing cost and/or limited yield due to the deployment of embedded memories [122], then they can set a "don't use" attribute to the knob-settings that involve embedded memories, before running HLS. Since the KSs involving register arrays generally yield component instances with greater area than those involving embedded memories, a *reusable* component library should not be limited to keep only area-optimal KSs, which may be sub-optimal in terms of other design costs such as for mask generation and circuit testing in the example involving embedded memories. Consequently, in order to keep all the feasible KSs for any possible given system-design context while maintaining a concise library format, we propose the notion of *Component Knob-setting Trade-off (CKT) Library*.

[1]In Fig. 3.3, each θ of the system is actually calculated from the λ's of the components (one of each) as discussed in Section 3.1.

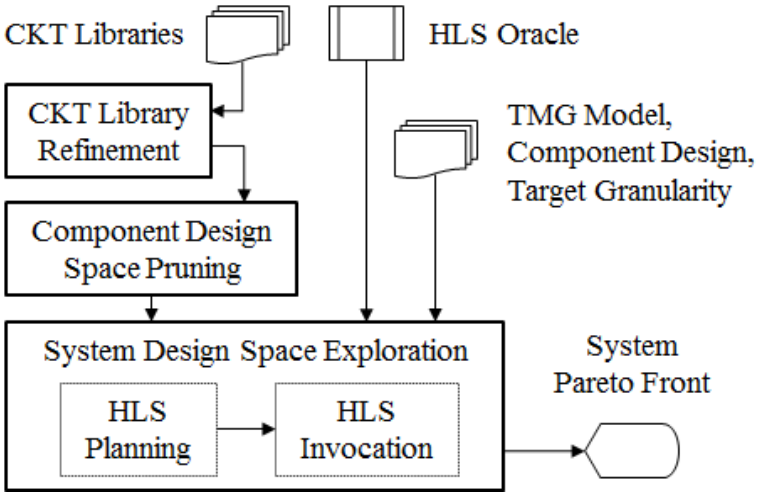


Figure 3.4: Our algorithm flowchart in the context of system design.

A CKT library, which is provided by component designers, stores all the *distinct* feasible KSs of a component: each KS indicates its fixed clock-cycle count, its extreme implementation costs, α_{max} and α_{min}, and its extreme effective latencies, λ_{min} and λ_{max}. The average or the worst-case clock-cycle count can be used when the clock-cycle count is not deterministic. The fixed clock-cycle count allows us to explore the intra-knob-setting α vs. λ trade-off space by adjusting the clock period to synthesize the component. In addition, the extreme values of α and λ outline the sub-design space of a KS.

To assist the system designer, we propose the algorithms charted in Fig. 3.4 to explore the design space given the CKT libraries. In the context of a given system design, for each component the CKT library can optionally be refined by pre-selecting KSs as desired and/or re-characterizing the extreme α and λ if a different technology is considered (*CKT library refinement* in Fig. 3.4). For each refined CKT library, we then apply *component design space pruning* to find out KS candidates for deriving optimal component instances (Section 3.3). Next, for *system design space exploration*, we propose a HLS-planning algorithm to identify critical oracle queries before actually invoking them (Section 3.4).

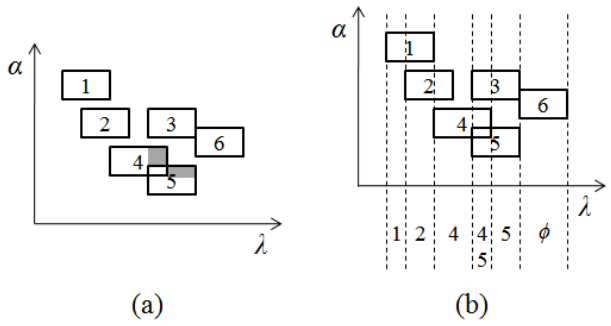


Figure 3.5: (a) An instance of Problem 2. (b) Dashed lines identify six non-overlapping λ intervals; their optimal knob-setting candidates for deriving Pareto-optimal instances are shown below the λ axis.

3.3 Component Design-Space Pruning

For each refined CKT library with s distinct KSs, we consider the following problem, where a KS *dominates* another KS iff the former can achieve a lower implementation cost α w.r.t. a fixed effective latency λ, or the former can achieve both a lower α and a shorter λ.

Problem 2. *Given s distinct knob-settings, each of which yields a component sub-design space bounded by $(\lambda^i_{min}, \alpha^i_{max})$ and $(\lambda^i_{max}, \alpha^i_{min})$, $i \in \{1, 2, \ldots, s\}$, find the ordered list of non-overlapping λ intervals, each of which indicates its non-dominated knob-settings.*

Fig. 3.5(a) shows an instance of Problem 2 with six distinct KSs, each of which yields a design-space *rectangle* bounded by upper-left and lower-right extreme points. In a certain λ interval, KS 3 is dominated by both KS 4 and KS 5 because they can always yield lower-cost instances than KS 3. A cross-λ-interval example is that KS 6 is dominated by KS 5. Obviously, only non-dominated KSs should be kept as candidates for deriving Pareto-optimal component instances. Our goal is to find the ordered list of non-overlapping λ intervals with their optimal KS candidates, as shown in Fig. 3.5(b).

Theorem 1. *Problem 2 requires an $\Omega(s \log s)$-time algorithm to compute exact solutions.*

Proof. We first construct a problem-reduction procedure to reduce the positive-integer sorting problem with an input size s to Problem 2. Since the sorting problem is known to hold an $\Omega(s \log s)$ time complexity [44], Theorem 1 can be proved if we show that *(i)* the reduction procedure with

any algorithm that solves Problem 2 can solve the sorting problem and *(ii)* the reduction procedure takes $o(s \log s)$ time.

- **Problem reduction:** (**Step 1**) Given s distinct positive integers $\{n_1, n_2, \ldots, n_s\}$, for each n_i create a unique KS$_i$ with $(\lambda^i_{min}, \lambda^i_{max}, \alpha^i_{min}, \alpha^i_{max}) = (n_i - 0.5, n_i, (n_i - 0.5)^2, n_i^2)$. Without loss of generality, assume $n_j < n_i < n_k$ and there does not exist n_x such that $n_j < n_x < n_i$ or $n_i < n_x < n_k$ (*Assumption 1*). Fig. 3.6 visualizes three such KSs for n_j, n_i, and n_k in the α vs. λ space. (**Step 2**) For each KS$_i$ whose $(\lambda^i_{min}, \lambda^i_{max}) = (n_i - 0.5, n_i)$, convert the λ's to $(\lambda^i_{min}, \lambda^i_{max}) = (-n_i, -(n_i - 0.5))$ (i.e. reflect each KS across the α axis). (**Step 3**) Feed the converted KSs to Problem 2. The output of Problem 2 is an ordered list of $s + (s-1) = 2s - 1$ λ intervals, such that each interval I_i has either one or zero non-dominating KS$_i$ with $(\lambda^i_{min}, \lambda^i_{max}, \alpha^i_{min}, \alpha^i_{max}) = (-n_i, -(n_i - 0.5), (n_i - 0.5)^2, n_i^2)$. (**Step 4**) Return a list of s integers $-\lambda^i_{min}$ for $i = 1, 3, 5, \ldots, 2s - 1$. The list is the sorted list of the s input integers.

- **Claim 1:** *the problem reduction procedure solves the sorting problem.* First, by Assumption 1 in Step 1, the three KSs shown in Fig. 3.6 instantiate five *non-overlapping* λ intervals λ_1–λ_5, such that λ_1, λ_3, and λ_3 has *one* KS each, while λ_2 and λ_4 has *none* each. Second, by Assumption 1 again, the three KSs instantiate five similar non-overlapping α intervals α_1–α_5. Obviously, for the converted KSs, which are reflected across the α axis, each is a non-dominating KS in the α vs. λ space. Moreover, the smaller a converted KS's λ_{min} and λ_{max}, the greater its α_{min} and α_{max}. Hence, the returned list of the above reduction procedure is a sorted list of the input integers.

- **Claim 2:** *the problem reduction procedure takes $o(s \log s)$ time.* Steps 1, 2, and 4 in the reduction procedure each takes $O(s)$ time, so the claim is true.

By showing Claims 1 and 2, we proved Theorem 1. □

Given s KSs with s pairs of extreme λ's, $(\lambda^i_{min}, \lambda^i_{max})$, $i \in \{1, 2, \ldots, s\}$, by regarding these s pairs as s *horizontal* line segments with $2s$ end points, whose ordered x-coordinates are $x_1 < x_2 < \cdots < x_{2s}$, we have the following $2s + 1$ *atomic intervals*: $[-\infty, x_1], [x_1, x_2], \ldots, [x_{2s-1}, x_{2s}], [x_{2s}, \infty]$. Fig. 3.7(a) shows an example of 3 KSs and 7 atomic intervals: the middle horizontal line segments

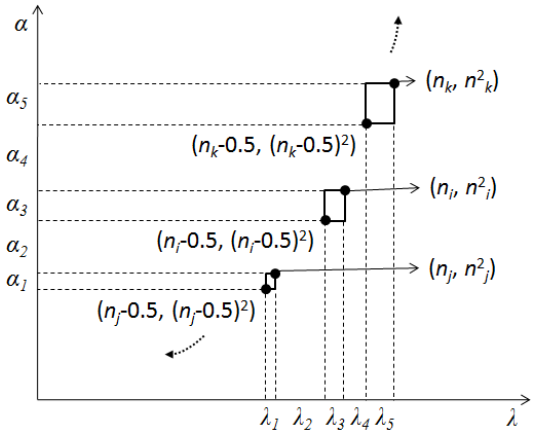


Figure 3.6: Visualization for the proof of Theorem 1.

are projected from the upper rectangles (i.e. the sub-design spaces of the 3 KSs), and the bottom atomic intervals are derived from the middle line segments.

Next, we describe a tree-based data structure, which is used to store the input KSs as line segments. A *segment tree* [51] is a balanced binary tree, such that: *(i)* each leaf node represents an atomic interval, *(ii)* each non-leaf node u, with children v and w, represents an interval $I_u = I_v \cup I_w$, i.e., u represents a coarser interval partition of its children's (see Fig. 3.7(b), where the common x-coordinate x_j of I_v and I_w is stored in node u), and *(iii)* an input line segment can be split into several parts, each of which is stored in a tree node as close to the root as possible. Fig. 3.7(c) shows an example segment tree, where the atomic intervals and the input line segments are copied from Fig. 3.7(a) and shown below the tree; the number right above a tree node indicates the line segments stored in that node.

To retrieve the line segments that enclose an x-coordinate x, we can start from the root and proceed by binary search (comparing x with the x-coordinate stored in the non-leaf nodes) down to the leaf node which encloses x. For instance, in Fig. 3.7(a), to retrieve the line segments that enclose the atomic interval $[x_4, x_5]$, we follow the dashed trace of Fig. 3.7(c) and obtain Segments 2 and 3. Since a stored line segment corresponds to a KS, the tree search procedure allows to retrieve all the KSs whose $[\lambda_{min}, \lambda_{max}]$ encloses any given atomic interval.

For each atomic interval which is enclosed by r KSs, we present the following procedure to identify its non-dominated KSs. First, these r KSs have r pairs of extreme α's. Similarly, we regard these r pairs as r *vertical* line segments with $2r$ end points, whose ordered y-coordinates

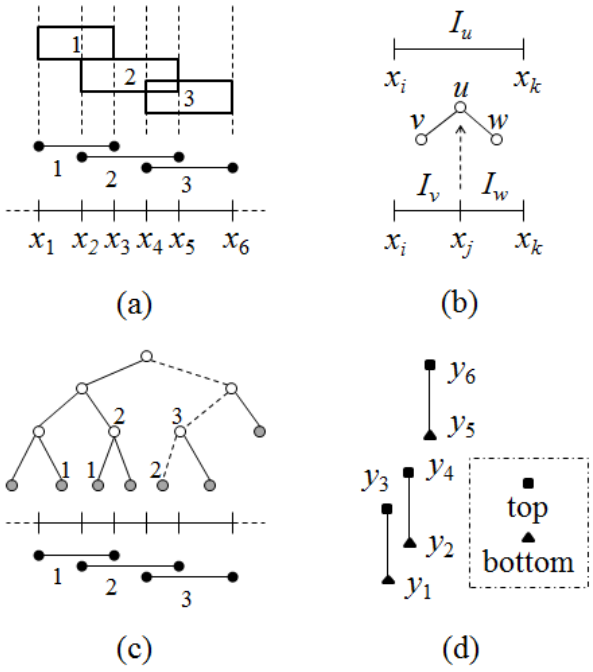


Figure 3.7: Illustration of Algorithm 1.

are $y_1 \leq y_2 \leq \cdots \leq y_{2r}$. We assign two end-point types to each line segment, whose end-point y-coordinates are $y_i < y_j$: *(i)* "bottom" for the y_i-end and *(ii)* "top" for the y_j-end (Fig. 3.7(d) shows an example of three line segments). Trivially, the y_1-end is a bottom end and the first/lowest line segment is non-dominated. Iterating through y_i for $i = 2, 3, \ldots, r$, we check the end-point type of y_i: if y_i is a bottom end, the y_i-corresponding line segment must overlap the previous one(s), and therefore is non-dominated; otherwise, we stop iterating. For example, in Fig. 3.7(d), the procedure stops after knowing that y_3 is a top end, and thus identifies $[y_1, y_3]$ and $[y_2, y_4]$ as non-dominated line segments.

Algorithm 1 gives the pseudocode of our complete procedure for Problem 2. Lines 1–2 setup a segment tree for the input KSs. In the main loop (lines 4–15), we iterate through atomic intervals from left to right. For each atomic interval, enclosing KSs are retrieved (line 5), and non-dominated KSs are identified (lines 7–15). Note that lines 3, 6, and 16 prune the KSs which are dominated *across* atomic intervals or dominated by a minimum α (represented by the variable y_{min} in the pseudocode). For example, in Fig. 3.5(a), KS 6 is pruned because it is dominated by KS 5. Besides, the shaded region of KS 4 can be pruned because of the upper-left extreme point of KS 5, so can

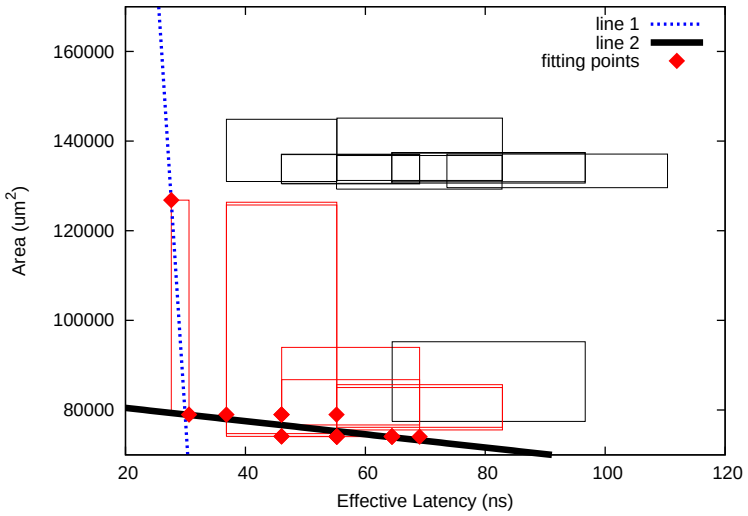


Figure 3.8: The pruning and piecewise-linear-fitting results of a double-precision floating-point divider (DIV) component.

the shaded region of KS 5 because of the lower-right extreme point of KS 4.

A real example of the execution of Algorithm 1 on a double-precision floating-point divider (DIV) component is shown in Fig. 3.8. In this example, we explored 16 alternative micro-architectures (i.e. 16 KSs). The seven non-dominated and nine dominated KSs are represented by red and black rectangles in the figure, respectively. There are 15 atomic intervals, each of which has 0–6 non-dominated KSs with an average number of 1.3. That is, on average in each interval, we only need to consider around 8% (1.3/16) of all the KSs for oracle query. This shows the effectiveness of the algorithm for component design-space pruning .

We remark here that Algorithm 1 outputs at most $2s - 1$ non-overlapping (atomic) intervals, some of which can be further merged if they contain an identical set of non-dominated KSs. We skip this merge optimization in our pseudocode because the number of output intervals is already economic in practice ($O(s)$). Most importantly, even without the merge, the actual synthesis jobs are not redundant because we cache the required KSs before running the synthesis with them (see Section 3.4).

Theorem 2. *The running time of Algorithm 1 is $O(s \log s + sr \log r)$, where s is the number of distinct knob-settings and r is the max number of enclosing knob-settings of an atomic interval.*

Algorithm 1 Component Design Space Pruning

Input: s distinct knob-settings with extreme λ's and α's

Output: atomic λ intervals w/ non-dominated knob-settings

1: $I \equiv [-\infty, x_1], \ldots, [x_{2s}, \infty] \leftarrow$ sort knob-settings by λ
2: $\Upsilon \leftarrow$ build a segment tree for I
3: $y_{min} \leftarrow \infty$
4: **for** $i = 1 \rightarrow 2s - 1$ **do**
5: $S \leftarrow$ search Υ by $x = (x_i + x_{i+1})/2$
6: prune S by y_{min}
7: $y_1 \leq \cdots \leq y_{2r} \leftarrow$ sort S by α
8: **for** $j = 1 \rightarrow r$ **do**
9: **if** the y_j-end is a bottom end **then**
10: output the y_j-corresponding knob-setting
11: **else**
12: stop this loop
13: **end if**
14: $j \leftarrow j + 1$
15: **end for**
16: update y_{min}
17: $i \leftarrow i + 1$
18: **end for**

Proof. The sorting in Line 1 takes $O(s \log s)$ time. Line 2 also requires $O(s \log s)$ time because the construction of a segment tree for s line segments is $O(s \log s)$ [51]. Therefore, the initialization of the algorithm (Lines 1-3) takes $O(s \log s)$ time.

In each atomic interval, the search of its non-dominated KSs (Line 5) takes $O(\log s + r)$ time ($O(\log s)$ for segment-tree traversal [51] and $O(r)$ for bookkeeping the KSs). The pruning and sorting in Lines 6 and 7 takes $O(r)$ and $O(r \log r)$ time, respectively. The whole nested "for" loop (Lines 8–15) requires only linear time $O(r)$. Overall, the body of the outer "for" loop (Lines 5–17) takes $O(\log s + r \log r)$ time.

Since the outer "for" loop (Line 4) needs $O(s)$ iterations, the major steps of the algorithm (Lines

4–18) takes $O(s \log s + sr \log r)$, which dominates the algorithm initialization time and thereby the whole algorithm running time. □

Note that since $r \leq s$, in the worst case the running time of Algorithm 1 is $O(s^2 \log s)$, whereas in the case of $r \log r = O(\log s)$ the algorithm runs in $O(s \log s)$ time, which achieves the theoretical lower bound indicated by Theorem 1. Hence, Algorithm 1 is not only exact but also *optimal* in the latter case.

3.4 System Design-Space Exploration

Given a strongly connected TMG, to compute its minimum cycle time (Equation (3.1)), Maggot proposed the linear program [117]:

$$\begin{aligned} \min \quad & c \\ \text{s.t.} \quad & A\sigma + M_0 c \geq \tau^- \\ & c \geq 0, \end{aligned} \tag{3.2}$$

where c is the minimum cycle time, $\sigma \in \mathbb{R}^n$ is the transition-firing initiation-time vector, $M_0 \in \mathbb{N}^m$ is the initial marking, $\tau^- \in \mathbb{R}^m$ is the input-transition firing-delay vector (i.e., τ_i^- equals the firing-delay τ_k of the $t_k \in \bullet p_i$), and $A \in \{-1, 0, 1\}^{m \times n}$ is the incidence matrix such that

$$A[i,j] = \begin{cases} 1 & \text{if } t_j \text{ is the output transition of } p_i, \\ -1 & \text{if } t_j \text{ is the input transition of } p_i, \\ 0 & \text{otherwise.} \end{cases}$$

In this linear program, c and σ are decision variables.

We give a running example of the linear program for the TMG shown in Fig. 3.1(a) as follows.

$$\begin{aligned} \min \quad & c \\ \text{s.t.} \quad & \sigma_2 - \sigma_1 \geq 1 \\ & \sigma_3 - \sigma_1 \geq 1 \\ & \sigma_4 - \sigma_2 \geq 2 \\ & \sigma_4 - \sigma_3 \geq 3 \\ & \sigma_1 - \sigma_4 + 1 \times c \geq 1 \\ & c \geq 0, \end{aligned} \tag{3.3}$$

The minimum cycle time c to Equation (3.3) is 5 and an optimal σ is $[0, 1, 1, 4]$. Therefore, the effective throughput of this example is $1/5 = 0.2$.

Based on Equation (3.2), we propose the following effective-throughput-θ-constrained cost-minimization linear program (recall that $c = 1/\theta$):

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^{n} f_i(\tau_i) \\ \text{s.t.} \quad & A\sigma + M_0/\theta \geq \tau^- \\ & \tau^-_{min} \leq \tau^- \leq \tau^-_{max}, \end{aligned} \tag{3.4}$$

where the function f_i yields the ith component's implementation cost given the firing-delay τ_i of the transition t_i (i.e., the λ of the t_i-corresponding component equals τ_i), and τ^-_{min} and τ^-_{max} are given according to the extreme λ's of the components. The decision variable is the vector τ, i.e. the λ requirements of the components.

Equation (3.4) works for strongly-connected TMGs. For acyclic TMGs (such as the example TMG in Fig. 3.1(a) if one assumes to remove the feedback path from transition t_4 to transition t_1), we propose the following linear program:

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^{n} \alpha_i \\ \text{s.t.} \quad & \tau_i \leq 1/\theta, \text{ for } i = 1, 2, \ldots, n \\ & f_i^{-1}(\alpha_i) \leq \tau_i, \text{ for } i = 1, 2, \ldots, n \\ & \tau^-_{min} \leq \tau^- \leq \tau^-_{max}, \end{aligned} \tag{3.5}$$

where α_i is the implementation-cost variable of the ith component, and $f_i^{-1}()$ is the inverse function of the function $f_i()$ in Equation (3.4).

In general, the cost functions f_i's in Equation (3.4) are unknown a-priori. However, based on our experimental results and recent publications [92; 104], we observe that these cost functions generally exhibit *convexity* [22]. Therefore, for each component, given its pruned design space (discussed in Section 3.3), we propose to approximate its cost function f_i with *convex piecewise-linear* functions. Fig. 3.9 shows an illustrative approximation using a convex piecewise-linear function[2]

$$\bar{f}(\lambda) = \max_{j=1,2}(a_j\lambda + b_j),$$

where a_j and b_j are coefficients. A real fitting example on the DIV component is also shown in Fig. 3.8, where the red fitting points are derived by the y_{min} variable in Algorithm 1 for each

[2]Generally, the larger number of linear functions are used, the more accurate the approximation can be.

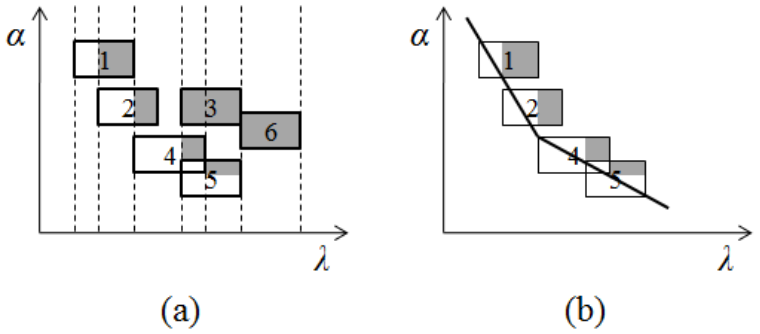


Figure 3.9: (a) Pruned component design space: shaded space is sub-optimal. (b) Pruned-space approximation by a convex piecewise-linear function.

atomic interval. Note that fitting a given set of data points using a convex piecewise-linear function while minimizing its total squared-fitting errors can be solved optimally in polynomial time with quadratic programming [22]. Moreover, the derivation of the inverse function of a piecewise-linear function is straightforward.

Due to the design space pruning procedure presented in Section 3.3, the fitting function $\bar{f}$ is not only convex piecewise-linear but also *monotonically decreasing*. If we replace the f_i's in Equation (3.4) with the fitting functions $\bar{f}_i$'s, the objective function $\sum_{i=1}^{n} \bar{f}_i$ will also be convex piecewise-linear and monotonically decreasing. A minimization linear program with its objective function being convex piecewise-linear can be transformed to a standard linear program [22] and, therefore, is polynomial-time solvable. Besides, the monotonically-decreasing objective function guarantees that all the components not in the performance-critical cycle of the TMG will be run at their maximum possible λ values to minimize their implementation costs. As a result, our linear program is not only computationally tractable, but can also minimize the number of distinct non-critical component instances across different system throughput θ requirements.

Finally, Algorithm 2 gives the pseudocode of our complete procedure for Problem 1. The first loop (lines 1–4), for each component, prunes its design space by using Algorithm 1 and approximates its implementation cost function. The second loop (lines 8–11) iterates through a geometric progression of θ requirements with a ratio $1 + \delta$ (δ is the target granularity of Problem 1; see Fig. 3.10(a) for illustration). For each θ requirement, after solving the proposed linear program (line 9), we cache the optimal λ requirements of the components in a hash table Π, which is initial-

Algorithm 2 System Design Space Exploration

Input: a HLS oracle, a target granularity δ, and a TMG model of a system with a set of components

Output: α vs. θ trade-off

1: **for all** components **do**
2: $\Sigma_i \leftarrow$ run Algorithm 1 for the current component
3: $\bar{f}_i \leftarrow$ run convex piecewise-linear fitting for Σ_i
4: **end for**
5: $\Pi \leftarrow$ an empty hash table for optimal λ requirements
6: $\theta \leftarrow$ min throughput w/ all components at max λ's
7: $\hat{\theta} \leftarrow$ max throughput w/ all components at min λ's
8: **while** $\theta \leq \hat{\theta}$ **do**
9: $\Pi \leftarrow$ solve Equation (3.4) with $\bar{f}_i$
10: $\theta \leftarrow \theta \times (1 + \delta)$
11: **end while**
12: $\Pi \leftarrow$ solve Equation (3.6) whenever necessary
13: query the oracle according to Π and Σ

ized in line 5. After the loop, to see if the current maximum $\mathcal{RD}$ is less than or equal to δ, we check if there exists any $\theta_{i+1} = \theta_i \times (1+\delta)$ whose corresponding α_{i+1} and α_i differ by $\alpha_{i+1}/\alpha_i - 1 > \delta$ (see Fig. 3.10(b)). If so, we explore between α_i and α_{i+1} (line 12) by a similar geometric progression of α (e.g., the $(1 + \delta)^j \alpha_i$, for $j = 1$ and 2, in Fig. 3.10(b)) and solve the following α-constrained θ-maximization linear program[3]:

$$\begin{aligned} \max \quad & \theta \\ \text{s.t.} \quad & A\sigma + M_0/\theta \geq \tau^- \\ & \textstyle\sum_{i=1}^{n} \bar{f}_i(\tau_i) \leq \alpha \\ & \tau^-_{min} \leq \tau^- \leq \tau^-_{max}, \end{aligned} \tag{3.6}$$

where θ and τ are decision variables of a total size equal to $1 + n$ (n is the number of components).

[3] The current forms of Equations (3.6) and (3.7) are non-linear. However, their equivalent c-minimization programs are linear because the minimum cycle time c is equal to $1/\theta$.

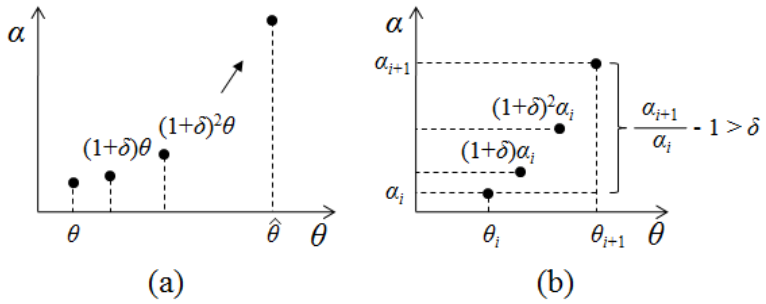


Figure 3.10: Illustration of Algorithm 2.

In the final step of Algorithm 2 (line 13), we invoke all the oracle queries on components, according to the λ requirements cached in Π and the KS candidates found by Algorithm 1.

On the other hand, for acyclic TMGs, the α-constrained θ-maximization linear program is modified as follows:

$$\begin{aligned} \max \quad & \theta \\ \text{s.t.} \quad & \tau_i \leq 1/\theta, \text{ for } i = 1, 2, \ldots, n \\ & \bar{f}^{-1}(\alpha_i) \leq \tau_i, \text{ for } i = 1, 2, \ldots, n \\ & \textstyle\sum_{i=1}^{n} \alpha_i \leq \alpha \\ & \tau^-_{min} \leq \tau^- \leq \tau^-_{max}. \end{aligned} \tag{3.7}$$

We remark that in Algorithm 2: *(i)* we do not invoke oracle queries until we have identified all the non-redundant λ requirements, thereby minimizing the number of actual queries, whose runtime dominates the whole exploration procedure, and *(ii)* we can parallelize all the linear-program solving (lines 9 and 12, by pre-computing the geometric progressions of θ and α), and most importantly, the time-consuming oracle queries (line 13).

3.5 Experimental Results

We implemented Algorithms 1 and 2 as part of a tool called SPEED (Synthesis Planning for Exploring ESL Design). SPEED was tested on four benchmark designs with various data flows, whose TMG models are shown in Fig. 3.11: (a) a JPEG decoding system (JPEG) including three components with a pipelined data stream, (b) a Reed-Solomon decoding system (RS) including five components with concurrent data flows, (c) a double-precision floating-point sine approximation system (DFSIN) including five components with concurrent data flows, feedback paths, and mul-

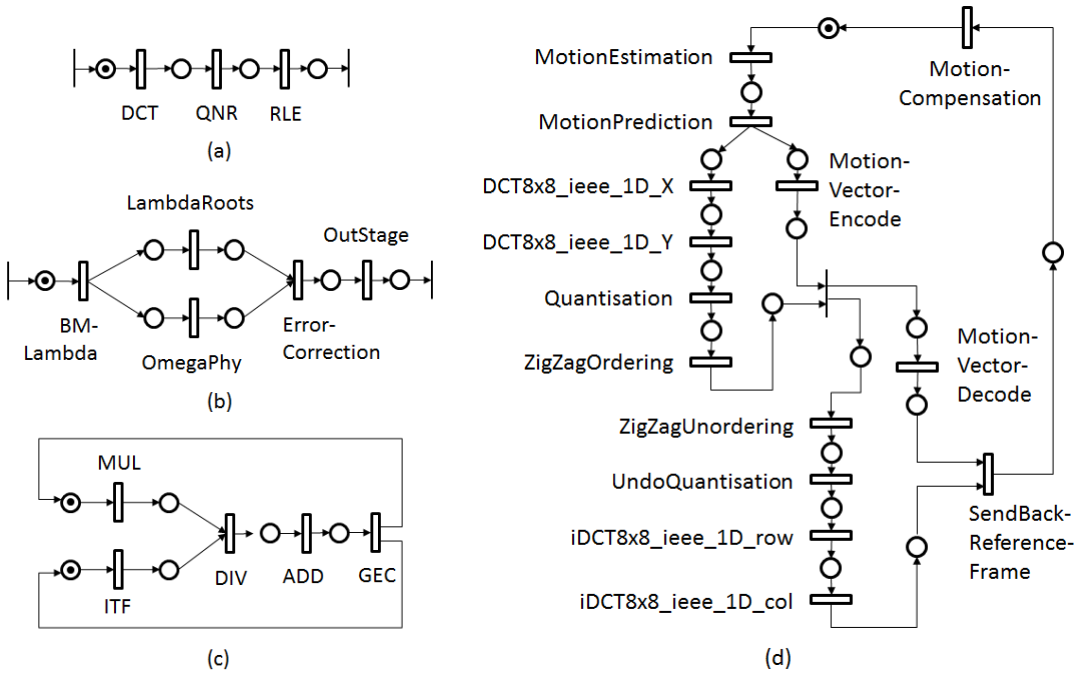


Figure 3.11: The TMG models of the four benchmark systems (a) JPEG, (b) RS, (c) DFSIN, and (d) MPEG2. The transitions represented by solid lines are for modeling the token generation, receiving, or synchronization, whose transition times are small constants and are independent of the system exploration (so they are ignored when computing the effective throughputs of the systems).

tiple places with initial tokens, and (d) an MPEG encoding/decoding system (MPEG2) including fourteen components with multiple concurrent data flows and a feedback path. All the systems were designed in SystemC and synthesized with a commercial HLS tool for an industrial $45nm$ technology.

Over all the system components, we characterized on average 12.2 distinct KSs per component to build the CKT libraries. Note that these CKT libraries can thereafter be reused in system designs other than the four benchmark systems. In the experimental system-design context, we simply kept all the distinct KSs as implementation alternatives. After applying SPEED to prune these CKT libraries, we got on average 2.5 non-dominated KSs per atomic interval. This result shows that in order to derive an optimal component instance given its λ requirement, we can prune on average 80% $((12.2-2.5)/12.2)$ of HLS invocations, instead of synthesizing with all the available KSs.

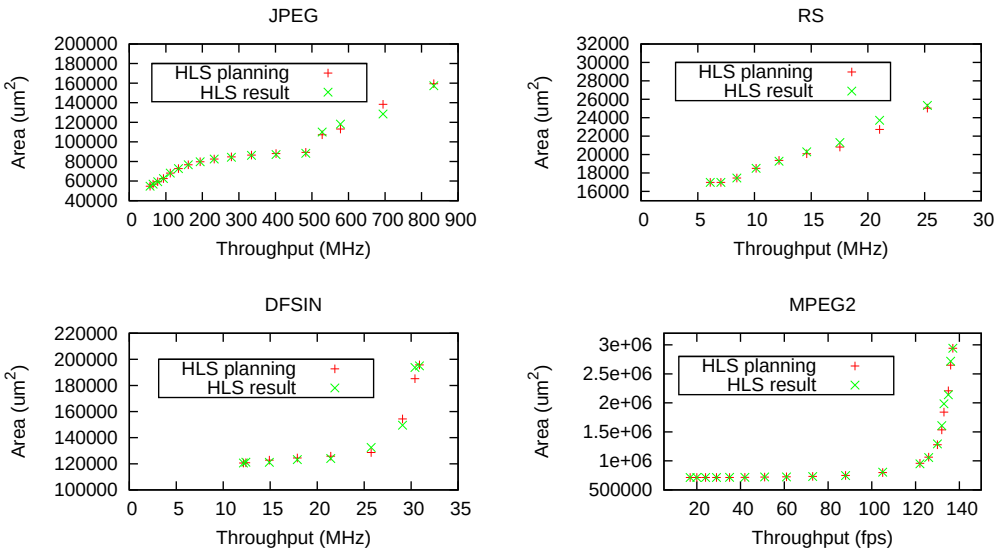


Figure 3.12: System design-space-exploration results for JPEG, RS, DFSIN, and MPEG2 in terms of area vs. throughput trade-off, with a target granularity $\delta = 0.2$.

We then applied SPEED with a target granularity $\delta = 0.2$ to explore the area vs. throughput trade-off of the four benchmark systems. For each pruned component design space, we used 1–3 linear functions to compose a convex piecewise-linear approximation. The exploration results are shown in Fig. 3.12, where the "HLS planning" points result from solving Equations (3.4) and (3.6) (Equations (3.5) and (3.7)) on DFSIN and MPEG2 (on JPEG and RS), while the "HLS result" points are obtained by invoking actual HLS.

To quantify the mismatch between an "HLS planning" point d_1 and an "HLS result" point d_2 w.r.t. a specific system throughput, we apply the following area-mismatch ($\mathcal{AM}$) metric:

$$\mathcal{AM}(d_1, d_2) = |d_{1\alpha} - d_{2\alpha}|/d_{2\alpha},$$

where $d_{1\alpha}$ ($d_{2\alpha}$) is the area of d_1 (d_2). The explorations in Fig. 3.12 yield 17, 9, 9, and 19 system-design points for JPEG, RS, DFSIN, and MPEG2, respectively, with an average $\mathcal{AM}$ of 1%–2% and a max $\mathcal{AM}$ of 4%–7%. Over the four benchmarks, good 78%–89% system-design points have an $\mathcal{AM}$ below 3%. This result shows that the component-design approximation based on piecewise-linear functions delivers a good first-order approximation for estimating the system Pareto front.

Fig. 3.13 shows the component query distributions to achieve the exploration results in Fig. 3.12.[4]

[4]Note that an oracle query may invoke the HLS tool more than once, depending on the number of non-dominated

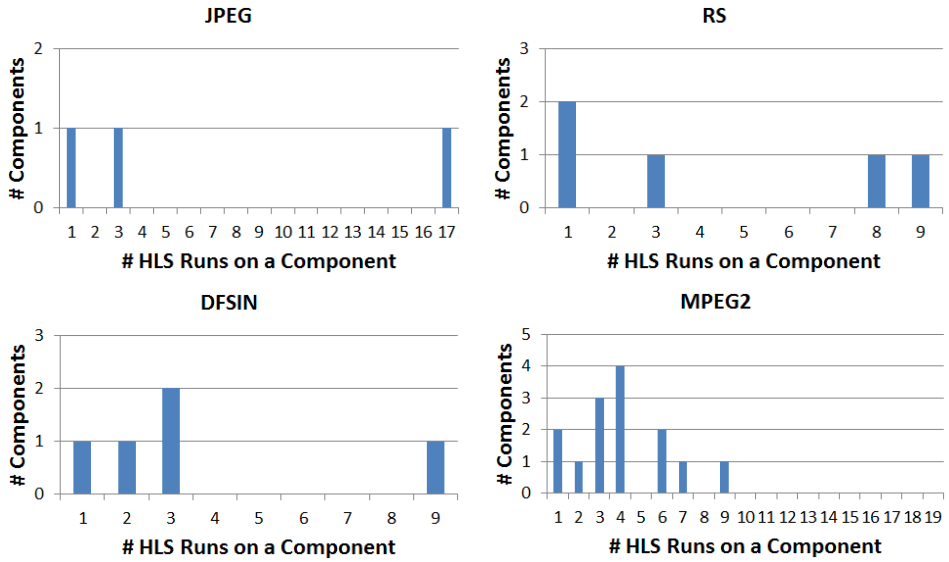


Figure 3.13: Component query distributions for the explorations in Fig. 3.12.

For each benchmark system, the maximum number of queries on a component is equal to the number of system points, i.e. to query each component once for each system throughput requirement. However, SPEED can identify critical queries by solving Equations (3.4)–(3.7), cache the effective-latency requirements, and reuse them. Hence, as shown in Fig. 3.13, JPEG, RS, and DFSIN have each one to two critical components that get queried most frequently (on the right ends of the query distributions), while the non-critical components only get queried far less than the number of system points by 66%–82%. In total, 21, 22, and 18 component queries were made for JPEG, RS, and DFSIN, respectively. In comparison with exhaustive queries (requiring "# of system points × # of components" queries), SPEED achieves a 2.0×–2.5× query efficiency. Moreover, for the MPEG2 system, 57 component queries were made in total (a 4.7× query efficiency), including ten non-critical components each getting queried less than five times and four critical ones each getting queried six–nine times. We show the MPEG2 component queries in the area vs. effective-latency space in Fig. 3.14, where the critical-component results are placed in the top row. The critical components are MotionEstimation, Quantization, DCT8x8_ieee_ID_X, and MotionCompensation, which are *automatically* discovered by SPEED.

knob-settings. Therefore, we use the terms "oracle query" and "synthesis invocation" separately: a query may invoke more than one synthesis jobs.

In terms of the aggregated HLS runtime, the explorations in Fig. 3.12 require 5-, 22-, 40-, and 35-hour *sequential* HLS-tool invocations for JPEG, RS, DFSIN, and MPEG2, respectively, whereas the sequential exhaustive synthesis requires 7, 27, 54, and 124 hours, respectively. That is, SPEED achieves a 1.2×–3.5× speed-up. The speed-ups for JPEG, RS, and DFSIN are smaller than that for the MPEG2 because the HLS runtime of the critical components of the first three systems is much longer than the runtime of the non-critical counterparts. As the critical components get synthesized more often, the speed-up diminishes for those three systems, although their query efficiencies are still more than 2× than the efficiency of the exhaustive query. In fact, compared with the other steps in Algorithm 2 that run in seconds, the HLS tool invocation dominates the whole exploration runtime. Considering that the maximum runtime of a single component synthesis over the four systems is around 2.5 hours, the power of being able to parallelize all the synthesis tasks is significant. This ability is another advantage of SPEED.

We summarized the results for $\delta = 0.2$ in Table 3.2, where the results for $\delta \in \{0.15, 0.1\}$ are also listed. As δ decreases from 0.2 to 0.1, the number of explored system-design points increases as expected. In terms of the approximation accuracy, the average $\mathcal{AM}$ remains at 1%–2% as δ varies. Although the maximum $\mathcal{AM}$ may increase slightly as δ decreases, there are constantly 88% points with $\mathcal{AM} \leq 3\%$, thus showing the general approximation robustness even though we improve the granularity δ. Furthermore, the total number of component queries also grows as we decrease δ in order to explore more system-design points. Meanwhile, as δ decreases, the query efficiency compared with exhaustive queries improves for all four systems, implying that critical queries are reused even more effectively among system-design points. The same trend can also be found w.r.t. the sequential HLS tool runtime which increases as δ decreases; meanwhile, the speed-up over exhaustive synthesis goes up as well. Both trends suggest that SPEED scale well with the target granularity δ.

3.6 Remarks

SPEED can effectively *(i)* prune sub-optimal component-implementation alternatives by accurate design-space approximation and *(ii)* reuse and parallelize component-HLS plans for efficient system-level exploration. In addition to these advantages, SPEED's static performance-analysis models

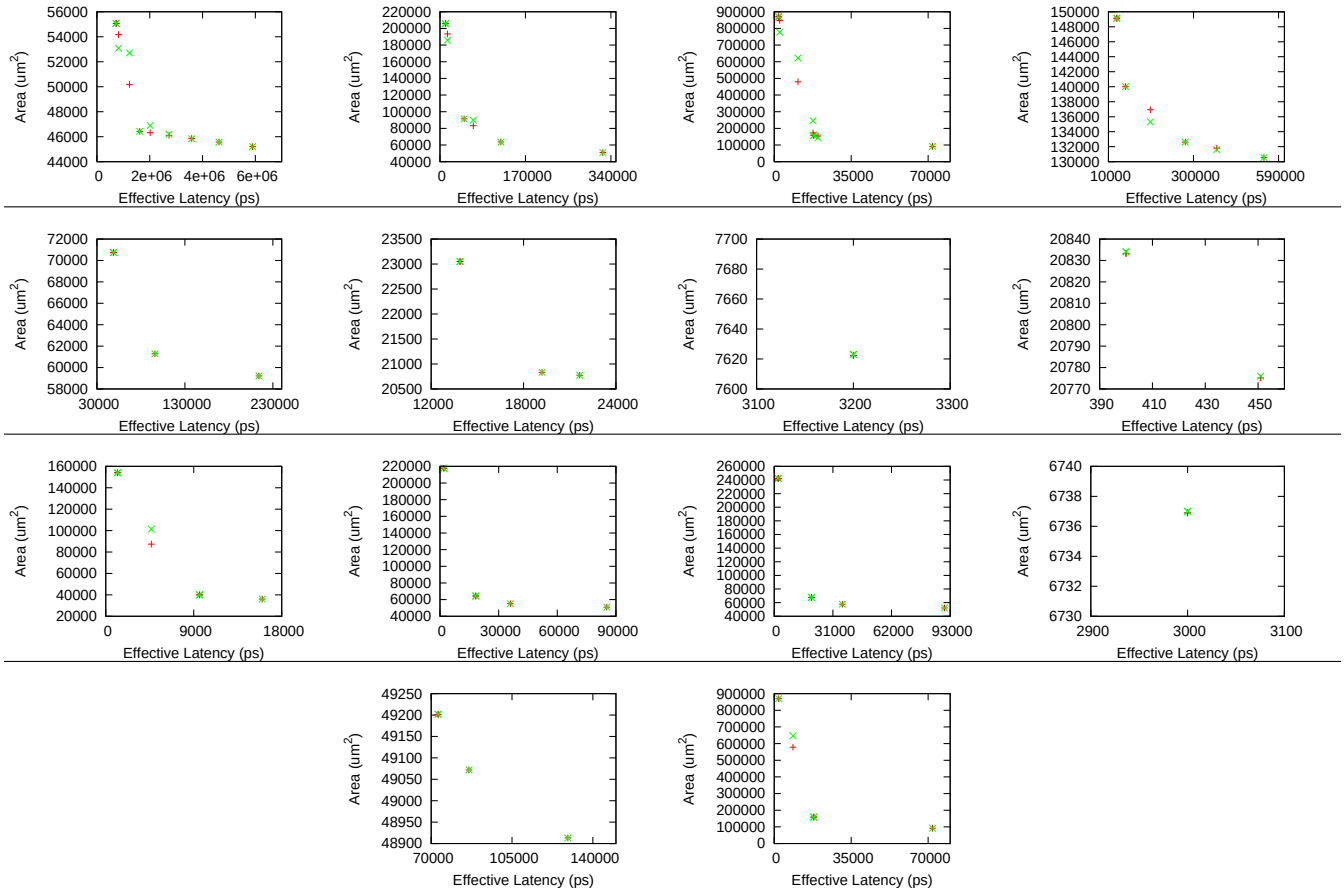


Figure 3.14: Adaptive component query of the MPEG2 system. The top row from left to right shows the query results (red from planning; green from HLS) of the MotionEstimation, Quantization, DCT8x8_ieee_1D_X, and MontionCompensation components. These components are the top-4 most influential components toward the area-throughput optimization of the MPEG2 system.

enable rapid system-level exploration at the early design stage. In particular, the automatic discovery of system bottlenecks (critical components) promotes tighter collaboration between the system architect and the component designers:

- The system architect can explore alternative system topologies and estimate potential cost and performance improvements. Examples of topological changes include splitting critical components and/or merging non-critical components. The splitting of critical components can be realized by shrinking component I/O sizes and adding more concurrent data flows to the system (or iteratively processing on smaller chunks of data). The merge of non-critical

Table 3.2: System area vs. throughput exploration as function of δ.

Benchmark	δ	0.2	0.15	0.1
JPEG	# System-Design Points	17	21	30
	Average $\mathcal{AM}$	1%	1%	1%
	Maximum $\mathcal{AM}$	7%	8%	9%
	# Points w/ $\mathcal{AM} \leq 3\%$	15	19	27
	Total # Queries	21	26	36
	Query Efficiency	2.4×	2.4×	2.5×
	Sequential HLS Tool Runtime	5hr	6hr	8hr
	Runtime Speed-up	1.4×	1.4×	1.5×
RS	# System-Design Points	9	12	17
	Average $\mathcal{AM}$	2%	2%	1%
	Maximum $\mathcal{AM}$	4%	5%	5%
	# Points w/ $\mathcal{AM} \leq 3\%$	7	11	14
	Total # Queries	22	28	36
	Query Efficiency	2.0×	2.1×	2.4×
	Sequential HLS Tool Runtime	22hr	29hr	39hr
	Runtime Speed-up	1.2×	1.2×	1.3×
DFSIN	# System-Design Points	9	10	14
	Average $\mathcal{AM}$	2%	2%	2%
	Maximum $\mathcal{AM}$	4%	13%	9%
	# Points w/ $\mathcal{AM} \leq 3\%$	7	9	12
	Total # Queries	18	19	23
	Query Efficiency	2.5×	2.6×	3.0×
	Sequential HLS Tool Runtime	40hr	41hr	49hr
	Runtime Speed-up	1.3×	1.4×	1.6×
MPEG2	# System-Design Points	19	24	35
	Average $\mathcal{AM}$	1%	1%	1%
	Maximum $\mathcal{AM}$	7%	10%	11%
	# Points w/ $\mathcal{AM} \leq 3\%$	17	22	31
	Total # Queries	57	63	77
	Query Efficiency	4.7×	5.3×	6.4×
	Sequential HLS Tool Runtime	35hr	38hr	46hr
	Runtime Speed-up	3.5×	4.1×	5.0×

components can rebalance the component implementation efforts, including synthesis runtime and query efficiency. Then, the system architect updates the cost/performance targets for the affected components and provides this information back to the component designers.

- Knowing particular new optimization goals, a responsible component designer can fine tune the component implementation with more aggressive or cost-effective optimizations, such as hand-crafting part of the design at RTL, applying a new or broader set of HLS knob settings, or a combination of the two optimizations. If the new optimization goals are achievable, the system architect should verify the optimization with a re-characterized component library. If they are not, then the system architect should resort to other optimization alternatives at the system level.

The above exploration process repeats until the final optimization goals are met. To this end, a rapid performance-analysis mechanism (such as the one used in SPEED) is instrumental in the fast convergence of the exploration process.

Related Work. General design space exploration algorithms include local-search heuristics like Simulated Annealing [159] and Genetic Algorithms [145]. Techniques based on clustering dependent parameters were proposed to prune the search space before exploration [77; 160]. Techniques based on predicting design qualities were also proposed to reduce the number of simulation/synthesis during exploration [18; 120]. However, all these approaches treat a system design as a whole entity, which unfortunately cannot be digested by current synthesis tools with practical runtime and memory requirements.

For component-based design, various approaches which exploit design hierarchies to compose system-level Pareto fronts have been proposed, e.g. [75; 85; 113]. Among these, our work is closer to the top-down approach of the "supervised exploration framework" [113], which adaptively selects RTL components for logic synthesis and thus gradually improves the accuracy of the system Pareto front. However, the important differences are that we operate at a higher level of abstraction, can handle components specified in SystemC, and can parallelize HLS runs to construct system Pareto fronts.

Concluding Remarks. We have presented a compositional methodology for HLS-driven design-space exploration of SoCs. Our approach is based on two new algorithms that combined allow

us to derive a set of alternative Pareto-optimal implementations for a given SoC specification by selecting and combining the best implementations of its components from a pre-designed soft-IP library.

Chapter 4

Compositional Approximation for SoC Characterization

In the previous chapter, I have presented the SDSE framework for system-level exploration with HLS. In this chapter, I present a complementary framework for the exploration with logic synthesis (LS), as RTL currently remains the mainstream entry point for IC design. Nevertheless, as the entry point moves to the system level, the two frameworks can be combined by using the HLS-based framework to explore micro-architectures and the LS-based counterpart to explore synthesizable clock periods.

Contributions. We present a novel and general approach to effectively coordinate the execution of logic-synthesis runs needed to explore the design space of a system that consists of a set C of interacting components. We model this computational task as a bi-objective optimization problem, in a rigorous mathematical framework, which we then solve with CAPS, a new algorithm that can derive an approximation of the *Pareto set* of the system by iteratively invoking a logic-synthesis tool (*the oracle*) to synthesize various instances of the components and, in doing so, implicitly build approximations for their Pareto sets (Fig. 4.1). For a given approximation error ϵ, CAPS minimizes the number k of queries necessary to derive the corresponding Pareto set of the system. Dually, for a given budget k of queries, CAPS is guaranteed to return an approximation of the Pareto set with the best approximation error ϵ. A typical application of our approach is the characterization of the power-performance trade-offs in the implementation of a medium-size IP core. In particular,

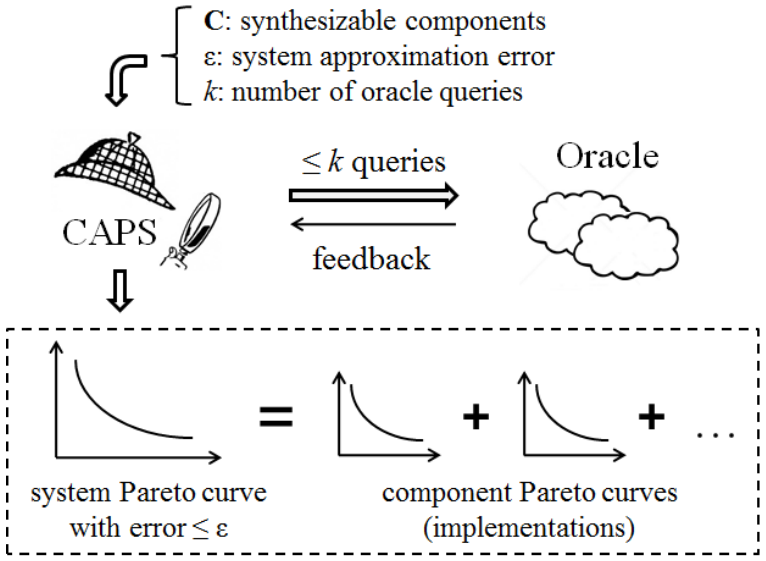


Figure 4.1: Supervised design-space exploration by Compositional Approximation of Pareto Sets (CAPS).

our algorithm:

- is an efficient online approximation algorithm, within a natural framework of bi-objective optimization problems, in terms of *(i)* the accuracy of the returned approximation and *(ii)* the CPU time needed to achieve such an approximation;
- is inherently scalable; it works only on the single components, and never runs a synthesis of the whole system, which would be impractical and/or infeasible for large systems;
- copes with the inherent *limited-information* setting in our context: i.e. the fact that no information about any of the components is known a-priori, and that at any given step the synthesis tool can only sample *one* point of the design space of *one* component;
- works on a bi-objective optimization problem coping with the typical asymmetry of synthesis tools, which allow controllability only on the clock period, but leave just observability on the power and area of the returned implementation.
- because of its simplicity, is amenable to parallelization and various heuristic improvements that may potentially improve its performance in practice.

A prototype CAD tool based on CAPS is described in Appendix A.2.

4.1 Problem Description

A multi-objective optimization problem Π has a set $\mathcal{I}$ of *valid instances*, every instance $I \in \mathcal{I}$ has a set of feasible solutions $\mathcal{S}(I)$. There are d objective functions, $f_1, \ldots, f_d$, each of which maps every instance I and solution $s \in \mathcal{S}(I)$ to a value $f_j(I, s)$. The problem specifies for each objective whether it is to be maximized or minimized. We assume that the objective functions have positive values. In our context, a "feasible solution" at the system (resp. component) level is an implementation of the system (resp. component).

Dominance Relation, Pareto Set. We say that a d-vector u *dominates* another d-vector v if it is at least as good in all the objectives, i.e. $u_j \geq v_j$ if f_j is to be maximized ($u_j \leq v_j$ if f_j is to be minimized). Similarly, we define domination between any solutions according to the d-vectors of their objective values. Given an instance I, the *Pareto set* $P(I)$ is the set of undominated d-vectors of values of the solutions in $\mathcal{S}(I)$. Note that for any instance, the Pareto set is unique.

Approximate Dominance Relation, Approximate Pareto Set. We say that a d-vector u *ϵ-covers* another d-vector v ($\epsilon \geq 0$) if u is at least as good as v up to a factor of $1 + \epsilon$ in all the objectives, i.e. $u_j \geq v_j/(1+\epsilon)$ if f_j is to be maximized ($u_j \leq (1+\epsilon)v_j$ if f_j is to be minimized). Given an instance I and $\epsilon > 0$, an *ϵ-Pareto set* $P_\epsilon(I)$ is a set of d-vectors of values of solutions that $(1+\epsilon)$-cover all vectors in $P(I)$.

Throughout this chapter, we will be concerned with bi-objective problems (i.e. optimization problems with two criteria / objective functions). We will use the symbols x and y to denote the two objective functions. In the experiments of Section 4.3 x is the clock period of a circuit and y is either its power dissipation or area occupation. Hence, in our setting, both objectives are to be minimized. Consider the plane whose coordinates correspond to the two objectives. Every feasible solution (design) is mapped to a point on this plane. If q is a point, we use $x(q)$, $y(q)$ to denote its coordinates; that is, $q = \big(x(q), y(q)\big)$.

Ratio Distance. We now define the *ratio distance* from p to q as $\mathcal{RD}(p,q) = \max\{x(q)/x(p) - 1, y(q)/y(p) - 1, 0\}$. By definition, the value $\mathcal{RD}(p,q)$ is the minimum value of $\epsilon \geq 0$ such that q ϵ-covers p. The ratio distance $\mathcal{RD}(p,q)$ intuitively captures the following: "how much worse is

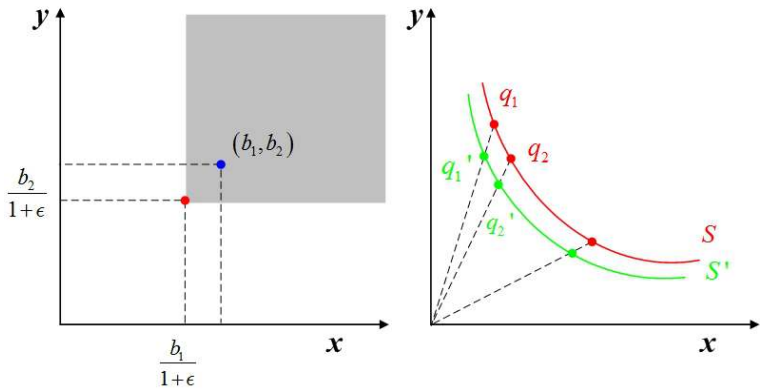


Figure 4.2: The point b ϵ-covers all points in the shaded region (left). Illustration of the ratio distance between two curves S, S' (right).

point q from point p". Note that this notion is asymmetric in p and q, which is why the ratio distance is not symmetric (see the example below). We also define the ratio distance between sets of points. If $S, S' \subseteq \mathbb{R}^2_+$, then $\mathcal{RD}(S, S') = \max_{q \in S} \min_{q' \in S'} \mathcal{RD}(q, q')$. In words, for a given point q in S, we find its closest point q' in S' (according to the ratio distance), and then we take the maximum over all points in S. As a corollary of this definition, the set $S \subseteq A$ is an ϵ-Pareto set for A if and only if $\mathcal{RD}(A, S) \leq \epsilon$. See Fig. 4.2 for an illustration of these definitions.

Example: Consider the points $q_1 = (1, 10)$ and $q_2 = (5, 6)$. Intuitively, we would like to say that q_2 is four times worse than q_1 (because of the ratio in the first coordinate), while q_1 is only $2/3$ times worse than q_2 (because of the ratio in the second coordinate). This is captured by the ratio distance and approximate dominance relation. According to our definitions, $\mathcal{RD}(q_1, q_2) = 4$ and q_2 4-covers q_1. On the other hand, $\mathcal{RD}(q_2, q_1) = 2/3$ hence q_1 $2/3$-covers q_2.

Restricted Problem and Oracle Access. It should be stressed that the objective space of our bi-objective problem is not given explicitly, but rather implicitly through the instance. In particular, we access the objective space $\mathcal{I}$ of Π via an appropriate oracle. We now describe our way of accessing the Pareto set *for each individual component*. We assume we can efficiently minimize one objective (the y-coordinate, aka power) subject to a constraint on the other (the x-coordinate, aka clock period). Our *oracle* is an efficient program that solves the following optimization problem.

Restricted Problem (for the y-objective): For a given component C_i (e.g. with a set of possible feasible designs $\mathcal{S}(C_i)$ discoverable by the commercial tool) and a bound b, either return a feasible

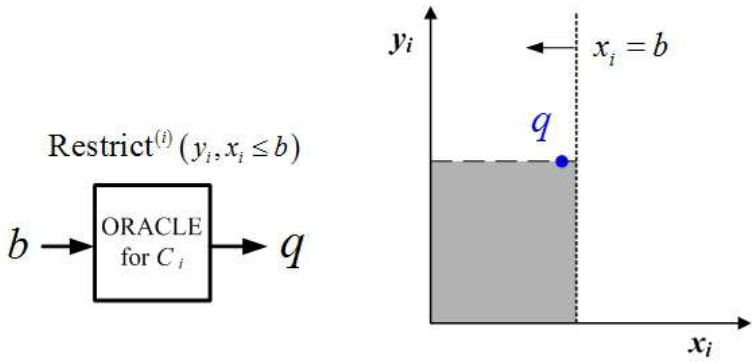


Figure 4.3: The oracle solves the Restricted Problem (left) while guaranteeing that there are no solution points in the shaded region (right).

solution point (i.e. an implementation for this component) q_i satisfying $x(q_i) \le b$ and

$$y(q_i) \le \min\{y \text{ over all designs } q \in \mathcal{S}(C_i) \text{ with } x(q) \le b\}$$

or report that there does not exist any solution q such that $x(q) \le b$.

For simplicity, we will drop the instance from the notation and use Restrict$^{(i)}(y, x \le b)$ to denote the solution returned by the corresponding oracle. (The superscript "(i)" means that we are dealing with the i-th component.)

We call *ideal* an oracle that satisfies the above. For the rest of this section, we assume an ideal oracle. In presenting the experimental results of Section 4.3 we will discuss the performance of our approach when the oracle presents a noisy behavior. In practice, if the oracle does not return a solution, it returns its best achievable value for x, i.e. its lowest bound (Fig. 4.3.)

Combining Components. Let $n \ge 1$ be the number of components $\{C_i\}_{i=1}^n$. Let $q_i \in \mathcal{S}(C_i)$, $i \in [n]$ be a feasible design for the i-th component and q be the design for the system obtained as the union of the q_i's. (In particular, q_i is a 2-dimensional vector—a point in the xy plane—whose first coordinate is the x-value, i.e. the clock period of the corresponding design, and whose second coordinate is its y-value, e.g. power.) We assume there are two *combining functions* $f_x, f_y : \mathbb{R}^n \to \mathbb{R}$ that tell us how q is related to the q_i's. In particular, $x(q) = f_x(x(q_1), \ldots, x(q_n))$ and $y(q) = f_y(y(q_1), \ldots, y(q_n))$. Both functions are assumed to be monotone increasing in each coordinate and efficiently computable. Note that, in our concrete setting, we have $f_x(x_1, \ldots, x_n) = \max_i x_i$ and $f_y(y_1, \ldots, y_n) = \sum_i y_i$. The clock period is the maximum among the clock periods of the

Algorithm 3 Single-Component Characterization Algorithm

Input: Oracle for the component's Restricted Problem

Output: Pareto set approximation for the component

1: $(b_{\min}, b_{\max}) \leftarrow$ the (min, max) values of the x–objective
2: $q_r \leftarrow \text{Restrict}(y, x \leq b_{\max})$
3: $q_l \leftarrow \text{Restrict}(y, x \leq b_{\min})$
4: $Q \leftarrow \{q_l, q_r\}$
5: **while** Error $\geq \epsilon$ and queries $< k$ **do**
6: Sort $Q = \{q_1, q_2, \ldots\}$ in increasing x-coordinate.
7: **for all** intervals (q_i, q_{i+1}) **do**
8: compute Error_i
9: **end for**
10: $I \leftarrow (q_j, q_{j+1})$ s.t. $j = \text{argmax}_i \text{Error}_i$
11: $b \leftarrow \sqrt{x(q_j) \cdot x(q_{j+1})}$
12: $q \leftarrow \text{Restrict}(y, x \leq b)$
13: $Q \leftarrow Q \cup \{q\}$;
14: **end while**

components and the total power is the sum of the individual component powers.

Formally, we consider the following problems:

Primal Problem. Given an error tolerance ϵ, compute an ϵ-Pareto set for the system using as few queries to the oracle as possible.

Dual Problem. Given a budget k on the number of queries, make k queries to the oracle and compute an ϵ-Pareto set for the system for the minimum possible value of ϵ.

4.2 Compositional Approximation of Pareto Sets

We first analyze the case of a system comprising of a single component, and then present the CAPS algorithm for the case of many components.

4.2.1 Single Component

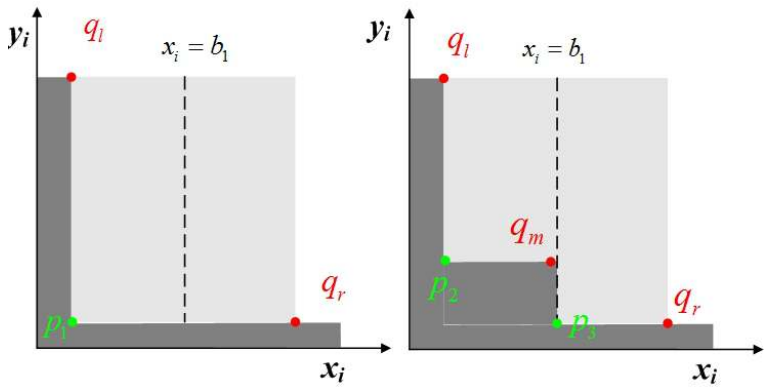


Figure 4.4: A geometric interpretation of the algorithm for the case of one component. The ideal oracle guarantees that there exist no solution points in the dark-shaded region.

We describe an algorithm for characterizing a single-component system. We start by providing an intuitive explanation of the algorithm. See Algorithm 3 for a detailed pseudo-code.

Our algorithm is iterative. In every step, it maintains an *upper approximation* and a *lower approximation* to the Pareto set. As the number of iterations increases, these two approximations become finer and finer, hence the error decreases.

The algorithm starts by finding the extreme points $q_l = (x(q_l), y(q_l))$ (leftmost), $q_r = (x(q_r), y(q_r))$ (rightmost) of the Pareto set (see Fig. 4.4). Each such point can be computed using appropriate queries to the oracle. Consider the point $p_1 = (x(q_l), y(q_r))$. At this point, the algorithm *knows* that the exact Pareto set lies entirely to the right of q_l and above q_r (see Fig. 4.3). Consider the triangle $\triangle(q_l p_1 q_r)$. By definition, the error of the initial approximation $\{q_l, q_r\}$ is at most $\mathcal{RD}(p_1, \{q_l, q_r\})$. That is, $\{q_l, q_r\}$ is the current *upper* approximation to the Pareto set, while p_1 is the current *lower* approximation. If $\mathcal{RD}(p_1, \{q_l, q_r\}) \leq \epsilon$, then the algorithm terminates and returns $\{q_l, q_r\}$. Otherwise, we call the oracle for b_1 being the "midpoint" of q_l, q_r. In particular, we set $b_1 = \sqrt{x(q_l) \cdot x(q_r)}$. This step corresponds to a *geometric subdivision* procedure; that is, a "binary search" in the *logarithms* of x and y. The reason we use a geometric subdivision is that our error measure (the ratio distance) is defined multiplicatively, as opposed to additively.

Let q_m be the obtained point. The right part of Fig. 4.4 shows the information we obtain about the space after q_m is returned. In particular, this implies that the error in the interval $\{q_l, q_m\}$ is at most $\mathcal{RD}(p_2, \{q_l, q_m\})$, while in the interval $\{q_m, q_r\}$ is at most $\mathcal{RD}(p_3, \{q_m, q_r\})$. The question

that now arises is which interval to update next. The algorithm uses a greedy criterion: it selects to update the interval in which the error is maximum. Similarly to the first stage, it divides the chosen interval geometrically and queries the oracle appropriately. Note that the algorithm is essentially the same for both the primal problem (ϵ is given) and the dual problem (k is given). The only difference is in the termination condition. In the former case, the algorithm terminates when the desired accuracy is attained, while in the latter case when the budget on the number of queries is exhausted.

4.2.2 Many Components

In the following, we describe an adaptive online algorithm CAPS to solve both the primal and the dual problems for the case of many components. Our algorithm is "online" in the following sense: it discovers the objective space by querying the oracle appropriately and uses the returned solutions (points) to compute an ϵ-Pareto set.

Consider a system comprising of many components. If we had access to the oracle to solve the Restricted Problem *for the system*, then it would be possible to implement Algorithm 3 and obtain the desired approximation. The main difficulty arises from the fact that we only have access to the oracle to solve the Restricted Problem *for the individual components*. However, we can overcome this difficulty as follows: at every step we identify the interval *at the system-level* where the approximation error is maximum. To achieve this, we first combine all component-level points to derive system-level Pareto points by f_x and f_y, and then compute for each system-level interval its corresponding approximation error by ratio distances. At this point, another difficulty arises. Since we can afford only one query at the given step, we need to select which component to update for the selected system-level interval. In fact, there are several ways to update an interval at the system-level (potentially as many as the number of components). We choose to update the component that can potentially reduce the system error as much as possible. This requires the application of f_x, f_y, and the ratio distance for each component to estimate a potential system-level error reduction due to that component. Once we decide which component to update and the relevant interval in that component, we update that interval by dividing it into halves as in Algorithm 3. This step sends a query to the oracle and updates the selected component's points. Finally, we add (still by f_x and f_y) a new system-level point according to the component update,

Algorithm 4 CAPS

Input: Oracle for the components' Restricted Problem

Output: System Pareto set approximation

1: $(b_{\min}, b_{\max}) \leftarrow$ the (min, max) values of the x–objective
2: **for** $i = 1, 2, \ldots, n$ **do**
3: $\quad q_r^{(i)} \leftarrow \text{Restrict}^{(i)}(y, x \leq b_{\max})$
4: $\quad q_l^{(i)} \leftarrow \text{Restrict}^{(i)}(y, x \leq b_{\min})$
5: **end for**
6: $Q = (q_1, \ldots, q_m) \leftarrow$ combine q_r's and q_l's by f_x, f_y
7: sort Q by increasing x–coordinates, i.e. from left to right
8: **while** Error $\geq \epsilon$ and queries $< k$ **do**
9: $\quad$ **for all** interval (q_i, q_{i+1}) **do**
10: $\quad\quad$ compute Error_i
11: $\quad$ **end for**
12: $\quad I \leftarrow (q_j, q_{j+1})$ s.t. $j = \text{argmax}_{i=1,\ldots,m}\text{Error}_i$
13: $\quad C_l \leftarrow$ component w/ max potential error reduction in I
14: $\quad (q_k^{(l)}, q_{k+1}^{(l)}) \leftarrow$ the relevant interval of C_l
15: $\quad b_m \leftarrow (x(q_k^{(l)}) + x(q_{k+1}^{(l)}))/2$
16: $\quad q_m^{(l)} \leftarrow \text{Restrict}^{(l)}(y_l, x_l \leq b_m)$
17: $\quad q_{sys} \leftarrow$ derive a new system point using f_x, f_y, considering $q_m^{(l)}$
18: $\quad$ add q_{sys} into Q, sorted by increasing x–coordinates
19: **end while**

and then proceed to the next system-level interval for error reduction. The iteration continues until a termination condition (discussed below) is satisfied.

The pseudocode for CAPS, given as Algorithm 4, is basically the same for both the primal and the dual problems. The only difference is the termination condition. For the Primal Problem, it terminates when the error falls below ϵ. For the Dual Problem, it terminates when it makes k queries.

4.3 Experimental Results

In this section we experimentally verify: *(i)* the accuracy of the approximated Pareto set produced by CAPS i.e. the ratio distance from the exact Pareto curve to the approximation, and *(ii)* the convergence time of CAPS i.e. the time it takes to deliver the required approximation (or, dually, the best approximation within the time budget). We will show that *besides having approximation-accuracy guarantees on its worst-case performance, in practice* CAPS *can return results that are more accurate in a shorter time, even in case of a non ideal (i.e., noisy) oracle.*

Experimental Set-Up. We implemented the CAPS algorithm and evaluated its efficiency by applying it to characterize three designs, which were specified at the RTL level in synthesizable Verilog/VHDL. Specifically, we used (a) *StereoDepth*, an SoC for video-rate stereo depth measurement that consists of 4 components [50]; (b) an *MPEG2 Encoder* consisting of 26 components; and (c) *MinSoC*, which consists of 8 components and was designed by starting from a public-domain extensible SoC based on the *OR1200* processor [3] and adding two more cores compatible with the Wishbone standard (a *DVI/VGA Video Controller* and an *SD Card Controller*).

To play the role of the *oracle* of Fig. 4.1, we chose a widely-adopted commercial tool for logic synthesis equipped with two industrial standard-cell libraries developed for two different technology processes ($90nm$ and $45nm$). While the three designs can be considered as fairly small with respect to state-of-the-art SoCs, they are large enough to require the execution of logic synthesis in a compositional fashion on their constituent components. Further, they are good candidates to represent medium-size IP cores that need to be designed for reusability to enable design space exploration of future multi-core SoCs for embedded applications as discussed at the beginning of this chapter.

We define performance as the maximum clock frequency at which the design can run. This is the inverse of the *target clock period* T_{clk} that is given to the oracle as the controllable constraint. To increase T_{clk} generally leads to the synthesis of circuit implementations that have both smaller power dissipation and area occupation. Hence, we used CAPS for exploring both *power/performance* and *area/performance* trade-offs as examples of two different bi-objective optimization problems. We will see that this choice allows us to evaluate CAPS in the presence of both ideal and noisy oracle behaviors.

Power/Performance Trade-offs. Fig. 4.5 shows the final approximation that we obtained by

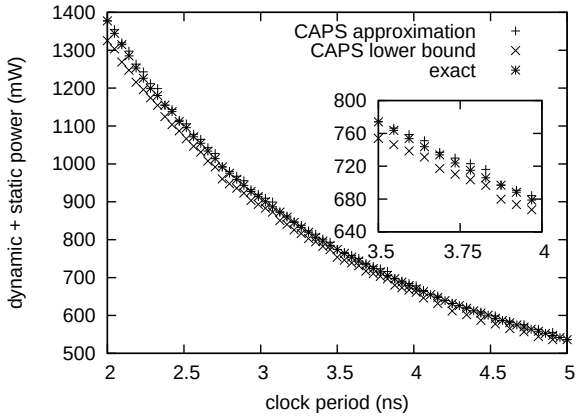


Figure 4.5: Power/performance trade-off of StereoDepth at $90nm$ ($\epsilon = 3\%$).

running CAPS to solve the Primal Problem (with error tolerance $\epsilon = 3\%$) to characterize the power/performance trade-offs of StereoDepth with the $90nm$ standard-cell library. As the *returned clock period*, i.e. the clock period returned by the oracle, varies between $2ns$ and $5ns$ the power dissipation varies between $550mW$ and $1400mW$.

The ratio distance between the approximation curve and the "hypothetical" lower-bound curve that are dynamically computed by CAPS matches the target error upper bound ϵ. The points of these curves are computed iteratively through a sequence of 65 oracle queries which required an aggregate 63 hours until the target $\epsilon = 3\%$ is matched. This is just 25% of the number of queries (and about 50% of the CPU time) necessary to compute the exact Pareto set, whose curve lays between the approximation and lower bound curves as we expected and as it is shown also in Fig. 4.5. Now, the exact Pareto set can generally be obtained by executing a *long* sequence of synthesis runs (corresponding to a series of equally-spaced values of T_{clk}) *for each component*. Instead, CAPS dynamically adapts the distribution of the queries to the portions of the design space that actually need to be explored to improve the accuracy of the approximation. For instance in the case of StereoDepth, as shown in Fig. 4.6, CAPS repeatedly invokes the oracle on *block2* and, to lesser extents, on *block0* and *block1*, while avoiding *block3* after the initial queries necessary to establish its extreme points. This is possible because in its effort to approximate efficiently the Pareto set of a system, CAPS keeps track of its improvements on approximating the Pareto set of its components as well as the impact that each component has on the overall system design. Indeed, the richer the Pareto set of a component is and the more a component impacts the design

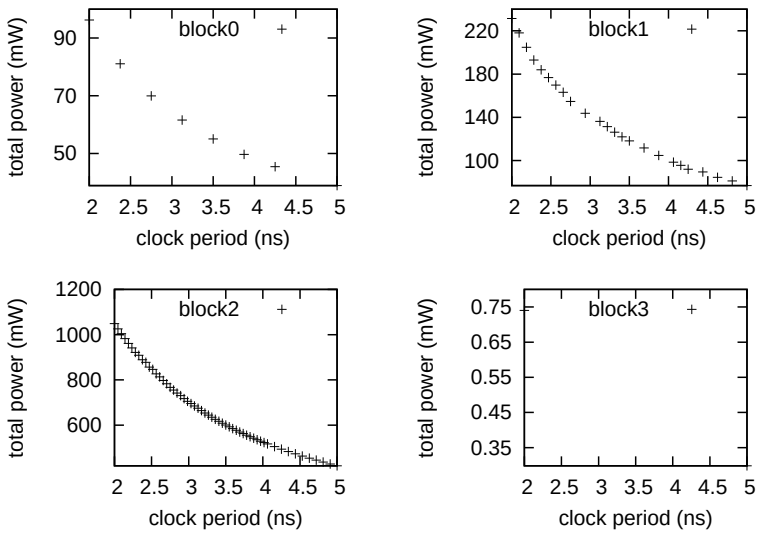


Figure 4.6: Component sampling of StereoDepth at $90nm$ to build the system approximation of Fig. 4.5.

exploration of the system in which it operates, the more CAPS samples its design space.

Fig. 4.7 shows the evolution of the *guaranteed error* ϵ for each new query that we obtained by running CAPS to solve the Primal Problem (with error tolerance $\epsilon = 3\%$) to characterize the power/performance trade-offs of the MPEG2 Encoder with the $45nm$ library. As expected, the error decreases monotonically over time, until it reaches the target approximation. Since for these experiments we also computed the exact Pareto set, we can evaluate also the *actual approximation error* ϵ_a, i.e. the ratio distance between the exact and the approximated Pareto sets. In practice, as shown in Fig. 4.7, after each query CAPS returns an actual error ϵ_a that is much lower than the guaranteed error ϵ. For example, in this case we aimed at $\epsilon = 3\%$ and CAPS returned the approximated set after 93 iterations (22 hours). In practice, though, the actual error ϵ_a at the end of the run is lower than 3% while the 3% error was achieved much earlier, after only 62 queries (16 hours). Note that for this design the derivation of the exact Pareto set needs a total of 1430 queries (142 hours). *Hence, in practice, with* $8\times$ *less queries (or CPU time) to the oracle, we can approximate the Pareto curve of power/performance with an error lower than* 3%.

CAPS adapts the convergence time to the design space that can be explored, depending on the design, the number of components, and the technology. Table 4.1 shows the results for each benchmark. Columns "Exhaustive Search" combines pre-computed component curves to derive

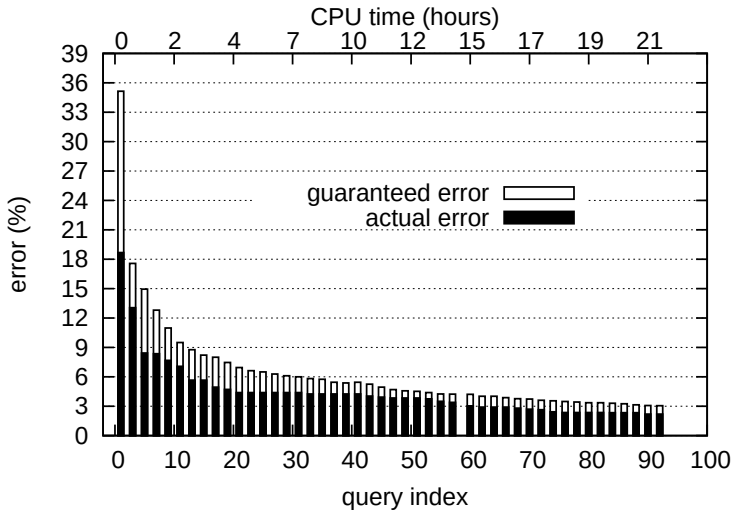


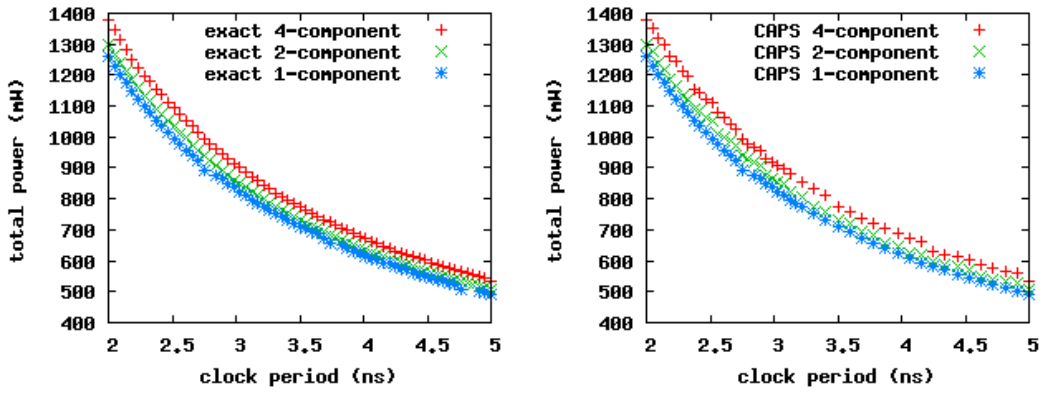
Figure 4.7: Error convergence of MPEG2 Encoder by query index (lower axis) and by CPU time (upper axis).

Table 4.1: Power/Performance trade-offs by CAPS. Note that the unit of CPU time is hours, and the parenthesized value of "k" and "CPU Time" is a ratio over *Exhaustive Search*.

Design	Tech	Exhaustive Search ($\epsilon = 0\%$)		CAPS Primal ($\epsilon = 4\%$)				CAPS Dual ($k = 60$)		
				ϵ (guaranteed)		ϵ_a (actual)				
		k	CPU Time	k	CPU Time	k	CPU Time	ϵ	ϵ_a	CPU Time
StereoDepth	$90nm$	260	126	50 (19%)	49 (39%)	44 (17%)	44 (35%)	3.2%	3.1%	60 (48%)
MinSoC	$90nm$	520	27	96 (18%)	9 (33%)	87 (17%)	8 (30%)	6.0%	4.7%	6 (22%)
MinSoC	$45nm$	520	29	37 (7%)	4 (14%)	25 (5%)	3 (10%)	2.7%	2.0%	7 (24%)
MPEG2 Encoder	$45nm$	1430	142	65 (5%)	16 (11%)	45 (3%)	11 (8%)	4.2%	3.0%	15 (10%)

an exact system curve. "CAPS Primal", given $\epsilon = 4\%$, finds an ϵ-Pareto curve for the system in k queries, and "CAPS Dual", given $k = 60$, finds a system Pareto curve with guaranteed error $\leq \epsilon$. For all the benchmarks, CAPS Primal requires only 5–19% queries and 11–39% CPU time of Exhaustive Search to achieve a guaranteed 4% ϵ (and even less for a 4% ϵ_a). On the other hand, CAPS Dual can achieve a guaranteed 2.7–6.0% ϵ (again, an even better ϵ_a) in 60 queries.

Sensitivity Analysis on System Decomposition. With the efficiency that CAPS can offer for system characterization, we are able to further apply it to facilitate the sensitivity analysis on system decomposition. How a system is decomposed could impact its design quality (e.g. total power consumption) versus design efforts (e.g. synthesis tool runtime). In RTL design, if a system is synthesized as a whole entity, synthesis tools have opportunities to perform aggressive optimization



(a) Exact Characterization

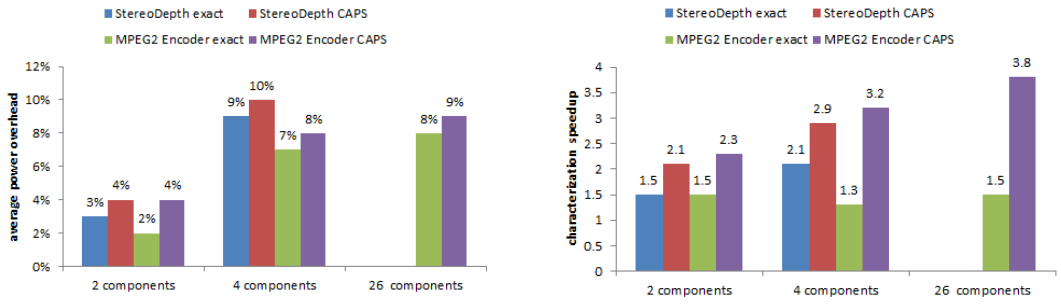
(b) CAPS Approximation ($\epsilon = 3\%$)

Figure 4.8: Power/performance trade-offs of StereoDepth at $90nm$ composed of 1, 2, and 4 components, by (a) exact characterization and (b) CAPS approximation with $\epsilon = 3\%$.

across the boundary among system hierarchies to achieve better quality of results in exchange for longer runtime and more memory. On the other hand, tools can gain runtime/memory reduction by separately synthesizing individual components of the system, while leaving input/output logic of the components unoptimized. A case study on the benchmark StereoDepth at $90nm$ is shown in Fig. 4.8.

In Fig. 4.8(a), we first compare the power/performance trade-offs of the *exact* Pareto sets of StereoDepth which comprises 4 components (i.e., the original design), 2 clustered components, and one supercluster component. As expected, the fewer components the system is decomposed into, the less total power consumption it requires, due to more synthesis optimization across hierarchical boundaries. Over all the points in the three exact Pareto sets, we see 3% and 9% average power consumption overheads of the 2- and 4-component cases, respectively, compared with the 1-component case. However, the CPU time required for the exact characterization of the 2- and 1-component cases, compared with the 4-component case which requires 126 hours, is on average 1.5× and 2.1× longer, respectively. This trend illustrates the power overhead and runtime advantage incurred by decomposing a system into more components for synthesis.

In Fig. 4.8(b), we apply CAPS with $\epsilon = 3\%$ to approximate the exact Pareto sets shown in Fig. 4.8(a). CAPS takes 63 hours (50% of the exact characterization) to approximate the 4-



(a) Power Consumption Overhead

(b) Characterization Runtime Speedup

Figure 4.9: Sensitivity analysis by comparing multi-component synthesis with whole-system (single-component) synthesis, in terms of (a) power consumption overhead and (b) characterization runtime speedup. CAPS runs with $\epsilon = 3\%$ on both benchmarks.

component set, and 2.1× and 2.9× longer for the 2- and 1-component sets, respectively. The average power overheads of the approximate 2- and 4-component sets, compared with the approximate 1-component set, is 4% and 10%, respectively. The additional 1% overhead over the exact characterization comes from the approximation error, and is bounded by user-specified ϵ (in this case study, 3%). Overall, for the sensitivity analysis on StereoDepth (3 Pareto sets), compared with exact characterization, CAPS offer 1.7× total runtime speedup (329 hours vs. 554 hours) with very limited inaccuracy (1%).

Another sensitivity analysis on the benchmark MPEG2 Encoder at $45nm$ is summarized in Fig. 4.9 with the StereoDepth benchmark results. The original MPEG2 Encoder contains 26 components, and was clustered into 4, 2, and 1 components for the analysis. Fig. 4.9(a) shows that the growth of average power overhead of MPEG2 Encoder becomes saturated at 7–9% as the number of components increases from 2 to 26. Also, the power overhead coming from approximation errors is bounded to 1–2% by ϵ, which we set to 3% in the analysis. On the other hand, in Fig. 4.9(b) we see that the exact-characterization runtime of the MPEG2 multi-component cases remains 1.3–1.5× faster than its 1-component case. Even better, the runtime speedup by CAPS keeps growing from 2.3× to 3.8× as the number of components increases. Moreover, the total runtime required to perform the sensitivity analysis on the MPEG2 Encoder (4 Pareto sets) by CAPS is 3.9× less than the case of exact characterization (165 hours vs. 641 hours). Since our benchmarks approximately rep-

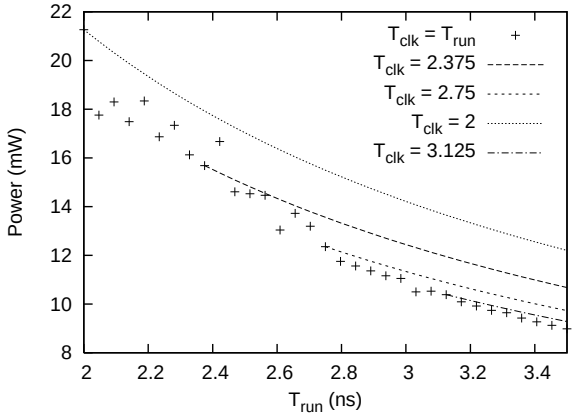
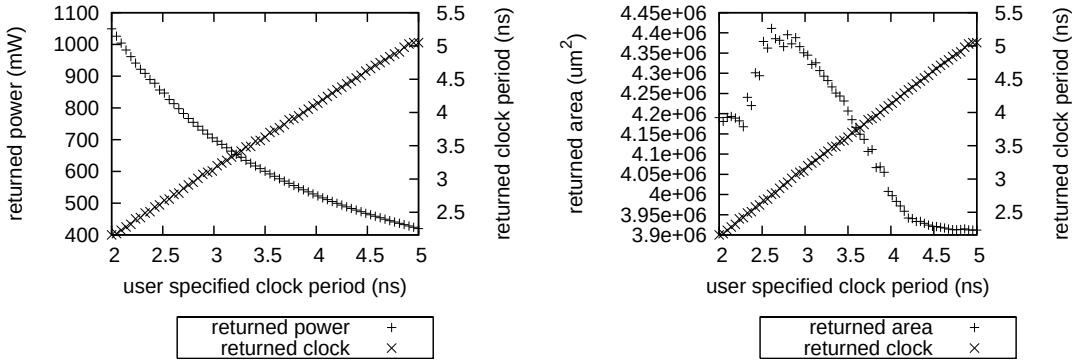


Figure 4.10: The importance of synthesis for power/performance characterization. T_{clk} and T_{run} are the target clock frequency during synthesis and the operative clock frequency, respectively.

resent medium-size IP cores, all the experimental results suggest that *(i)* component-based design is practical and inevitable for a big and complex SoC to shorten its design process, and *(ii)* CAPS provides effective and efficient realization of supervised design space exploration for characterizing such SoC in RTL.

Importance of Synthesis-Driven Power/Performance Characterization. While we limited our experiments to varying the target clock frequency T_{clk} for the nominal voltage supply, the reader may wonder why it is not sufficient to synthesize just a single implementation for the smallest possible value of T_{clk} at *design time*, and then simply reuse it also with higher values of the *operative clock period* T_{run} across various designs at *run time*. After all, this is what is typically done when applying dynamic clock frequency scaling methods for low power design. The answer is given by Fig. 4.10 which, for the case of the *OR1200* processor core of *MinSoC*, shows that power dissipation not only scales down with the *running* frequency of the clock, but also with the clock frequency targeted during synthesis, i.e. the inverse of T_{clk}. Each of the individual points in Fig. 4.10 represents the estimated power consumption returned by the oracle for a given T_{clk}, i.e. corresponding to a distinct circuit implementations of the processor that can be chosen at design time. The four lines, instead, show the power trend for four of these implementations corresponding to scaling the operative clock frequency T_{run}. Notice that the points are always below the lines, meaning that a circuit synthesized for the clock frequency T_{run} at which it will operate never wastes power with respect to another circuit that was over-optimized for T_{clk} and then clocked



(a) Power vs. Clock Period

(b) Area vs. Clock Period

Figure 4.11: Comparing the oracle behavior in terms of: (a) returned power and (b) area when synthesizing the *block2* component of StereoDepth at $90nm$ for various T_{clk} values. The diagonal line shows the returned clock period.

for $T_{run} > T_{clk}$. Also, note that every point that is not below the lines is actually a dominated point in the Pareto set, which CAPS automatically eliminates. Given the importance of design for reusability and the growing trends of designing multi-core SoCs with multiple clock domains, the accurate synthesis-driven characterization of the power/performance of an IP core is essential to demonstrate its best performance across a large dynamic range of power and frequency.

Area/Performance Trade-offs (and Noisy Oracle Behaviors). While power/performance trade-off is the most important characterization of an IP core, we analyze also the area/performance trade-off because it allows us to evaluate the CAPS algorithm when the oracle manifests a noisy behavior. Different from the power/performance design space, the area and operative clock period T_{run} are independent at run time: the area can be traded off for performance only at design time, by changing the target clock period T_{clk}. Fig. 4.11 compares the returned values estimated by the oracle for power dissipation (a) and area occupation (b) as well as the values of the returned clock period (the diagonal line in both figures) for values of T_{clk} increasing between $2ns$ and $5ns$. First, it is clear that the oracle is able to match T_{clk} with the returned clock period well, as it is expected: in fact, most synthesis tools are built to first meet a target performance constraint

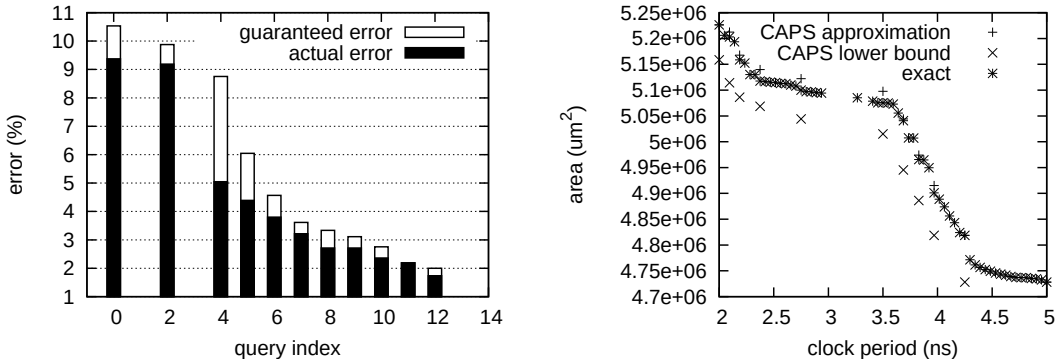
and then do a "best effort" optimization for other objective functions such as area and power. [1] Also, the returned power values are monotonically decreasing for increasing values of T_{clk} thanks to the strict decreasing dependency of the dynamic power portion on the operative clock period T_{run}. This is a confirmation that, to characterize power/performance trade-offs, our oracle behaves practically as the ideal oracle defined in Section 4.1. This is not the case, however, for the returned area values which do not decrease monotonically as T_{clk} decreases. For instance, when requested to synthesize a circuit that can run at $300MHz$ ($T_{clk} = 3.3ns$) the oracle returns an implementation that has an area larger than the one returned for a circuit that can run at $500MHz$ ($T_{clk} = 2ns$). This is what we refer to as a *noisy oracle behavior*. Practically, it can be explained as follows: in an effort to achieve $500MHz$, the commercial logic synthesis tool activates some additional, more time-consuming, heuristics that turn out to be beneficial also in terms of area; for less demanding timing constraints, the tool avoids to invoke unnecessary optimizations, advantaging instead a minimization of CPU time. The implication of having a noisy oracle is that it can return a point in the area/performance plane that is not part of the Pareto set, with a probability that is not negligible. In particular, two scenarios may arise:

- *The returned point dominates* other previously-returned points. In this case, CAPS treats the query as valid and progresses regularly. The only difference is that after this query the current approximation error might have temporarily increased, thus violating monotonicity. In fact, during the previous steps, the approximation of the Pareto set of the system (and, particularly, the lower-bound point that dictates the error) was based on some value that now cannot be considered valid anymore because it became dominated by the new acquired point.

- *The returned point is dominated* by the current Pareto set. Here, a more interesting problem arises. Not only is the query wasted, since it does not reduce the current approximation error of the system Pareto set, but also the portion of the space that has been just queried will likely remain the source of the biggest error (unless there is another portion that has exactly the same error) and CAPS will re-attempt the same query.

In order to overcome the source of deadlock due to the second scenario, we introduce a practical heuristic that drives the selection of the next query: when a query returns a dominated point, CAPS

[1] Indeed, this is why in our approach we define the restricted optimization problem in an asymmetric way with only one controllable objective function.



(a) Error Convergence

(b) Area vs. Clock Period

Figure 4.12: Area/performance trade-offs (coping with a noisy oracle behavior) for StereoDepth at $90nm$.

performs a second query in the middle of the right semi-space, where the clock period is longer. The rationale behind this decision is the following: if the returned point is dominated, then the noise of the oracle is a consequence of very limited room for exploration in the left semi-space, i.e., the considered clock-period increase from the faster design that we already sampled (left-hand bound) is not enough to allow the tool to reduce the area by much. Hence, it is likely that the space that deserves to be explored lies on the right semi-space, where the clock-period increase is even bigger. However, if the exploration in the right semi-space keeps returning a dominated point, we turn to explore the left semi-space. Finally, if a dominated point is still encountered, we search in other components.

Fig. 4.12(a) shows that even when CAPS has to rely on a noisy oracle to characterize StereoDepth at $90nm$ for its area/performance trade-off, the approximation of the Pareto set can still converge to a target $\epsilon = 2\%$. The missing bars in the figure represent queries that returned a dominated point. Note that, even though the oracle is very noisy for at least one component, the returned point is dominated only in two occasions. Further, this happens mostly at the beginning, where the space to be searched is bigger. After just a few queries, the regions of the space where a trade-off is meaningful will keep being queried, while CAPS mitigates the noise of the oracle for the other regions. Note that, as expected, CAPS converges much more quickly for area than power, because the area trade-off space is smaller. Finally Fig. 4.12(b) shows the area distribution of the

approximated and the reference exact Pareto sets for StereoDepth at $90nm$.

4.4 Remarks

Related Work. Multi-objective optimization, which provides the theoretical framework for design-space exploration, is an area at the interface of operations research and computer science that has been being extensively investigated in the literature [67]. While our model is very similar to the standard multi-objective framework defined by Papadimitriou and Yannakakis in [137] (and further studied in [59; 60; 61; 180]), our algorithmic results are novel and are not implied by these works. The main differences between these works and our setting lies in (i) the compositional aspect of the current work and (ii) the metric defining the performance ratio.

Roughly, the results in [59; 60; 61; 180] correspond to the case of one component. However, we point out that, even for the single-component setting, our oracle in this work is different (in fact, weaker) than the corresponding oracles used in those works. This is dictated by our underlying application, since we used a commercial tool to implement the oracle. More specifically, the aforementioned related works use a symmetric oracle (which allows in addition to minimize x subject to a bound on y).

Moreover, in this work, we are interested in minimizing the number of *queries* to the oracles, while the authors of [59; 60; 61; 180] consider the minimum number of *points* in an ϵ-approximation. While these notions are related for the case of one component, they can be very different even for a system of two components.

Concluding Remarks. We have presented a novel supervised design space exploration framework for system-level characterization, whose Pareto set is accurately constructed by combining approximated Pareto sets of components. The framework queries oracles on individual components only, and does not require a-priori knowledge about the components. In particular, for RTL to gate-level synthesis oracles, we have proposed an online approximation algorithm, which can coordinate the order of synthesis runs to derive the system-level approximation in the shortest possible time.

Part II

Component-Level Design-Space Exploration

Chapter 5

Knob-Setting Optimization for High-Level Synthesis

The SDSE frameworks for system-level exploration (presented in Part I) are based on the characterization of the component designs that are ready for release. However, the design of the components per se is challenging due to the bottlenecks described next.

The industrial adoption of HLS tools is still at an evaluation stage [123]. One of the major bottlenecks is that DSE with HLS requires substantial efforts for setting the micro-architecture knobs, whose cardinality in general grows exponentially with the size of real design. Another bottleneck is the long runtime of HLS tools: in fact, a simple Discrete Fourier Transform (DFT) design in SystemC may require half to an hour of CPU time for one HLS run to complete on a modern computer. Combined, these two challenges are a major roadblock towards the successful realization of automatic ESL design.

To address these challenges, DSE methods based on *iterative refinement with learning models* are becoming popular [112; 121; 134; 186; 187; 197; 198]. These methods leverage *learning*, *training*, *refinement*, and *sampling* optimizations to boost the effectiveness and the efficiency of the DSE process. The learning optimization corresponds to the selection of a machine-learning model that is responsible for predicting the HLS QoR without the need to actually invoking an HLS job. Each design objective requires a dedicated model instance for the prediction. The prediction accuracy of a learning model is affected by the generation of initial training examples (i.e. the training

optimization) and the expansion of the training examples (i.e. the refinement optimization). The training examples are a set of actual mappings from an HLS script to an objective value of a design implementation. Along with the DSE process, the learning models are used to identify Pareto-promising knob settings (i.e. the sampling optimization). A *Pareto-promising* knob setting is a knob setting that is likely to yield a Pareto-optimal implementation based on the predicted objective values.

In this chapter, targeting the HLS of hardware accelerators within an ASIC design flow, we make the following contributions to address the DSE-with-HLS challenges.

- We apply the general iterative-refinement framework [134] to HLS, with three particular enhancements: *(i)* a general training optimization based on Transductive Experimental Design for improving the accuracy of an arbitrary learning model, *(ii)* a scalable sampling using randomized selection for handling large design spaces, and *(iii)* a learning-model selection tailored to the synthesis of accelerators in ASIC designs. Our enhancements are complementary to the enhancement presented by Xydis *et al.* [187], who focus instead on the refinement optimization. Also, compared to recent works that also focus on the optimizations of design-space sampling and learning-model selection [197; 198], our approach for DSE-with-HLS offers a more accurate learning model that can achieve a faster convergence of the DSE process. Furthermore, the duration of the DSE process can be further reduced by the application of our training optimization.

- We developed a tool for DSE with HLS called LEISTER which stands for "Learning-based Exploration with Intelligent Sampling, Transductive Experimental design, and Refinement." With two execution modes, LEISTER offers a unique freedom to switch between (a) searching the optimal DSE results and (b) saving the DSE turnaround time. A prototype implementation of LEISTER is described in Appendix A.3.

- We applied LEISTER to the design of eight hardware accelerators for eight benchmark applications from the CHStone suite. In summary, our experimental results show that compared with the previous approach [197; 198]: *(i)* LEISTER can converge faster by 2.20X on average and *(ii)* the learning model of LEISTER is more accurate in predicting the circuit execution latency by 7.01% and the circuit area by 2.87%, with a 50% less runtime for model evalua-

tion. Moreover, with the scalable sampling optimization, LEISTER can further reduce the average runtime for training sample generation by 93.31X and for training sample expansion by 7.98X.

5.1 Problem Description

We refer to the directives used in a HLS script for determining micro-architecture choices as *knobs*. Table 3.1 has already listed some common knobs and their settings available in modern HLS tools. Depending on the knob types, the choices of each knob setting can vary. For example, while the choice of function inlining is binary (either to inline or not to), loop manipulation offer many alternative options. Let us denote by c the max number of choices of a given knob setting.

Given a design specified in high-level languages, e.g. C or SystemC, HLS tools generate an optimized RTL by taking a set of knob settings as input constraints. For a given design specification, the number of places where HLS knobs can be applied can vary and is generally large. For instance, even a small SystemC design may have many functions, many (nested) loops, many arrays, and so on. Let p denote the number of knob-applicable places. Then the total combination of knob settings grows exponentially ($O(c^p)$).

To give an example of the design complexity, we use a case study of DFT designed in SystemC. The DFT design implements an iterative-FFT algorithm [44] with a synthesizable fix-point library using 45nm technology. The maximum number of knob-setting combinations for the DFT design is 360,448. In order to establish a ground truth for the analysis of Pareto optimality, we synthesized a restricted set of 242 knob settings, which cover 11 loop manipulations, 2 function-inlining choices, and 11 feasible clock periods. The 242 HLS runs took an aggregated 84-hour CPU time with a commercial HLS tool. Throughout this chapter, we refer to the 242 knob settings as the knob-setting space of the DFT.

In general, the exponentially-growing knob-setting space makes DSE with HLS very different from DSE with processor simulators or IP generators. In the latter cases, p can be assumed to be a constant, in practice. For example, only a limited set of parameters needs to be determined for processor simulators, such as the number of cores and the size of L1/L2 cache, and similarly for IP generators, for which the parameters are the I/O size, algorithms, etc. Instead, the fact that

p is typically a large constant for DSE with HLS makes it a more challenging problem. Since to exhaustively explore the set of all possible RTL designs that can be obtained with HLS is unfeasible, the goal of DSE with HLS is to approximate the Pareto set of RTL designs.

We define the *DSE with HLS* problem as follows.

Problem 3. *Given a high-level design specification and HLS tool with a budget of b runs, find the best approximate Pareto-optimal set of RTL designs without exceeding b.*

Note that we consider a multi-objective DSE problem. Although throughout this chapter we focus on 2-objective cases for simplicity, all our discussions and findings are generally applicable to higher dimensional cases.

In order to measure the quality of an approximate set of Pareto-optimal designs, we utilize the metric of *average distance from reference set* (ADRS) [134]. Consider a two-dimensional (area $\mathcal{A}$ vs. effective-latency $\mathcal{T}$) design space[1]. For both objectives, the smaller the objective, the better the RTL implementation. Given a reference Pareto set $\Pi = \{\pi_1, \pi_2, \dots | \pi_i = (a, t), a \in \mathcal{A}, t \in \mathcal{T}\}$ and an approximate Pareto set $\Lambda = \{\lambda_1, \lambda_2, \dots | \lambda_j = (a, t), a \in \mathcal{A}, t \in \mathcal{T}\}$,

$$\text{ADRS}(\Pi, \Lambda) = \frac{1}{|\Pi|} \sum_{\pi \in \Pi} \min_{\lambda \in \Lambda} \delta(\pi, \lambda),$$

where

$$\delta(\pi = (a_\pi, t_\pi), \lambda = (a_\lambda, t_\lambda)) = \max\{0, \frac{a_\lambda - a_\pi}{a_\pi}, \frac{t_\lambda - t_\pi}{t_\pi}\}.$$

Note that the lower $\text{ADRS}(\Pi, \Lambda)$, the closer the approximate set Λ to the reference set Π.

5.2 An Iterative Refinement Framework

Fig. 5.1 illustrates an iterative-refinement framework [121; 134; 186; 197]. The main idea of the framework is twofold: *(i)* to approximate the HLS tool H by a learning model $\tilde{H}$, and *(ii)* to use the fast $\tilde{H}$ for predicting the Quality-of-Result (QoR) space, instead of invoking the time-consuming H. Algorithm 5 implements the major steps of the framework. Initially, the framework trains $\tilde{H}$ by spending some HLS runs (the *training* stage in Lines 6–8), and then finds an approximate Pareto

[1]We define the *effective latency* as the product of *(i)* the clock period of the circuit and *(ii)* the clock cycle count to execute a task.

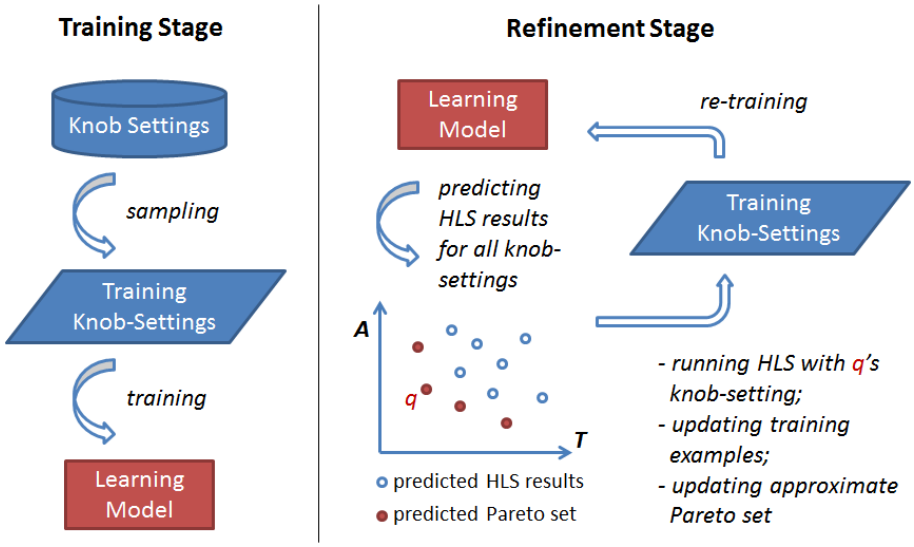


Figure 5.1: An iterative-refinement DSE framework [121; 134; 186; 197].

set by iteratively refining $\tilde{H}$ and the current Pareto approximation (the *refinement* stage in Lines 9–16).

Clearly, the effectiveness of the iterative-refinement framework relies on the accuracy of $\tilde{H}$. Intuitively, guided by a highly-accurate $\tilde{H}$, the refinement stage could search in the right space. However, spending too many HLS runs in the training stage while aiming for highly-accurate $\tilde{H}$, may not necessarily guarantee the final Pareto optimality, since the total number of HLS runs is limited to a given budget b. We approach this dilemma by suggesting a more accurate learning model for HLS in Section 5.3 and a more effective training scheme that is beneficial for general models in Section 5.4. Moreover, in Section 5.5, we propose novel algorithms to improve the scalability of the framework. All our algorithmic findings are inspired by recent advancements of machine-learning theory.

5.3 Learning-Model Selection

We examined nine advanced learning models for predicting the DFT's area and effective latency. These models include Gaussian Process Regression (GPR) [153], Multivariate Adaptive Regression Splines (MARS) [71], Regression Tree (RT) [26], Boosted Regression Tree (BRT) [72], Random Forest (RF) [25], Neural Network (NN), Radial Basis Function Network (RBF) [29], Support

Algorithm 5 Iterative-Refinement Framework

Input: HLS tool H, HLS budget b, input design D

Output: Approximate Pareto set Λ

1: Let $\mathcal{K}$ be the knob-setting space of D
2: Let $\tilde{H}$ be a learning model
3: Let $\tilde{\mathcal{K}} \subset \mathcal{K}$ be a training set
4: Let $S = \phi$ be the set of all HLS results
5: **/** Training Stage **/**
6: synthesize all $\tilde{\mathbf{k}} \in \tilde{\mathcal{K}}$; add the results to $\mathcal{S}$
7: remove $\tilde{\mathcal{K}}$ from $\mathcal{K}$
8: train $\tilde{H}$ by $(\tilde{\mathcal{K}}, \mathcal{S})$
9: **/** Refinement Stage **/**
10: $\Lambda \leftarrow$ current Pareto approximation of $\mathcal{S}$
11: **for** $i = |S| + 1 \rightarrow b$ **do**
12: $\quad \tilde{\mathcal{Q}} \leftarrow$ predicted HLS results $\forall \mathbf{k} \in \mathcal{K}$ by $\tilde{H}$
13: $\quad$ pick one $q \in \tilde{\mathcal{Q}}$ for HLS; add q's result to $\mathcal{S}$
14: $\quad$ move q's knob setting from $\mathcal{K}$ to $\tilde{\mathcal{K}}$
15: $\quad$ retrain $\tilde{H}$ by $(\tilde{\mathcal{K}}, \mathcal{S})$
16: $\quad \Lambda \leftarrow$ current Pareto approximation of $\mathcal{S}$
17: **end for**

Vector Regression (SVR) [64], and Transductive Regression (TR) [45][2]. They cover non-linear non-parametric models (GPR and MARS), tree-based models (RT, BRT, and RF), network-based models (NN, RBF, and SVR), and models that know all the testing instances in advance (TR).

Fig. 5.2 shows the prediction accuracy of these models that were trained on randomly-sampled training sets[3]. The results show that RF is consistently more accurate than GPR, which was previously reported as the best model for processor simulators and IP generators. The adoption of

[2]All these models are publicly available in separate R packages [149], except TR [45], which we implemented in R by ourselves. Notice that we finely tuned all these models via *model selection* [162] to achieve the best accuracy.

[3]Throughout this chapter, all the DFT results involving randomized procedures are the average results obtained from 100 trials.

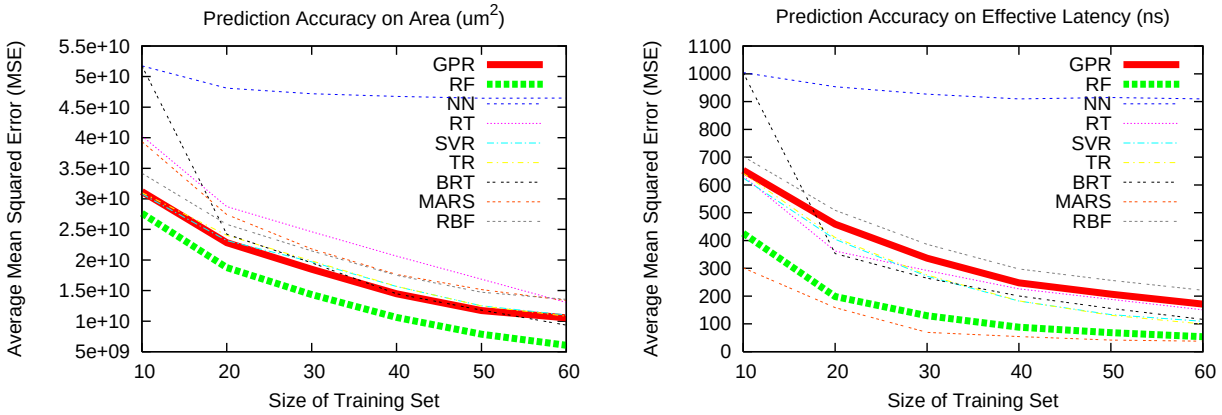


Figure 5.2: Learning-model accuracy for predicting DFT area (left) and effective latency (right). The training sets are randomly sampled.

RF for HLS instead of GPR brings the following benefits:

- The tree-based RF model can easily handle binary decisions by introducing a node with two branches separating the two decisions. HLS knobs which provide binary choices are common, e.g., function inlining for a function call, state insertion for a certain edge in the control/data flow graph, array implementation in either registers or memory, and others. In this regard, GPR assumes that any combination of knob variables follows a joint Gaussian distribution, which is less effective for modeling the HLS knobs.

- RF is an ensemble model consisting of multiple regression trees [25]. Given a training set, the set is internally and randomly partitioned for training individual trees. Then, the final prediction is made by collective vote from the individual trees. The two steps combined are capable of minimizing both the generalization error and prediction variance. A recent study in the machine learning literature also shows the superior accuracy of RF, especially for *high dimensional* data [33].

- We observe that the CPU time required for training and prediction with RF is around 50% less than that with GPR. Moreover, the internal partitioning-then-training scheme by nature makes RF suitable for running on multi-core machines. From an implementation viewpoint, this is another advantage of adopting RF.

Assuming that the learning model $\tilde{H}$ is either GPR or RF, we face the following decision: how to select a best knob-setting for next HLS in the refinement stage, i.e., Line 13 in Algorithm 5. For GPR, the prediction of a design objective consists of a mean and a variance. The mean represents the predicted value and the variance suggests the uncertainty of GPR about the prediction. Therefore, as suggested in [197], a *predicted* Pareto set $\tilde{\Lambda}$ is first extracted from the predicted QoR space $\tilde{\mathcal{Q}}$ (i.e., Line 12 in Algorithm 5), and then the element in $\tilde{\Lambda}$ with the max variance (uncertainty) across all objectives is selected for the next HLS. On the other hand, for RF, since the prediction uncertainty is minimized by RF's collective-vote scheme, we can randomly pick one element in $\tilde{\Lambda}$ for next HLS.

5.4 Transductive Experimental Design

In the prior work [121; 197], the training set $\tilde{\mathcal{K}}$ is randomly sampled from the knob-setting space $\mathcal{K}$ (Line 3 in Algorithm 5). Alternatively, we introduce *transductive experimental design* (TED) [190], that aims for selecting *representative* as well as *hard-to-predict* $\tilde{\mathcal{K}}$, in order to effectively train the learning model for predicting $\mathcal{K}$. *Note that TED assumes no priori knowledge about the learning model and should therefore be beneficial for any model.*

Assume that overall we have n knob settings ($|\mathcal{K}| = n$), from which we want to select a training subset $\tilde{\mathcal{K}}$ such that $|\tilde{\mathcal{K}}| = m$. In order to minimize the prediction error $H(\mathbf{k}) - \tilde{H}(\mathbf{k})$ for all $\mathbf{k} \in \mathcal{K}$, TED is shown to be equivalent to the following problem [190]:

$$\begin{aligned} \max_{\tilde{\mathcal{K}}} \quad & T\left[\mathcal{K}\tilde{\mathcal{K}}^\top(\tilde{\mathcal{K}}\tilde{\mathcal{K}}^\top + \mu\mathbf{I})^{-1}\tilde{\mathcal{K}}\mathcal{K}^\top\right] \\ \text{s.t.} \quad & \tilde{\mathcal{K}} \subset \mathcal{K}, |\tilde{\mathcal{K}}| = m, \end{aligned} \tag{5.1}$$

where $T[\,]$ is the matrix trace and $\mu > 0$ is a given regularization parameter [162]. The solution to the problem can be interpreted as follows. It tends to find *representative* data samples $\tilde{\mathcal{K}}$ that span a linear space to retain most of the information of $\mathcal{K}$ [190].

Equation 5.1 corresponds to TED sampling for *linear* regression. For *non-linear* regression, TED can be extended by a kernel function

$$f(\mathbf{u}, \mathbf{v}) = \theta(\mathbf{u}) \cdot \theta(\mathbf{v}),$$

where $\mathbf{u}, \mathbf{v} \in \mathcal{R}^d, \theta : \mathcal{R}^d \rightarrow \mathcal{F}$ is a function mapping from the knob-setting space to a feature

Algorithm 6 Sequential TED

Input: Set $\mathcal{K}$ of n knob settings, training-set size m

Output: Training set $\tilde{\mathcal{K}}$

1: $\mathbf{F} \leftarrow \mathbf{F}_{\mathbf{k},\mathbf{k}}$

2: $\tilde{\mathcal{K}} \leftarrow \phi$

3: **for** $i = 1 \rightarrow m$ **do**

4: select $\mathbf{k}_i \in \mathcal{K}$ with the largest $||\mathbf{F}_{\mathbf{k}_i}||^2/(f(\mathbf{k}_i, \mathbf{k}_i) + \mu)$, where $\mathbf{F}_{\mathbf{k}_i}$ and $f(\mathbf{k}_i, \mathbf{k}_i)$ are $\mathbf{k}_i$'s corresponding column and diagonal entry in current $\mathbf{F}$

5: add $\mathbf{k}_i$ to $\tilde{\mathcal{K}}$,

6: $denom \leftarrow f(\mathbf{k}_i, \mathbf{k}_i) + \mu$

7: $(\mathbf{F})_{jk} \leftarrow (\mathbf{F})_{jk} - \frac{(\mathbf{F})_{ji}(\mathbf{F})_{ki}}{denom}, \forall 1 \leq j, k \leq n$

8: **end for**

space [162]. Specifically, we use the Gaussian [190]:

$$f(\mathbf{u}, \mathbf{v}) = e^{\frac{||\mathbf{u}-\mathbf{v}||^2}{2\sigma^2}},$$

where σ is a given non-zero constant. Then, kernelized TED can be expressed as

$$\begin{aligned} \max_{\tilde{\mathcal{K}}} \quad & T\left[\mathbf{F}_{\mathbf{k}\tilde{\mathbf{k}}}(\mathbf{F}_{\tilde{\mathbf{k}}\tilde{\mathbf{k}}} + \mu\mathbf{I})^{-1}\mathbf{F}_{\tilde{\mathbf{k}}\mathbf{k}}\right] \\ \text{s.t.} \quad & \tilde{\mathcal{K}} \subset \mathcal{K}, |\tilde{\mathcal{K}}| = m, \end{aligned} \tag{5.2}$$

where the matrix entries $(\mathbf{F}_{\mathbf{k}\tilde{\mathbf{k}}})_{ij}$ equals $f(\mathbf{k}_i, \tilde{\mathbf{k}}_j)$, $(\mathbf{F}_{\tilde{\mathbf{k}}\tilde{\mathbf{k}}})_{ij}$ equals $f(\tilde{\mathbf{k}}_i, \tilde{\mathbf{k}}_j)$, $(\mathbf{F}_{\tilde{\mathbf{k}}\mathbf{k}})_{ij}$ equals $f(\tilde{\mathbf{k}}_i, \mathbf{k}_j)$, vectors $\mathbf{k}_i, \mathbf{k}_j$ are the row and column vectors of $\mathcal{K}$, and vectors $\tilde{\mathbf{k}}_i, \tilde{\mathbf{k}}_j$ are the row and column vectors of $\tilde{\mathcal{K}}$.

Unfortunately, both TED problems (Equations 5.1 and 5.2) are proven to be NP-hard [190]. Therefore, to solve Equation 5.2, we apply an efficient greedy algorithm called *sequential TED* [190]. The algorithm (given as Algorithm 6) can be interpreted as follows. Once a best $\mathbf{k}_i \in \mathcal{K}$ is selected (Line 4), the kernel matrix $\mathbf{F}$ is updated (Line 7) to represent the residual knob settings, so that the next selection would be picked among those under-represented by previously selected knob settings. In other words, the algorithm tends to select the knob settings that cannot be well represented by the already selected ones, i.e., to favor potentially *hard-to-predict* knob settings if they are not selected as training examples.

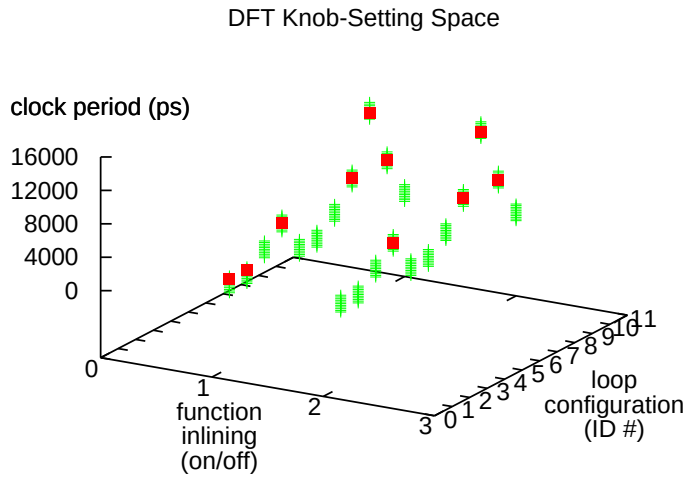


Figure 5.3: Training-set sampling by Transductive Experimental Design (TED).

Fig. 5.3 illustrates the TED sampling results of the DFT knob-setting space. We follow the suggestion in [190] to set $\mu = 0.1$ for Algorithm 6 and $\sigma = 0.1$ for the Gaussian kernel. As a result, a training set of 10 knob settings (red squares) is selected from the 242 candidates (green pluses). We can see that TED indeed selects *representative* samples by distributing its selections in the whole space, without having dense clusters. Besides, TED distributes 6 out of the 10 selections to the subspace where the loop-configuration ID number is greater than 6, i.e., a subspace where candidates are sparser than its counterpart. The loop configurations are generated by varying one loop-manipulation knob for consecutive configurations. Hence, the knob settings (green pluses) that are close to each other are very likely to produce similar HLS QoR. Based on this assumption, TED considers the samples in the sparse subspace *hard-to-predict*, thereby selecting more samples from the sparse subspace.

Fig. 5.4 shows the prediction accuracy of the combination of random/TED sampling with GPR/RF models. Starting from the sample size of 20 knob-settings for predicting area and 30 knob-settings for predicting effective latency, TED indeed reduces the prediction error for GPR and RF, respectively (see the dashed vs. solid lines). On the other hand, for very small training-set size, random sampling is helpful to escape from local optimal where the sequential-TED algorithm could be trapped. In general, TED is effective for sampling good training sets for any learning models. We also remark that with the aid of TED sampling, the prediction error of RF is still

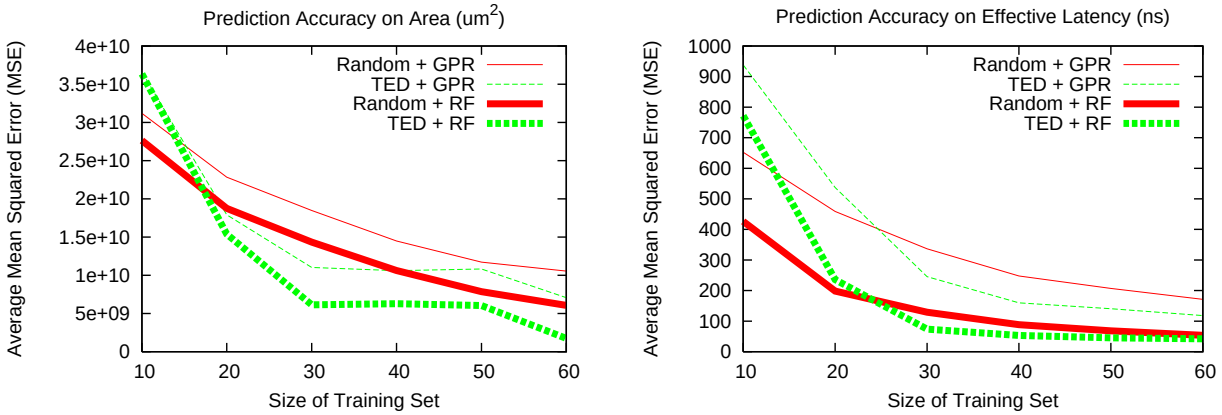


Figure 5.4: Learning-model accuracy for predicting DFT area (left) and effective latency (right). "Random" and "TED" indicates training-set sampling algorithms. "GPR" and "RF" are learning models.

consistently lower than that of GPR (see the dashed lines in Fig. 5.4).

Finally, similar to the distribution in the knob-setting space (Fig. 5.3), the corresponding distribution in the QoR space would not likely have dense clusters either. In fact, through our experimental results on designs with large design spaces, the TED-generated distribution in the QoR space empirically also corresponds to a good initial approximate Pareto set. On the other hand, blind random sampling not only can generate biasing training examples, but also often leads to an initial approximate Pareto set that is inferior to the one achieved by TED.

5.5 Scalable Design-Space Sampling

The size of knob-setting space $\mathcal{K}$ grows exponentially as explained in Section 5.1. Therefore, even if a very fast learning model $\tilde{H}$ is used, the exhaustive search in $\mathcal{K}$ (Line 12 in Algorithm 5) can incur noticeable computational overheads. From now on, we refer to Problem 3 with very large $\mathcal{K}$ as the *Extreme DSE-with-HLS Problem*, and refer to Problem 3 with tractable $\mathcal{K}$ as the *Basic DSE-with-HLS Problem*.

To solve the Extreme DSE-with-HLS Problem, we introduce the following theorem on *randomized selection* [162].

Theorem 3 (Ranks on Random Subsets)**.** *Let* $M = \{x_1, \ldots, x_\alpha\} \subset \mathcal{R}$*, and let* $\tilde{M} \subset M$ *be a*

Algorithm 7 Randomized TED

Input: Set $\mathcal{K}$ of n knob settings, training-set size m

Output: Training set $\tilde{\mathcal{K}}$

1: $\tilde{\mathcal{K}} \leftarrow \phi$

2: **for** $i = 1 \rightarrow m$ **do**

3: $\quad \tilde{M} = \{\mathbf{m}_1, \ldots, \mathbf{m}_{N_{rted}}\} \leftarrow$ a random subset of $\mathcal{K}$

4: $\quad \tilde{M} = \tilde{M} \cup \tilde{\mathcal{K}}$; $\mathbf{F} \leftarrow \mathbf{F}_{\mathbf{m},\mathbf{m}}$

5: $\quad \forall \mathbf{m}_i \in \tilde{\mathcal{K}}$, update $\mathbf{F}$ as Lines 6–7 in Algorithm 6

6: $\quad$ select $\mathbf{m}_i \in \tilde{M}$ as Line 4 in Algorithm 6; add $\mathbf{m}_i$ to $\tilde{\mathcal{K}}$

7: **end for**

random subset of size β. Then the probability that $\max \tilde{M}$ *is greater than γ elements of M is at least* $1 - (\frac{\gamma}{\alpha})^{\beta}$.

According to the theorem, if we draw a random subset $\tilde{M}$ of size 59 ($\beta = 59$), then $\max \tilde{M}$ would be greater than 95% ($\frac{\gamma}{\alpha} = 95\%$) elements of M with at least $1 - 5\% = 95\%$ confidence, since $(\frac{\gamma}{\alpha})^{\beta} = 0.95^{59} < 5\%$. Note that the sample size 59 is a *constant* for any large-sized M to achieve a 95% confidence in the 95% percentile range.

Based on Theorem 3, we propose a simple modification for selecting the next HLS knob-setting in Algorithm 5 (Lines 12–13): draw a random subset $\tilde{M} \subset \mathcal{K}$ of size N_{next}, and then pick the $q \in \tilde{M}$ with the smallest

$$\tilde{H}_{\mathcal{A}}(q) + \tilde{H}_{\mathcal{T}}(q), \tag{5.3}$$

where $\tilde{H}_{\mathcal{A}}(q)$ and $\tilde{H}_{\mathcal{T}}(q)$ are the predicted area and effective latency of q, respectively. Clearly, Theorem 3 also applies to selecting a minimum element. Consequently, we pick the best $q^* \in \tilde{M}$ for HLS, based on the cost function defined as Equation 5.3, which predicts the aggregated QoR of any $q \in \tilde{M}$. Note that to set $N_{next} = 59$ should suffice to achieve a very good approximation even for a very large $\mathcal{K}$, as inferred from Theorem 3.

Consider again the prohibitively large size of $\mathcal{K}$ of the Extreme DSE-with-HLS Problem. This makes even the training-set sampling by TED not scalable because an $O(|\mathcal{K}|^2)$ matrix computation is required in the sequential-TED algorithm. To address this scalability issue, inspired again by Theorem 3, we propose the *randomized TED* algorithm (as Algorithm 7). The main idea is that in

Table 5.1: Comparison of DSE Methods

Method	Reference	Learning Model	Training-Set Sampling	Next-HLS Selection
`baseline`	[197; 198]	GPR	random sampling	exhaustive search
`basic`	Section 5.3	RF	random sampling	exhaustive search
`basic-ST`	Section 5.4	RF	sequential TED	exhaustive search
`extreme`	Section 5.5	RF	random sampling	randomized selection
`extreme-RT`	Section 5.5	RF	randomized TED	randomized selection

each iteration we draw a random subset $\tilde{M} \subset \mathcal{K}$ of size N_{rted} and add previously selected samples to $\tilde{M}$. Subset $\tilde{M}$ is now treated as the $\mathcal{K}$ in the sequential-TED algorithm. We update the kernel matrix as if the previously selected samples are selected again in this iteration. Then, we follow the same criterion as in the sequential-TED algorithm to select the best residual $\mathbf{m} \in \tilde{M}$, and we iterate until m samples are selected. Overall, our algorithm utilizes the randomized-selection scheme to reduce the computational cost, while still preserving the principle of TED.

5.6 Case Study on DFT

We continue our DFT case study to compare the four DSE methods that we have presented, with respect to the existing method proposed in [197; 198], which we denote as `baseline`. Table 5.1 summarizes the five methods. We call `basic` our method that utilizes RF as the learning model, random sampling for selecting training knob-settings, and exhaustive search for selecting the next HLS knob-setting. We call `basic-ST` the variant of `basic` that utilizes sequential TED for selecting training knob-settings. Then, we refer to our method that utilizes RF as the learning model, random sampling for selecting training knob-settings, and randomized selection for selecting the next HLS knob-setting, as `extreme`. And finally we call `extreme-RT` the variant of `extreme` that utilizes randomized TED for selecting training knob-settings.

For each DSE method, we prepared training sets of size $m \in \{10, 20, \ldots, 60\}$. For each value of m, we set a HLS budget $\in \{m + 10, m + 20, \ldots, 120\}$ and we tracked the average ADRS that each method can achieve, with the exact Pareto set as a reference set. The HLS budget was capped at 120, i.e., less than 50% of DFT's knob-setting space. Across all training-set sizes, each method

Table 5.2: Number of HLS Runs to Find the Exact Pareto Set

Training-Set Size	10	20	30	40	50	60
`baseline`	120	NA	NA	NA	NA	NA
`basic`	115	113	NA	NA	NA	NA
`basic-ST`	112	91	100	100	109	119

should have $11 + 10 + 9 + 8 + 7 + 6 = 51$ ADRS records.

5.6.1 Basic DSE-with-HLS Results

For the Basic Problem, we compare three methods: `basic`, `basic-ST`, and `baseline`.

Table 5.2 summarizes the total number of HLS runs that each method requires to find the exact Pareto set (i.e., ADRS = 0). We can see that `basic-ST`, which utilizes TED to sample training sets, is the only method that can find the exact Pareto set within 120 HLS runs for any training-set size. Besides, for any training-set size, both `basic` and `basic-ST`, which adopt the RF learning model, outperform `baseline`, which adopts GPR instead. These results confirm that a more accurate learning model with a more effective training-set sampling scheme (in our case, RF with TED) indeed facilitates the convergence of the iterative-refinement framework. Also note that the best training-set size happens at 20, followed by 30 and 40. This result suggests that `basic-ST` only requires small training sets to achieve its best performance, because TED can judiciously select those *representative* and *hard-to-predict* knob-settings as training examples.

As explained in Section 5.1, in real applications of DSE with HLS it is infeasible to search for the exact Pareto set. Therefore, we now examine the performance of the three DSE methods given different HLS budgets. For HLS budget $b \in \{20, 30, \ldots, 120\}$, Table 5.3 summarizes the best DSE method, its average ADRS, and the training-set size it requires. We can observe that for smaller budgets ($20 \leq b \leq 50$), `basic` (abbreviated as `bs` in Table 5.3) can yield the minimum average-ADRS, requiring a training-set of size around 20. For greater budgets ($60 \leq b \leq 120$), `basic-ST` (abbreviated as `ST`) stands out to achieve the minimum average-ADRS, requiring a training-set of size around 30. For this DFT design, `basic-ST` starts with a higher ADRS because TED sampling does not particularly favor the knob-settings that can result in Pareto-optimal RTLs, whereas they

Table 5.3: `baseline` vs. `basic` (`bs`) vs. `basic-ST` (`ST`) given HLS budget b

Budget b	20	30	40	50	60	70	80	90	100	110	120
Best Method	bs	bs	bs	bs	ST	ST	ST	ST	ST	ST	ST
Average ADRS (%)	21.54	9.01	3.58	1.74	0.98	0.21	0.11	0.01	0.00	0.00	0.00
Training-Set Size	10	20	20	20	30	30	30	20–30	20–40	20–50	≥10

could be sampled at random instead. However, the advantage of random sampling vanishes when applied to large design spaces (which we will see later in Section 5.7). On the other hand, `basic-ST` can approach the exact Pareto set faster, due to the more accurate learning model resulting from the aid of TED. These two reasons combined explain why `basic` performs better for smaller HLS budgets for small designs, whereas `basic-ST` is better for greater budgets. Overall, both our methods, `basic` and `basic-ST`, outperform `baseline`.

5.6.2 Extreme DSE-with-HLS Results

For the Extreme Problem, we focus on two methods: `extreme` and `extreme-RT`.

First we compare `extreme` with `basic` to see how effective the approximation using randomized-selection can be. For `extreme`, we set $N_{next} = 59$ for drawing a random subset of size 59 to select the next knob-setting for HLS. As expected, most of the ADRS achieved by `extreme` are higher (worse) than those achieved by `basic`, but we find that the difference is small: for training-set size $10 \leq m \leq 30$, the max difference is less than 3.0%, and for $40 \leq m \leq 60$, the max difference is even less than 0.8%. These small differences show that the approximation using randomized-selection can be very effective. Moreover, since the ADRS differences are marginal, we observed that 41 out of 51 (80%) ADRSs achieved by `extreme` are also lower (better) than those achieved by `baseline`. For $N_{next} \in \{59, 79, 99\}$, we see no significant difference on ADRS: in fact, 46 out of 51 (90%) ADRS differences are less than 0.9%. Therefore, we set $N_{next} = 59$ by default for `extreme`.

Next, we show the results of `extreme-RT`. For `extreme-RT`, we fix its $N_{next} = 59$ and set $N_{rted} \in \{59, 79, 99\}$ for drawing a random subset of size N_{rted} to select a training knob-setting by the randomized-TED algorithm. Comparing with `extreme`, if we set `extreme-RT`'s N_{rted} to be 59, we observe that only 17 out of 51 (33%) ADRSs achieved by `extreme-RT` are lower (better). If we increase the N_{rted} to 79, then 30 out of 51 (59%) ADRSs are better. The improvement saturates

Table 5.4: `extreme` (`ex`) vs. `extreme-RT` (`RT`) given HLS budget b

Budget b	20	30	40	50	60	70	80	90	100	110	120
Best Method	`RT`	`RT`	`RT`	`RT`	`RT`	tie	`ex`	`ex`	`RT`	`RT`	tie
Average ADRS (%)	19.12	9.23	5.82	3.52	2.00	1.09	0.58	0.17	0.10	0.04	0.01
Training-Set Size	10	20	20	20	20	30/50	40	40	40	40	40

at $N_{rted} = 99$, where 31 out of 51 (61%) ADRSs are better. As a result, the randomized-TED algorithm works best with $N_{rted} = 99$. Therefore, we set $N_{rted} = 99$ by default for `extreme-RT`.

Finally, for the Extreme Problem with different HLS budgets, Table 5.4 summarizes the best method, its average ADRS, and the training-set size it requires. We see that `extreme-RT` (abbreviated as `RT`) almost outperforms `extreme` (abbreviated as `ex`) for every budget b. This result is different from the `basic` vs. `basic-ST` result (see Table 5.3), where `basic` (`basic-ST`) works better for smaller (greater) budgets. This is because the random-subset drawing used in the randomized-TED algorithm can alleviate the high initial ADRS otherwise caused by the sequential-TED algorithm. Overall, we find `extreme-RT` very effective for addressing the Extreme Problem, because our randomized-TED algorithm successfully avoids the exhaustive search in the full knob-setting space while still selecting *representative* and *hard-to-predict* knob-settings as training examples.

5.7 Additional Results on CHStone Suite

In this section, we present additional experimental results on large benchmarks from the CHStone suite [84]. CHStone includes eight benchmarks, each having a design space with more than 5K HLS configurations[4]: (1) adaptive differential pulse code modulation encoder for voice compression (ADPCM), (2) encryption function of advanced encryption standard (AES), (3) encryption function of data encryption standard (BLOWFISH), (4) double-precision floating-point sine function involving addition, multiplication, and division (DFSIN), (5) linear predictive coding analysis of global

[4] We do not include the MIPS benchmark (a simplified microprocessor) from CHStone because of its very limited design space. The MIPS benchmark is described as a simple switch statement that performs primitive operations based on the opcodes. It has none of the advanced features that are common to modern processors, such as branch prediction, hazard detection, out-of-order execution, simultaneous multi-threading, etc. In general we expect our DSE methods to be effective for exploring complex processor designs as well.

Table 5.5: Benchmark Characteristics

Benchmark	Application	Input Token	Output Token	#LoopC	#StateC	#FuncC	#ArrayC	#TotalC
ADPCM	DSP	2 16-bit int	1 16-bit int	1	8	128	4	20,480
AES	Security	64 16-bit int	32 16-bit int	18	1	64	2	11,520
BLOWFISH	Security	40 8-bit char	40 8-bit char	243	1	4	4	19,440
DFSIN	Arithmetic	1 64-bit int	1 64-bit int	8	2	256	1	20,480
GSM	DSP	160 16-bit int	8 32-bit int	27	1	32	4	17,280
JPEG	DSP	128 16-bit int	64 16-bit int	54	1	32	4	34,560
MOTION	DSP	32 8-bit char	32 8-bit char	9	4	32	1	5,760
SHA	Security	16 32-bit int	5 32-bit int	243	1	8	4	38,880

system for mobile communications (GSM), (6) JPEG image decompression (JPEG), (7) motion vector decoding of MPEG-2 (MOTION), and (8) transformation function of secure hashing algorithm (SHA).

Table 5.5 lists some important characteristics of the eight benchmarks (see [84] for more details). The benchmarks cover application domains from DSP, security, and arithmetic. For each benchmark, we also list its input/output token size, such that the effective latency of a benchmark corresponds to the total elapsed time for: reading an input token, processing the token according to the benchmark, and generating an output token.

5.7.1 Experimental Setup

We setup the benchmark design spaces as follows.

- *Loop Manipulation*: We consider loop breaking, partial/full unrolling with different numbers of loop bodies, and pipelining with different initiation intervals and pipeline stages for each major loop. Trivial loops such as initializing an array or copying values between two arrays are all completely unrolled. The total number of loop configurations is listed in column "#LoopC" in Table 5.5.

- *State Insertion*: We consider the insertion of extra states for the benchmarks that have few major loops in their source descriptions. The insertion is mainly for breaking long combinational paths in order to improve the circuit performance, meet the target clock period, or promote resource sharing among different clock cycles. The total number of state configura-

tions is listed in column "#StateC" in Table 5.5.

- *Function Inlining*: We either inline or do not touch a function for all the function definitions in a source description. Function inlining restructures the design hierarchy, which may affect synthesis QoR due to different degrees of resource-sharing opportunities [41; 80]. However, trivial functions such as functions that are only called once or twice are all inlined. The total number of function configurations is listed in column "#FuncC" in Table 5.5.

- *Array Configuration*: We either flatten an array with registers as memory elements or apply the default prototype-memory settings provided by the HLS tool. In general, flattened arrays allow parallel accesses to the array values at the cost of larger implementation area. Therefore, trivial (small) arrays are flattened by default. The total number of array configurations is listed in column "#ArrayC" in Table 5.5.

- *Target Clock-Period Setting*: For each benchmark we consider the five target clock periods in the set $\{0.8 \times T_{clk}, T_{clk}, 1.2 \times T_{clk}, 1.4 \times T_{clk}, 1.6 \times T_{clk}\}$, where T_{clk} is the synthesized clock period when configured with maximal loop unrolling, minimal state insertion, maximal function inlining, and maximal array flattening.

Given the above setups for each benchmark, its total number of HLS configurations is "#TotalC" = "#LoopC" × "#StateC" × "#FuncC" × "#ArrayC" × 5. Note that the maximum "#TotalC" could be even larger if the trivial configurations were also considered.

On the eight large benchmarks, we applied the best basic DSE method `basic-ST` and the best extreme DSE method `extreme-RT` against the `baseline` method. In light of the large design spaces, we set the number of training examples to be 30 for each of the DSE methods, and let each method run for additional 200 iterations at the refinement stage. Besides, for `extreme-RT`, we set the size of a random subset N_{next} for model evaluation per iteration to be 59 and the size of a random subset N_{rted} for generating training examples with randomized TED to be 199. Every result in the following discussions was averaged based on ten trials.

Since we cannot afford to exhaustively search the exact Pareto set for each benchmark, we take the approximate Pareto set that was collectively discovered by the three DSE methods as the *reference Pareto set*. In other words, the reference Pareto set is the Pareto set obtained from a collection of 690 HLS results because each DSE method submits 230 HLS jobs per DSE run.

Table 5.6: Comparison of DSE Method Convergence

	Initial ADRS (%)					# Refinement Iterations to Reach ADRS $\leq$ 2%				
Benchmark	Baseline	Basic-ST	Diff	Extreme-RT	Diff	Baseline	Basic-ST	Improvement	Extreme-RT	Overhead
ADPCM	12.15	7.09	-5.05	10.02	-2.13	168	88	1.91X	N/A	N/A
AES	15.48	13.59	-1.89	11.35	-4.13	79	35	2.26X	162	2.05X
BLOWFISH	10.20	5.90	-4.30	8.33	-1.88	102	39	2.62X	145	1.42X
DFSIN	14.38	7.86	-6.52	11.38	-3.00	178	63	2.83X	N/A	N/A
GSM	15.70	7.51	-8.19	10.94	-4.76	168	68	2.47X	N/A	N/A
JPEG	14.35	9.38	-4.98	13.05	-1.31	125	57	2.19X	182	1.46X
MOTION	13.20	9.90	-3.30	12.33	-0.88	86	38	2.26X	148	1.72X
SHA	22.11	11.92	-10.19	15.79	-6.32	125	55	2.27X	N/A	N/A
Average	14.70	9.14	-5.55	11.65	-3.05	122	55	2.20X	159	1.66X

5.7.2 Discussions

Fig. 5.5 plots the average ADRS (%) vs. the number of refinement iterations for the three methods on the eight benchmarks. The red, green, and blue curves represent the results from `baseline`, `basic-ST`, and `extreme-RT`, respectively. We observe that all the three DSE methods can refine the approximate Pareto set as the number of refinement iteration increases from 0 to 200. With less than 4% (230/5,760) of the design space explored, the room for Pareto-set refinement in terms of ADRS is 10% on average.

We extract the initial ADRSs and the number of refinement iterations required to achieve a target 2% ADRS from Fig. 5.5 and list them in Table 5.6. For the initial ADRSs, `baseline` generally starts with the worst (highest) ADRSs (with an average of 14.70%), outperformed by `extreme-RT` (average 11.65%), and further outperformed by `basic-ST` (average 9.14%). Therefore, `basic-ST` has an advantage of low initial-ADRS (we will discuss why in the next bullet) and `extreme-RT` is the second best in this regard.

Moreover, in terms of convergence speed, taking a 2% target ADRS for example, `baseline` can converge to the target in 122 refinement iterations on average. In contrast, `basic-ST` only needs 55 iterations on average for the convergence (a 2.20X improvement). On the other hand, `extreme-RT` can only converge for four of the eight benchmarks and, if it succeeds, it requires 1.66X more iterations on average than `baseline` ("N/A" in Table 5.6 denotes the failure of convergence).

Overall, `basic-ST` is a clear winner in terms of both the starting ADRS and the convergence speed. As for `baseline` and `extreme-RT`, the former converges faster but the latter starts with a lower ADRS. We continue to evaluate these three methods in greater detail with three pairwise

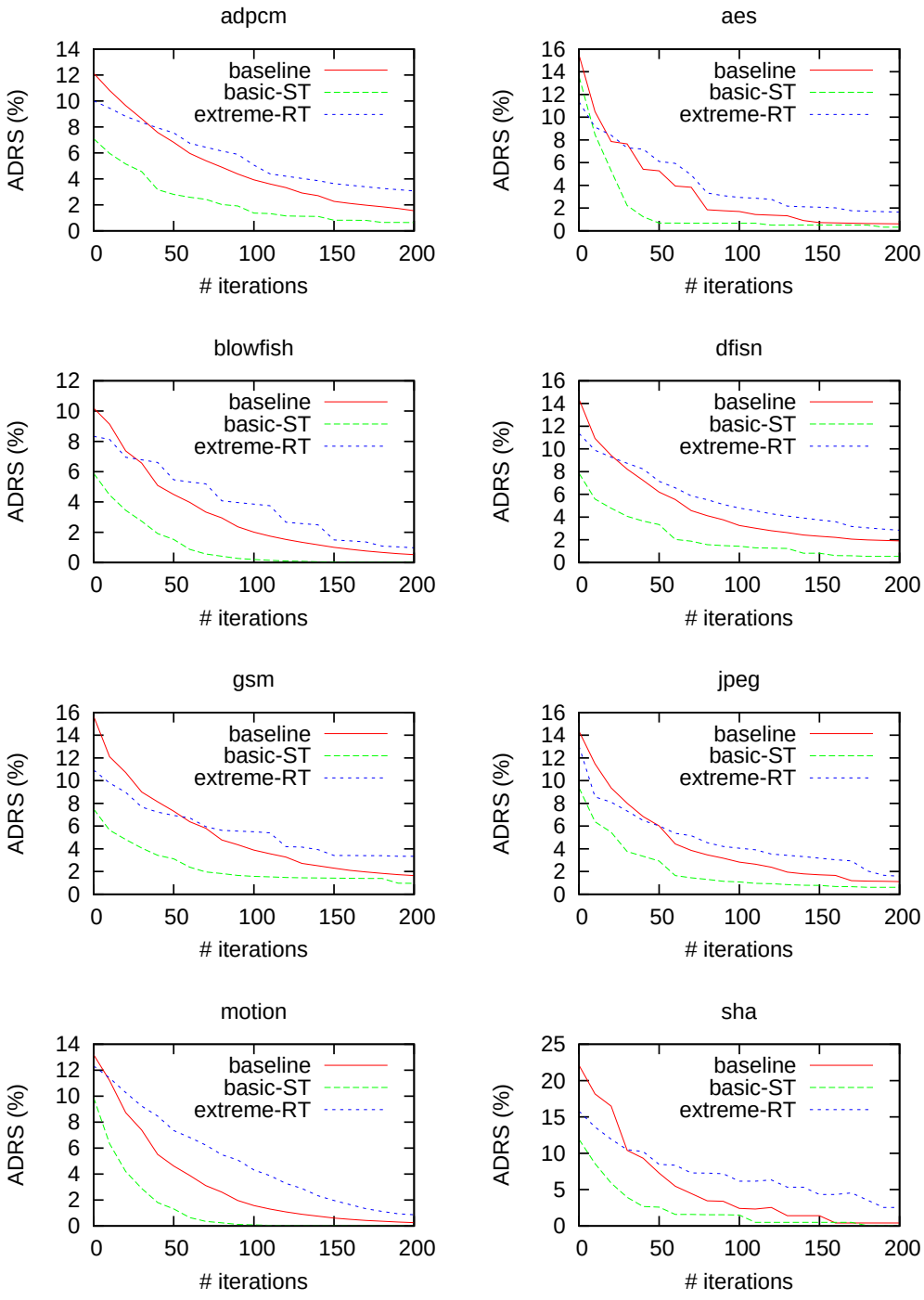


Figure 5.5: The average ADRS (%) vs. the number of refinement iterations for DSE methods `baseline`, `basic-ST`, and `extreme-RT`.

comparisons.

Comparison 1: `basic-ST` vs. `baseline`. In Fig. 5.5, across all the benchmarks, the `basic-ST` curves are always below the `baseline` curves, showing that `basic-ST` constantly works the best in terms of Pareto optimality. The optimality comes from *(i)* lower starting ADRSs (thanks to TED) and *(ii)* faster convergence speed because of higher model prediction accuracies (thanks to RF).

First, for large design spaces, we observe that random sampling cannot generate a good initial approximate Pareto set as it does in the DFT case. This is due to the limited sampling budget (30) versus the very large sample space (of size greater than 5K, i.e., the budget/size ratio is less than 0.6%). In contrast, the sample-space size was 242 for DFT (the ratio was 12.4%). However, using a training set with size larger than 5K$\times$12.4%=620 (i.e. to run more than 620 HLS jobs up front) requires too much CPU/memory resources to be practical. In this regard, `basic-ST`, which uses TED for generating representative training examples as the original goal, is also robust and practical for generating the initial approximate Pareto set to explore large design spaces.

Second, the prediction accuracy of RF (used in `basic-ST`) is constantly higher than GPR (used in `baseline`). Fig. 5.6(a) and Fig. 5.6(b) show the accuracies for predicting area and effective-latency, respectively. For area prediction on average, the *minimum*, *maximum*, *average*, and *standard deviation* of RF prediction errors are each better (lower) than those of GPR counterparts by 0.09%, 9.26%, 2.87%, and 1.98%, respectively. Similarly, for effective-latency prediction, the average improvements are 0.19%, 23.83%, 7.01%, and 5.34%, respectively (note the greater improvements compared with the prediction of area). The two results combined show the advantage of RF over GPR in terms of both quality and robustness.

We also observe that for both RF and GPR, their prediction error rates are higher in predicting effective latencies than predicting area. This is because the prediction of effective latency relies on the predictions of *both* the number of clock cycles and the synthesized clock period. Despite the difficulty, the advantage of RF over GPR is even more significant for predicting effective latencies.

Comparison 2: `extreme-RT` vs. `baseline`. In Fig. 5.5, the final ADRSs of the `extreme-RT` curves are higher (worse) than those of the `baseline` curves, even though the `extreme-RT` curves generally start with lower (better) ADRSs. The slower convergence of `extreme-RT` is due to the random subset sampling for efficient model evaluation, whereas the lower ADRS is due to the same reason we discussed in Comparison 1. Nevertheless, we want to note here that: *(i)* in a shorter DSE

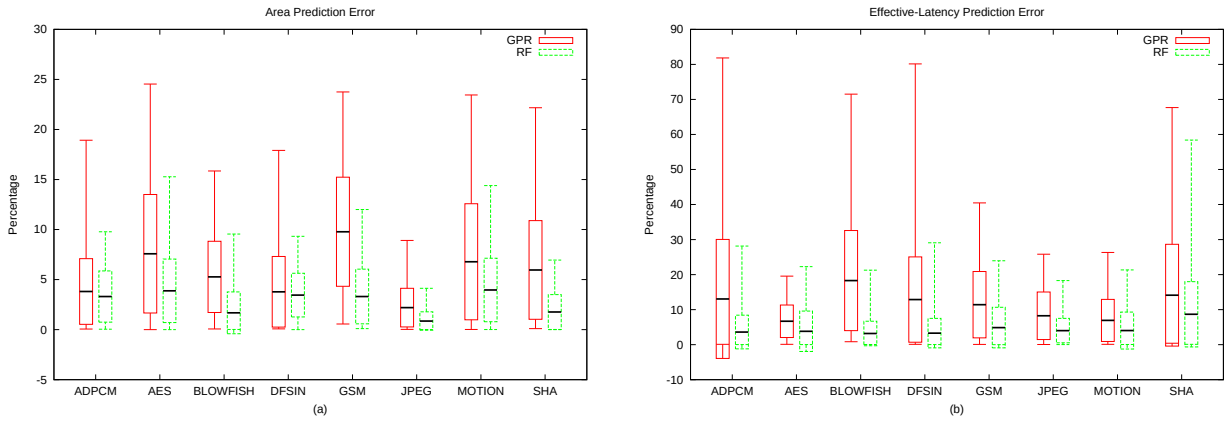


Figure 5.6: Model prediction error rates with GPR and RF for (a) area prediction and (b) effective-latency prediction. Each candlestick shows from top to bottom: the maximum, the average plus standard deviation, the average, the average minus standard deviation, and the minimum errors.

run (less than around 32 refinement iterations on average), the `extreme-RT`'s ADRS can still be better (lower) than the `baseline`'s ADRS, and *(ii)* in a long DSE run (200 iterations in this study), `extreme-RT` can still approximate the best `basic-ST` results by an average 2.06% difference in terms of the final ADRS. Moreover, `extreme-RT` offers unique advantages in saving computational time, which we discuss next.

Comparison 3: `basic-ST` vs. `extreme-RT`. Although `basic-ST` has a clear advantage in Pareto optimality, it may require substantial CPU time for the execution of TED and the exhaustive evaluation with RF per refinement iteration. Fig. 5.7(a) compares the CPU time required by TED (used in `basic-ST`) and randomized TED (or RTED, used in `extreme-RT`). Over all the benchmarks, TED requires 58 seconds to 3599 seconds (around 1 hour) with an average of 1165 seconds (around 20 minutes) for generating 30 training HLS scripts. On the other hand, RTED only needs 8 seconds to 34 seconds with an average of 15 seconds to do so (i.e. 93.31X faster).

Moreover, Fig. 5.7(b) compares the aggregated times (over 55 iterations[5]) for retraining the RF model plus predicting all or a subset of 199 unknown knob-setting QoRs. Recall that `basic-ST` predicts all unknown QoRs, whereas `extreme-RT` predicts a random subset of them. The average aggregated time for the exhaustive-prediction case (indicated by "RF (all)" in Fig. 5.7(b)) is 202 seconds. In contrast, the average aggregated time with random subsampling ("RF (199)" in

[5] With 55 refinement iterations on average, `basic-ST` can hit a target 2% ADRS as reported in Table 5.6.

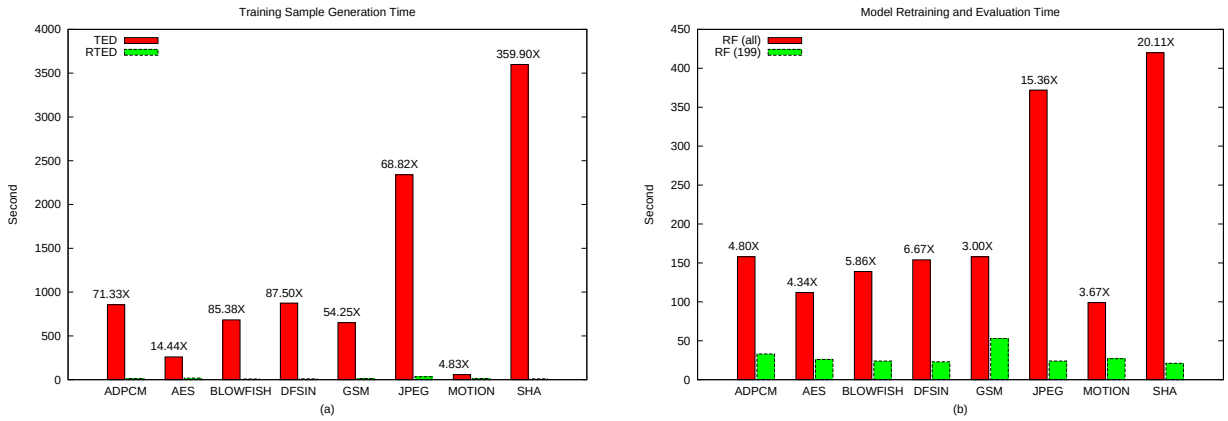


Figure 5.7: CPU time for (a) generating 30 training HLS scripts (without running HLS) and (b) retraining a learning model and evaluating the new model on all or 199 unseen HLS scripts, aggregated over 55 iterations. For each benchmark, the factor over each red bar shows the ratio of the height of the red bar over that of the green bar. The average factors are 93.31X and 7.98X for (a) and (b), respectively.

Fig. 5.7(b)) is only 29 seconds (i.e. 7.98X faster).

Note that although the CPU-time saving of RTED over TED is once per DSE run, the CPU-time saving of "RF (199)" over "RF (all)" will continue accumulating if the refinement goes beyond 55 iterations. As an additional note, although not shown in Fig. 5.7(b), the average aggregated time with GPR as the learning model is around 1.5X more than that of "RF (all)", further showing the efficiency of RF-based DSE methods.

Finally, we examine how much the CPU-time savings can contribute to the total elapsed time of a DSE run. Fig. 5.8 compares the total elapsed times with `basic-ST` (abbreviated as BST in the figure) and `extreme-RT` (abbreviated as ERT) using: (a) high-effort HLS and (b) low-effort HLS[6]. In Fig. 5.8(a), the total high-effort HLS runtime dominates the total elapsed time, so the sum of the *training*-time saving (previously shown in Fig. 5.7(a)) and the *retraining*-time saving (previously shown in Fig. 5.7(b)) is minor (2% on average) with respect to the total elapsed time. Alternatively speaking, to achieve the best DSE results using `basic-ST` with high-effort HLS, the runtime overhead

[6] Optimization effort is a common HLS-tool option for trading off between synthesis QoR and tool runtime. High HLS effort generally delivers better QoR at the cost of longer runtime. Our exploration results (Fig. 5.5, etc.) were obtained with high-effort HLS.

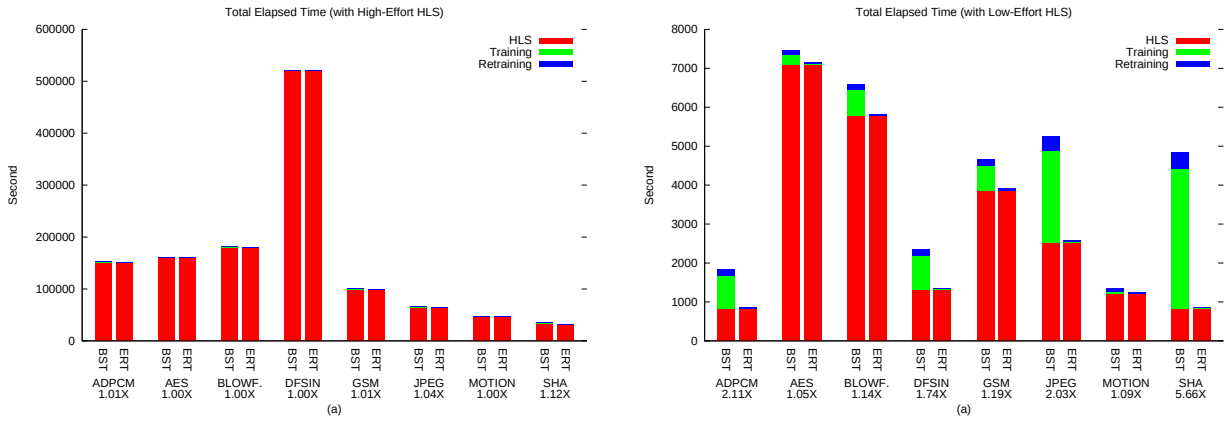


Figure 5.8: Total CPU time with (a) high-effort HLS and (b) low-effort HLS. For each benchmark, "BST" and "ERT" stand for the results obtained by `basic-ST` and `extreme-RT`, respectively. For each stacked bar, the "HLS" time includes 30 and 55 HLS runs at the training and the refinement stages, respectively; the "Training" and "Retraining" times are the CPU times shown in Fig. 5.7(a) and Fig. 5.7(b), respectively. Therefore, for the two stacked bars of each benchmark, the "HLS" time is constant, while the "Training" and "Retraining" times can vary. For each benchmark, the factor below the benchmark name shows the ratio of the height of the "BST" bar over that of the "ERT" bar. The average factors are 1.02X and 2.00X for (a) and (b), respectively.

is marginal (2% on average). On the other hand, in Fig. 5.8(b), the total low-effort HLS runtime can be comparable with the training and retraining time. Therefore, on average the total elapsed time with `extreme-RT` can be reduced by 2.00X compared with the time with `basic-ST`. In other words, to minimize the DSE runtime using `extreme-RT` with low-effort HLS, the runtime saving is significant (2X on average).

5.8 Remarks

DSE processes are essential for both final design optimization and early design evaluation. In general, the final optimization targets ultimate implementation QoR at the cost of acceptable DSE effort. On the other hand, the early evaluation aims for short turnaround time while the design specification, including its associated library and models, is still evolving. To switch between these two DSE modes, our tool LEISTER equipped with methods `basic-ST` and `extreme-RT` offers a

unique flexibility with the following recommended usages:

- *LEISTER-S or the standard mode.* LEISTER-S employs the `basic-ST` method, and set the HLS effort to standard or high, aiming for the best DSE results. This mode should be used for final accelerator implementations.

- *LEISTER-L or the lightweight mode.* LEISTER-L runs the `extreme-RT` method under the hood, and applies low-effort HLS, aiming for rapid DSE with short turnaround time. The lightweight mode should be used for early exploration on accelerator designs whose specifications are still under change, as it allows the system architect to estimate the accelerator's implementation costs and operating range.

We also note here that our DSE methods have been independently implemented and verified by Xydis *et al.* on another set of eight different DSP kernels [187]. The DSP kernels were synthesized as co-processors, while we targeted ASIC accelerators instead. Their study shows a clear ADRS advantage of the LEISTER methods over both the `baseline` and a local-search method, given the same budgets of HLS runs. Moreover, by relaxing the number of HLS runs per iteration from one (our setting) to a larger constant, the average ADRS can be further improved using spectrum analysis in each refinement iteration. This enhancement to the interactive refinement framework is complementary to our contributions, as we focus on the effective training scheme (TED), scalable sampling (randomized TED), and learning-model selection (RF).

Related Work. Due to the large solution (design) space, general multi-objective DSE algorithms rely on local-search techniques, e.g., Genetic Algorithms [53] or Simulating Annealing [171]. When applied with CAD tools, however, these algorithms require actual simulation/synthesis runs at every step to acquire solution qualities (i.e. the implementation QoR). Therefore, the total elapsed DSE runtime is still significant.

To reduce the DSE runtime, *Learning*-based methods were proposed to guide the DSE process by predicting solution qualities before running actual simulation/synthesis jobs (the earliest works are [18; 133; 134]). Compared with the local-search techniques, these methods can also discover better implementation QoR with fewer synthesis/simulation jobs. Among these learning-based methods, Beltrame *et al.* adopted an approach based on Markov Decision Processes, which intrinsically traverses an exponentially-growing state space [18]. Alternatively, Palermo *et al.* presented an effi-

cient DSE-with-processor-simulation framework based on *iterative refinement* [134], which has been further improved by several other researchers [121; 197; 198]. In particular, two recent studies by Mariani *et al.* [121] and Zuluaga *et al.* [197] independently reported that Gaussian Process [197], aka Kriging [121], was the most promising learning model, superior to Artificial Neural Network [133; 134] and other simple models. These results were obtained from DSE with *processor simulators* [121] or DSE with *IP generators* [197].

In the context of DSE with *HLS tools*, although specific techniques for incremental DSE [158] and loop-array exploration [142; 195] are emerging, *learning-based DSE methods remain popular for their general applicability and efficiency* [112; 161; 186; 187; 197]. Shafer and Wakabayashi were among the first to apply machine learning to the problem of general DSE-with-HLS [161]. They used a learning-model-assisted local-search approach that is more efficient than a pure Genetic Algorithm approach while still achieving a comparable Pareto set. On the other hand, in their method the model accuracy is not refined as in the iterative-refinement framework [134]. Several studies have shown that a refinement-based approach can lead to better DSE results than leaning-model-assisted local-search approaches [186; 187; 198]. An early refinement-based work for DSE with HLS was presented by Zuluaga *et al.* [197; 198]. Their work used Gaussian Process Regression as the underlying learning model and showed its effectiveness to explore the IP generation of DFT and Sorting Network. Another early related work was presented by Xydis *et al.* to explore co-processor designs [186] and was later improved by the same authors in [187]. Their latest work enhances the model-refinement scheme of the iterative-refinement framework [134] by spending a subset of non-Pareto-promising HLS jobs in addition to the Pareto-promising ones, in order to refine the learning-model accuracy more quickly. Ultimately this leads to faster convergence of the entire DSE process. The enhancement was validated by synthesizing co-processors for DSP applications.

Chapter 6

Parameter Tuning for Physical Synthesis

The previous chapter focuses on component-level design exploration with HLS. In this chapter, I present an industrial system that automates the DSE process for the logic and physical synthesis steps within an VLSI design flow. This system was successfully used during the design of an IBM high-performance 22nm server processor [182] that is currently in production and was crucial for achieving timing closure and meeting power targets. The original system (Section 6.2) was developed at IBM Research. Through my internship with IBM, I enhanced the system with new algorithms (Sections 6.3.2 and 6.3.3) based on the principles of Supervised Design-Space Exploration (SDSE). The SDSE enhancements achieved ∼20% better circuit performance than the original system on the design of a diverse set of components from the production processor.

6.1 Problem Description

Synthesis programs continue to become more sophisticated and provide numerous parameters that can significantly influence design quality. As an example of the wide design space available from modifying synthesis parameters, Figure 6.1 shows the scatter plot of achievable design points for a portion of a synthesized floating point multiplier, which is an example of a macro in our context[1].

[1] The term macro, used throughout this chapter, refers to a circuit partition which may range in size from 1K to 1M gates. Macros are synthesized and then integrated into the overall chip. A large industrial processor contains

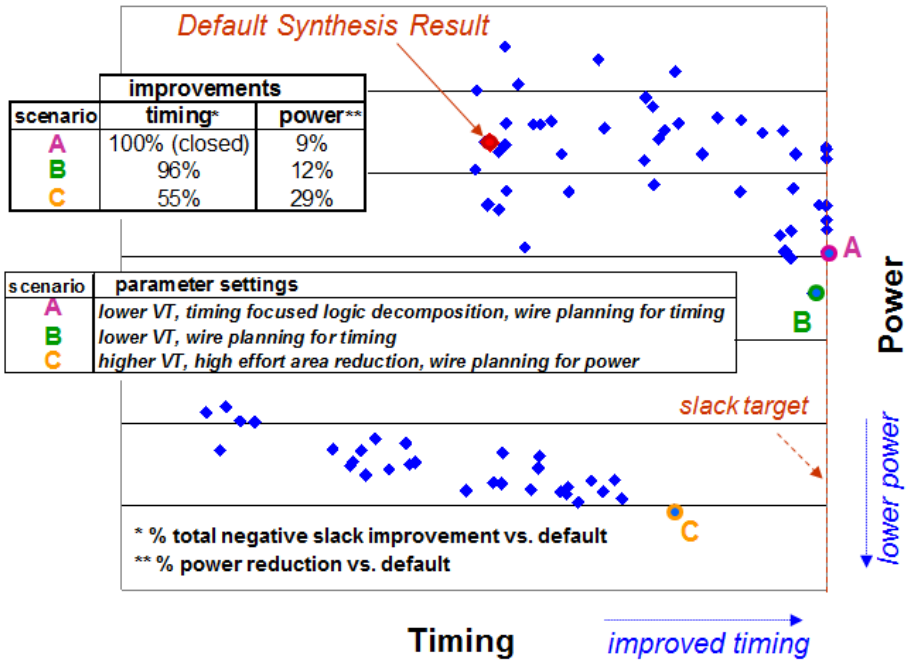


Figure 6.1: An example of the available design space by modifying synthesis parameters. The macro in this specific case is a portion of a floating point multiplier.

Each point denotes the timing and power values achieved by tuning the input parameters of the synthesis program. The ultimate goal of this process is to reach timing closure at the lowest achievable power. Quite often the default values for the parameters are not ideal for a specific macro, which would benefit instead from parameter customization. The figure also highlights three scenarios (A, B, and C) along the Pareto set. These scenarios show the available trade-offs between timing closure and power reduction, e.g., point A closes timing with a 9% power reduction, whereas point C improves timing by 55% with a 29% power reduction. These points along the Pareto set provide a number of potential steps towards the ultimate goal, depending on the additional techniques at the designer's disposal beyond parameter tuning. This example of a relatively simple macro underscores how significantly the parameters settings can affect a design.

However, the high flexibility and sophistication of advanced synthesis tools increases their complexity and makes navigating the design space difficult, sometimes non-intuitive, for their users. Since the number of parameters for synthesis tools can be in the thousands and synthesis runs may

100's of macros.

take several hours, or even days, exhaustive parameter-tuning is typically infeasible. Furthermore, while Figure 6.1 portrays the design space of two metrics, designers often need to consider many more metrics, sometimes dozens, when evaluating the quality of a synthesis scenario. In summary, efficient DSE using advanced synthesis tools is increasingly challenging for even experienced professional designers and daunting for novice designers.

To efficiently and fully utilize the optimization potential of advanced synthesis tools, IBM Research built a system for managing the design space exploration process that is called the Synthesis Tuning System (SynTunSys). The system amounts to a new level of abstraction where the human designer offloads the DSE task to automated tools. SynTunSys has been fully implemented and deployed during the design of an IBM high-performance 22nm server processor. A systematic study of ∼200 macros during the actual processor design cycle shows that SynTunSys provides, on average, 36% improvement in total negative slack and 7% power reduction. Further, it improves the macro internal negative slack by 60%, on average. *Since the tapeout of the processor design, SynTunSys has been enhanced with SDSE algorithms in preparation for future processor design.*

6.2 IBM Synthesis Tuning System

In order to perform automatically the tuning of synthesis parameters, SynTunSys constructs synthesis scenarios, runs the synthesis jobs, analyzes the results, and iteratively refines the solutions. The system employs parallel and iterative black-box search techniques to explore a design space which has been reduced by prior expert knowledge. In doing so, SynTunSys can efficiently scale to use the available resources in a compute cluster.

Figure 6.2 shows the SynTunSys architecture. We employ this system to tune the logical and physical synthesis steps in a VLSI design flow. Although the architecture of the system is general and could be applied to earlier steps (e.g. HLS) or later steps (e.g. routing) in a flow, we focus on the logical and physical synthesis steps because they have more accurate timing/area/power models (wrt HLS) and offer a stronger return on investment (ROI) of the compute resources (wrt routing). SynTunSys consists of a main tuning loop that, at each iteration, involves running multiple synthesis scenarios in parallel, monitoring the jobs in flight, analyzing the results of the jobs, and making a decision for the next set of scenarios. A second background loop archives the results of all runs

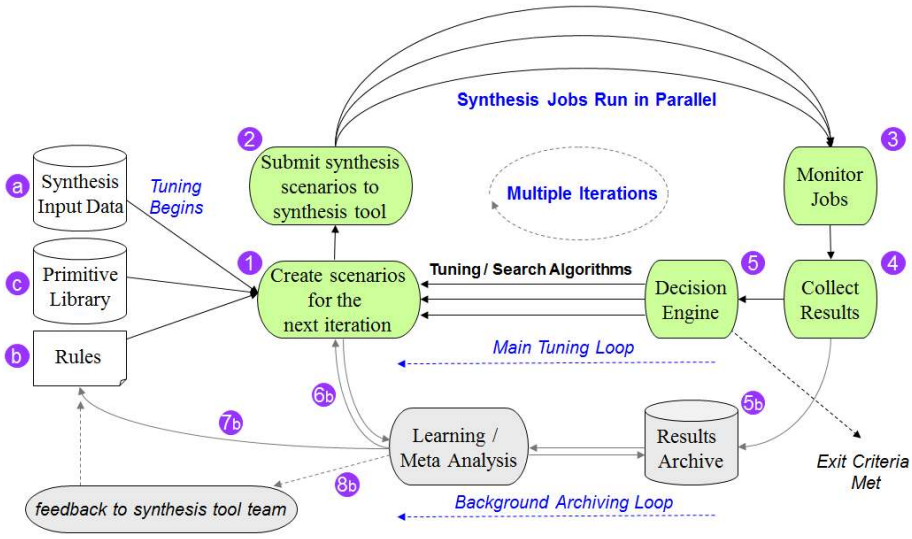


Figure 6.2: Architecture of the SynTunSys process. The program employs a parallel and iterative tuning process to optimize macros.

from all macros, users, and projects. As described later in the chapter, this archive is a database that can be mined for historical trends across multiple macros and to provide feedback in terms of the performance of synthesis parameters.

The SynTunSys process begins at Point (1) in Figure 6.2 where the initial synthesis scenarios are generated based on the following input data: (a) standard input data for circuit-level synthesis, which typically include an RTL description, a physical abstract view providing macro boundaries, pin locations, and timing assertions; (b) a "SynTunSys Rules" file specifying the primitives for the design space exploration and a cost function describing the optimization goals, among other options; and (c) a library of primitives that contains the detailed definitions of all potential exploration options. More details on (b) and (c) are provided later in this section. At Point (2) in Figure 6.2, multiple synthesis jobs are submitted in parallel to a compute cluster by issuing batch calls to the underlying synthesis tool. Following this submission, SynTunSys starts a monitor process that tracks the synthesis jobs (Point (3)). When either all jobs complete, or a large enough fraction of jobs complete, or a time limit is reached, the monitor process initiates the collection and analysis of the results of the parallel synthesis jobs (Point (4)). Based on the collected results, a decision algorithm at Point (5) creates a new set of scenarios (synthesis parameter settings) to be run in the next iteration. These new jobs begin with the initial input data and are again run in parallel, i.e. the

Table 6.1: An example library of primitives

Primitive Name	Primitive Description
`restruct_a`	Logic restructuring to reduce area
`restruct_t`	Logic restructuring to improve timing
`rvt_lvt10`	Native RVT, allow 10% LVT
`rvt_lvt50`	Native RVT, allow 50% LVT
`vtr_he`	High effort VT recovery
`area_he`	High effort area reduction
`wireopt_t`	Wire optimization for timing
`wireopt_c`	Wire optimization for congestion

next iteration does not modify the output of the prior iteration, but resynthesizes the macro from the beginning. The process iterates as it attempts to improve upon results until an exit criteria is met, e.g., the maximum number of user specific iterations is reached or the DSE algorithm exhausts the targeted search space.

6.2.1 Initial Design Space Reduction

Parameters. Advanced synthesis tools can have a vast number of tunable parameters making an exhaustive design space search infeasible. In our specific case, the synthesis tool we employ has over 1000 parameters. These parameters span the logic and physical synthesis space and the control settings for modifying the synthesis steps, such as: logic decomposition, technology mapping, placement, estimated wire optimization, power recovery, area recovery, and/or higher effort timing improvement. The parameters also vary in data type (Boolean, integer, floating point, and string). Considering that an exhaustive search of only 20 Boolean type parameters leads to over one million combinations, it is clear that intelligent search strategies are required.

Primitives. To reduce the ∼1000 multi-valued parameter space up front, we recast this DSE problem to have a ∼100 Boolean parameter space. This design space reduction involves a one-time offline effort to create a *library of primitives*. A primitive contains one or more synthesis parameters set to specific values. Table 6.1 shows an example of a small primitive library. The actual primitive

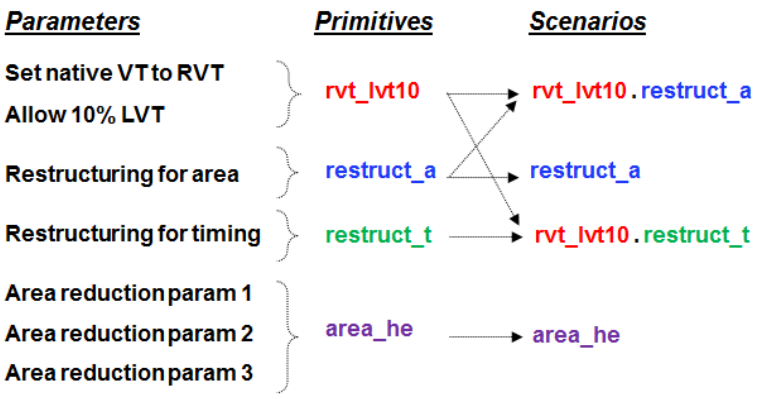


Figure 6.3: Illustration of the interaction of parameters, primitives, and scenarios. Primitives can consists of one or more parameters.

library in our case consists of ∼300 primitives. In general, a primitive targets a singular action and we use a short name for the primitive representative of the action. Thus, a SynTunSys primitive is a *binary* decision, whereas setting one or more parameters individually may require many more decisions.

Scenarios. SynTunSys ultimately creates scenarios consisting of one or more primitives that each contain one or more parameters. These scenarios are then run by SynTunSys as batch calls to the underlying synthesis tool. Figure 6.3 shows an example of the interaction between parameters, primitives, and scenarios. Scenarios are assembled by selecting one or more primitives and primitives contain one or more parameters set to specific values. The construction of scenarios is not trivial, motivating the need for intelligent decision algorithms (see Section 6.3). In particular, some primitives are complementary, e.g., `restruct_t` and `wireopt_t` for timing optimization, which however may require additional `wireopt_c` to compensate for routability. Other primitives are non-complementary, such as `restruct_t` and `area_he`; the former may upsize standard cells for larger driving strength or duplicate downstream cells for smaller capacitance load, while the latter may do the opposite changes for maximal area reduction. In practice, the optimal scenarios are macro-specific with respect to a set of weighted design objectives. In fact, a universal scenario recipe should not exist, as the default synthesis settings generally yield sub-optimal results (see Figure 6.1).

The library of primitives was created based on the aggregate expert knowledge of many de-

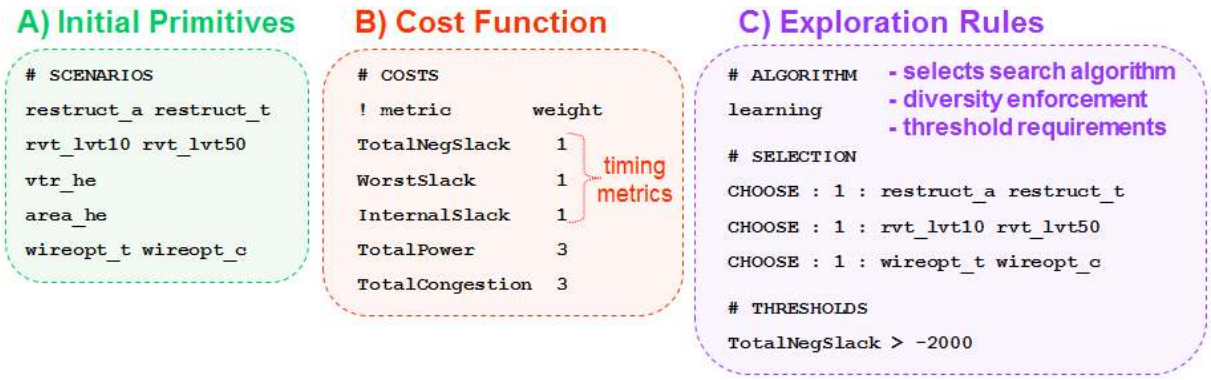


Figure 6.4: Components of the SynTunSys Rules file: A) the primitives to be explored during DSE; B) the cost function guiding the DSE; C) the search algorithm selection and configuration parameters.

signers. Although SynTunSys provides an official and documented primitive library, designers can extend it as needed to include new user-defined primitives. The official primitive library is also periodically updated as either designers or EDA developers provide new primitives suggestions, while primitives that have consistently performed poorly are removed.

To further reduce the design space to a reasonable size, independent of the size of the primitive library, we selected a subset of primitives from the primitive library for a specific SynTunSys run. The selected primitives are based on the optimization goals for a specific tuning run. As a rule of thumb, the number of selected primitives is comparable to the number of synthesis jobs that can be submitted in parallel to the compute cluster. The optimization goals and selected primitives for a specific tuning run are conveyed via the SynTunSys Rules file.

6.2.2 Rules File and Cost Function

The SynTunSys Rules file contains configuration settings for a specific tuning run. Figure 6.4 shows examples of the key sections of the "SynTunSys Rules" file: A) the primitives to be explored during the design space search; B) the cost function that will guide the DSE; C) the overall search algorithm as well as the parameters to configure the search algorithm. These parameters can adjust the effort level of the search as well as the exit criteria, and are often modified depending on the available compute resources and/or the size of the macro.

Cost Function. The SysTunSys cost function is a key setting that conveys the optimization goals

for the tuning run. It converts multiple design metrics into a single cost number, which allows cost-ranking scenarios. The user composes the cost function from one or more metrics based on data available from the results collection process. Examples of available metrics include: multiple timing metrics, power consumption, congestion metrics, utilized area, electrical violations, runtime, etc. The selected metrics are assigned weights to signify their relative importance. The cost function example in Figure 6.4 is a balanced cost function in terms of timing, power, and congestion. Each of the three metric categories has a total weight equal to three. The weight values in the timing category are uniformly divided across the three specific timing metrics. The overall cost function is then a *normalize weighted sum* of the m selected metrics, expressed by

$$Cost = \sum_{i=1}^{m} W_i \cdot Norm(M_i), \tag{6.1}$$

where W_i is the weight of $Norm(M_i)$, the normalized value of metric M_i across all the scenario results in a SynTunSys run.

In practice, multiple instances of the Rules files are created a priori for a chip design project that vary by cost function and search effort. Designers select an appropriate Rules file and optionally modify it for their specific needs. Designers also have the option to add their own user primitives to the search process, making the entire system expandable and customizable.

6.2.3 Tuning/Archiving Loops and Macro Construction Flow

The main tuning loop of SynTunSys begins when the monitor senses the completion of the current synthesis jobs, i.e., Step (3) in Figure 6.2. This process includes a thresholding step that eliminates scenarios with erroneous or missing results data and also scenarios that fail to meet minimum design criteria, as specified by the designer. The resulting scenarios are all valid for cost analysis, as described in the previous subsection. After cost ranking, an optional diversity enforcement routine may be applied. This process ensures that there is diversity among the parameters driving the DSE, e.g., primitives having higher cost than competitors across a dimension of the primitive space may be removed. The decision engine is the final step of the main tuning loop. This step is responsible for determining whether the process continues to a new iteration as well as the scenarios for the new iteration. Section 6.3 describes multiple decision engines developed during the system's evolution.

Running in parallel to the main tuning loop is a background loop that archives the results of all runs in a database, i.e., Step (5b) in Figure 6.2. The archived data provides a history of all previous runs that SynTunSys uses for multiple purposes. Archived data consists of high-level summaries of the input Rules files, output synthesis results for the entire tuning run, and more detailed log files from the synthesis runs. For this work we used the high-level summaries, but future work may make use of all log files. One use of the archived data is to generate DSE scenarios from historical primitive performance, represented by Arc (6b) in Figure 6.2. More details on this process are given in Section 6.3.3. The SynTunSys Rules files are periodically updated based on analysis of historical primitive performance. The historical performance is also useful feedback to the synthesis tool development team.

The overall macro construction flow covers logic synthesis to post-route optimization, as Figure 6.5 illustrates. For our case, SynTunSys operates on the logical and physical synthesis steps, as we see the highest ROI for DSE early in the post-RTL design flow; however, under other circumstances SynTunSys could include later steps as well. After SynTunSys, we employ a tailored design flow that selects a subset of the most attractive scenarios based on SynTunSys results. These scenarios continue through the routing phase of the construction flow. After routing, a smaller subset of scenarios is run through post-routing tuning algorithms and finally the top scenario is promoted to production data.

6.2.4 Addressing SynTunSys Overhead

Since SynTunSys runs multiple parallel synthesis jobs and multiple iterations, it inherently has a runtime and disk space overhead when compared to a single synthesis job. Knowing this up front, we aimed at minimizing the overall latency of a tuning run (number of iterations), the total number of synthesis jobs, and the disk space requirements for synthesis output data. These overheads, but also the quality of results, naturally scale with the size of the exploration design space, i.e., number of initial primitives and iterations.

The following section describes the SynTunSys decision algorithms for efficient DSE. Overall, these algorithms reduce the number of SynTunSys iterations to about 3–5 iterations, leading to a little over a 3–5X runtime increase versus a single synthesis run. Although this overhead may seem costly, within the scope of a large multi-year design project it is quite tolerable and provides a

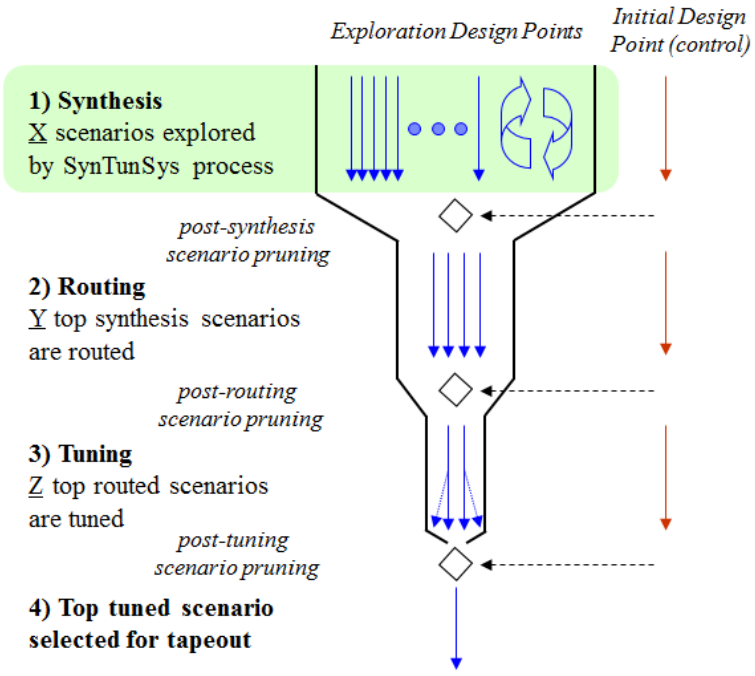


Figure 6.5: Overview of the DSE-based macro construction flow.

high ROI for the following reasons. First, running SynTunSys is not necessary every time a macro is synthesized; SynTunSys needs only to be run at certain points in the design cycle to locate customized parameters for a specific macro; during subsequent synthesis runs, the SynTunSys scenarios can be reused such that subsequent synthesis runs have no runtime overhead. Second, SynTunSys is an autonomous system that does not require designer effort once initiated. Thus, while the CPU runtime may be higher, the effort required from the designer is effectively the same as a single synthesis run. Thus, the ROI in terms of designer effort is very high (see the results presented in Section 6.4).

In terms of computing requirements, the algorithms described in the following section often lead to about 100–200 synthesis jobs across 3–5 iterations. However, the overall SynTunSys disk space footprint requirement is far less. SynTunSys minimizes the disk space usage by keeping only necessary data from promising scenarios (lowest cost scenarios) and dynamically deleting all other output data. This dynamic disk space recovery process often leads to a 75% reduction in disk requirements, i.e., resulting in a 25–50X disk space overhead for the 100–200 synthesis jobs.

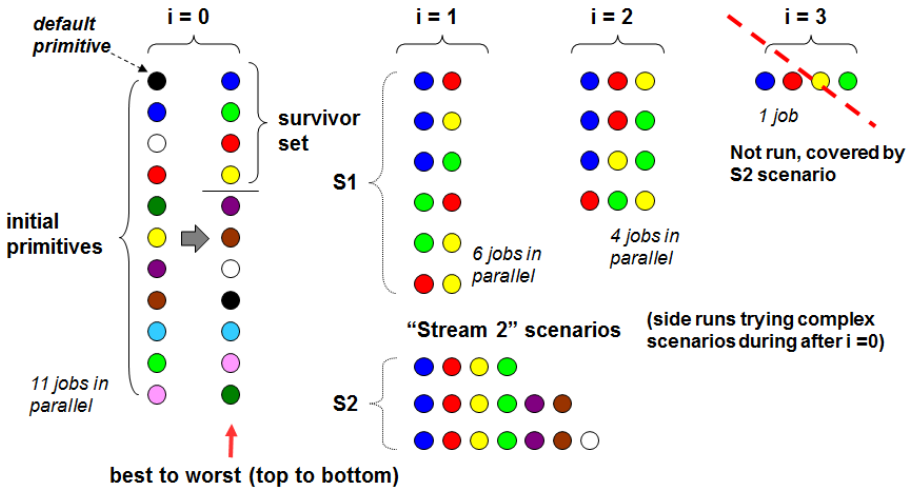


Figure 6.6: The Base decision algorithm, where each colored bubble represents a primitive, a horizontal sequence of adjacent bubbles represents a scenario, and "i" denotes the iteration number.

6.3 Decision Engine Algorithms

The SynTunSys decision engine and tuning algorithms are a key component of the system that determines the number of tuning iterations and the scenarios to be run during each iteration, as illustrated by Step (5) in Figure 6.2. The decision engine is also a component that can be upgraded independently. The goal of the design engine is to solve the following problem.

Problem 4. *Given a set of initial primitives, a cost function (the weights of Equation (6.1)), and a constant N, return the N lowest-cost scenarios.*

6.3.1 The Base Decision Algorithm

The initial SysTunSys decision algorithm, which we call the *Base* algorithm, was designed to emulate the manual DSE process that the designers would employ in absence of an automated system. It is a pseudo-genetic algorithm involving a survival of the fittest comparison (sensitivity test), followed by a dense search using the top primitives, as illustrated in Figure 6.6.

This algorithm begins with SynTunSys reading the Rules file and launching an initial iteration (i=0) consisting of one scenario for each primitive in the rules file. Each synthesis job in i=0 has only the single primitive enabled (1-hot), thus i=0 tests the sensitivity of each primitive as a scenario. When the SynTunSys monitor senses the completion of the current iteration and the cost

ranking is complete, the Base algorithm selects the top N (lowest cost N) scenarios as a *survivor set* and proceeds to iteration 1 (i=1).

Iteration 1 generates a stream S1 of more complex scenarios, consisting of combinations of primitives from the survivor set. The most common configuration is for each iteration i to generate all possible combinations of i+1 primitives.

Although this algorithm fully searches the design space of the survivor set, one limitation is that the size of the survivor set is often restricted due to compute resource requirements, i.e., the possible combinations of the survivor set determine the total number of scenarios to be run. To provide some relief to the issue of a restricted survivor set, an S2 stream of scenarios can also be added to the i>0 iterations. These scenarios are an attempt to search beyond the survivor set. The S2 scenarios are rule-based guesses, including: (1) combine the M lowest cost primitives ($M > N$, e.g. the middle S2 scenario in Figure 6.6), or (2) combine all the primitives that have lower cost than the default primitive or another reference primitive, (e.g. the bottom S2 scenario). In addition, the S2 scenarios also include the combination of all survivors (e.g. the top S2 scenario, which is essentially the S1 scenario from the last iteration), thus speeding up the overall tuning process (see "i=3" in Figure 6.6). Although the S2 scenarios were a late addition to Base algorithm to cover a known deficiency, they provided a bridge to allow search capabilities outside the survivor set until SDSE enhancements could be implemented.

6.3.2 The SDSE Learning Algorithm

Within the framework of the Base algorithm (i.e., a sensitivity test followed by iterative combination of scenarios), we present an SDSE decision algorithm, which we call the *Learning* algorithm. The Learning algorithm addresses two deficiencies of the Base algorithm: (1) a non-constant scenario count per iteration, which underutilizes parallel-computing resources for running multiple synthesis jobs, and (2) a search space restricted to the survival set, which was previously relieved by static search rules (the S2 scenarios), as described in Section 6.3.1. In contrast, the Learning algorithm always selects a constant number k of scenarios in each iteration as parallel synthesis jobs (i.e. max utilization), and dynamically adapts to the k scenarios that are more likely to return lower costs (i.e. adaptive exploration).

Figure 6.7 illustrates the main idea of the Learning algorithm. Following the sensitivity test

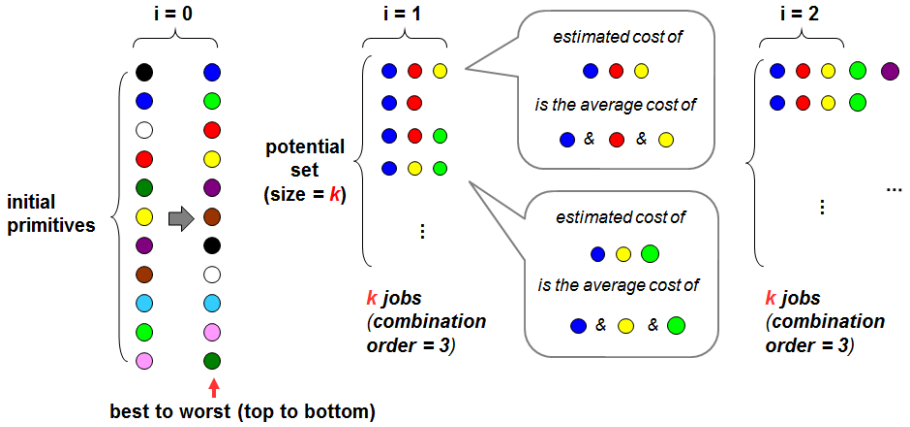


Figure 6.7: Illustration of the Learning decision algorithm.

(i=0) on the initial primitives (each as a scenario), the Learning algorithm estimates the cost of an unknown composite scenario by taking the average cost of its contributing scenarios as a cost predictor. For example, to estimate the cost of a scenario that comprises three primitives (shown in blue, red, and yellow), the Learning algorithm calculates the average cost of the three contributing scenarios that comprise the blue, red, and yellow primitives, respectively. The estimation is effective when the contributing scenarios are complementary. The max number of contributing scenarios to be evaluated is limited to a user-defined combination order (3 is used in Figure 6.7), so that the total number of scenario combinations would not explode to make the cost estimation infeasible. After the cost estimation, the Learning algorithm selects the top-k composite scenarios with the lowest estimated costs (i.e. a *potential set*) and then submits k parallel synthesis jobs with the selected scenarios. The estimation-selection-submission process repeats for every iteration until an exit criterion is met (e.g., the maximum value of i is reached).

The Learning algorithm leverages the iterative process to continuously refine its cost-estimation accuracy on non-complementary combinations. Specifically, at any iteration i, whenever a composite scenario, say "A+B", was predicted good (i.e. low cost) and selected for synthesis, but the synthesis result turned out to be bad (high cost), then the algorithm "learns" the actual effectiveness of combining scenarios A and B, and therefore at future iterations (>i) will demote any composite scenario that involves "A+B" (because as a unique contributing scenario, the cost of "A+B" is high). In summary, the Learning algorithm uses cost estimation to avoid non-promising scenarios and refines its estimation after learning the actual synthesis results.

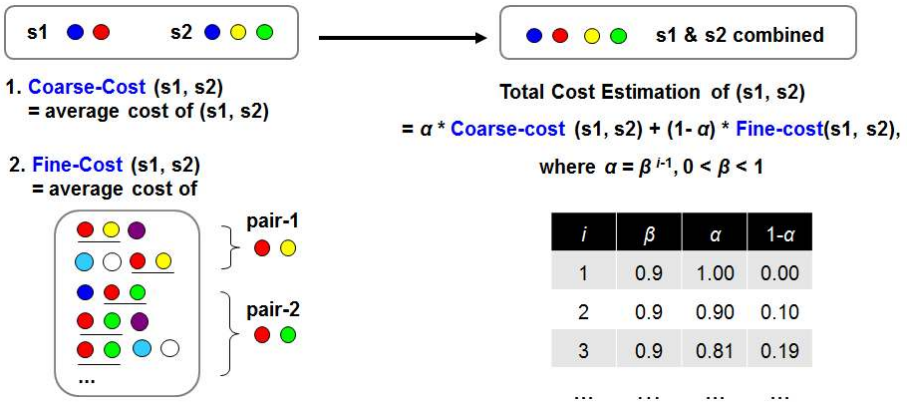


Figure 6.8: Cost estimation process for the Learning algorithm.

Moreover, in order to better estimate the cost based on non-trivial contributing scenarios (i.e., scenarios comprising more than one primitive), the Learning algorithm also includes a fine-grained cost estimation. For instance, as shown in Figure 6.8, given two scenarios, s1 = (blue + red) and s2 = (blue + yellow + green), the Learning algorithm regards the average cost of s1 and s2 as the *coarse-cost*, and additionally considers a *fine-cost* which is defined as follows. The fine-cost is the average cost of the *reference scenarios* that have been run in previous iterations and include a pair of primitives, such that one primitive comes from s1 and the other from s2 (see pair-1 and pair-2 in Figure 6.8). The scenarios listed to the left of pair-1 and pair-2 are the reference scenarios that include these pairs. Therefore, the fine-cost of s1 and s2 is the average cost of such listed scenarios.

Overall, the cost-estimation function of the Learning algorithm is a weighted sum of the coarse- and fine-cost with a weighting factor α for the coarse-cost and a factor $(1 - \alpha)$ for the fine-cost, where $0 \leq \alpha \leq 1$. For determining α we used the formula $\alpha = \beta^{i-1}$ where $0 < \beta < 1$ and i is the number of the current iteration. When i increases (and α decreases at a speed controlled by β), the Learning algorithm puts less weight on the coarse-cost, but more weight on the fine-cost. The rational is twofold: (1) as i increases, the reference scenarios comprise more primitives and (2) more reference scenarios become available. Figure 6.8 shows the change in α with increasing i, given $\beta = 0.9$.

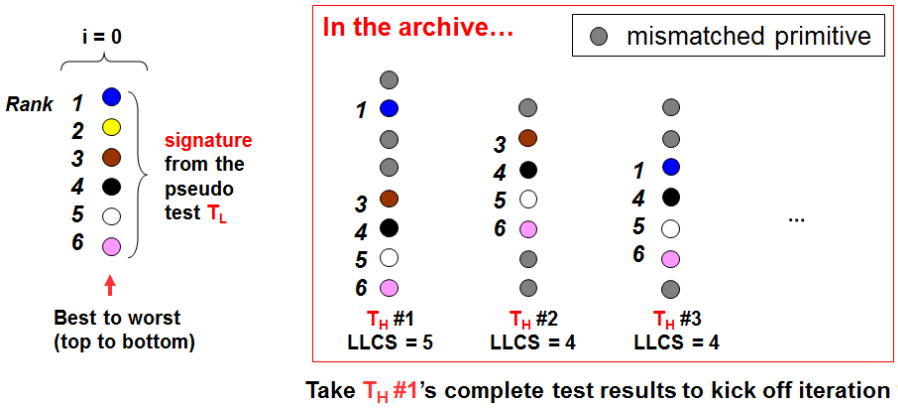


Figure 6.9: Illustration of the Jump-Start sensitivity test.

6.3.3 The SDSE Jump-Start Sensitivity Test

Another SDSE enhancement to the SynTunSys decision engine targets the reduction of the overhead of the sensitivity test, i.e., iteration i=0. The enhancement employs the archived data from prior SynTunSys runs on different macros. It can be applied to both Base and Learning algorithms. We use the Base algorithm to present the idea of the enhancement.

The default sensitivity test of the Base algorithm provides three levels of tuning effort: low, medium, and high, featuring 19, 38, and 54 initial primitives, respectively. In order to increase the efficiency of the sensitivity test (i.e., to approach the higher-effort exploration result based on a lower-effort sensitivity test), we present the *Jump-Start* test. The Jump-Start test is based on the archived high-effort tests from other macros. As illustrated in Figure 6.9), the idea is to run a *low-effort* sensitivity test for the current macro as a pseudo test T_L, and to take T_L's ranking of the initial primitives as a *signature*. Then, the Jump-Start test retrieves, from the archive, a *high-effort* test T_H which has the most similar primitive ranking to T_L's signature. Finally, the Jump-Start test adopts T_H's primitive ranking as the outcome of the sensitivity test and then uses the retrieved primitives to kick off the search iteration (i=1). The final exploration result should improve if the number of primitives in T_H is greater than that in T_L, and the primitive rankings (signatures) of T_H and T_L are very similar.

We measure the similarity between two signatures using the Length of Longest Common Subsequence (LLCS) metric. Each signature is represented by a sequence in the order according to the primitive ranking. For example, given primitives in $[a \ldots z]$, the longest common subsequence of two

signatures $< a, b, c, d >$ and $< a, f, d >$ is $< a, d >$ with LLCS equal to two. Figure 6.9 illustrates a low-effort sensitivity test T_L with six primitives, while in the archive the high-effort sensitivity tests with more than six primitives are identified and sorted according to their LLCS with respect to T_L's signature. Among these high-effort tests, T_H #1 has the maximum LLCS (five). Thus, the complete sensitivity-test result of T_H #1 is taken to kick off the first search iteration (i=1).

As a tie-breaker for selecting among T_H's with equal LLCS, we favor the T_H that includes more high-ranking primitives, e.g., T_H #3 is considered more similar to T_L than T_H #2 in Figure 6.9.

6.4 Experimental Results

We describe the use of SynTunSys during the design of an IBM high-performance server processor [182] in a 22nm technology. The processor underwent two chip releases (tapeouts) over a multi-year design cycle. The chip consists of a few hundred macros that average around 30K gates in size, with larger macros in the 300K gate range. SynTunSys was applied during both tapeouts, but with slightly different usage models. Also, at the time of this chip design, only the Base tuning algorithm was available, as most of the SynTunSys development effort involved scaling to widespread use across a large design team. In Sections 6.4.2 and 6.4.3, we validate the SDSE enhancements to SynTunSys on macros from the production processor.

6.4.1 Application to an IBM 22nm Server Processor

During the first chip release, a dedicated SynTunSys team performed tuning for hundreds of macros across the chip. In parallel, a number of designers further tuned the macros they owned, as needed. Based on the efforts of the dedicated tuning team we were able to track SynTunSys results on approximately 200 macros from the processor. The "pass 1" rows of Table 6.2 show the average improvements achieved by SynTunSys over the best solution previously achieved by the macro owners for the first chip release. Note that these results are based on the routed macro timing and power analysis; in most cases the best known prior solutions included manual parameter tuning by the macro owner. The first chip release of the processor targeted a very aggressive cycle time and SynTunSys was typically applied for timing, power, and congestion improvement. The first pass of SynTunSys resulted in a 36% improvement in total negative slack, a 60% improvement in

Table 6.2: Average SynTunSys improvement over best known prior solution based on post-route timing and power analysis

SynTunSys Pass (Chip Release)	Latch-to-Latch Slack	Total Negative Slack	Total Power
Improvement	(%)	(%)	(%)
Pass 1	60	36	7
Pass 2	24	2	3
Sum of 200 macros	(ps)	(ps)	(arb. units)
Pre-pass 1	-1929	-2150385	17770
Post-pass 1	-765	-1370731	16508
Sum of 25 macros	(ps)	(ps)	(arb. units)
Pre-pass 2	-260	-185087	3472
Post-pass 2	-198	-180896	3379

worst latch-to-latch slack (macro internal slack), and a 7% power reduction. The actual values of the metrics, summed across all the macros, underscores that the changes in the absolute numbers were significant. In terms of timing, the summed values are in picoseconds.

Following a successful use of SynTunSys during the first chip release, the design requirements for the second chip release included a second SynTunSys tuning run by the macro owners to further improve timing and power. These second-pass tuning runs build off the prior tuning run results to further search the design space. In some cases the macro logic was also quite different in the second chip release, leading to a different design space. In these cases we had a less controlled study, but wider usage by macro owners. Based on data from 25 macros (Table 6.2, "pass 2") we still see considerable improvements after the second SynTunSys pass on macros during the second chip release. While latch-to-latch slack improved by 25% and power was reduced by 3% (about half the rate as the first-pass tuning run), we believe the total negative slack reduction was gated by overly aggressive boundary assertion timing constraints.

While we saw a highly successful deployment of SynTunSys to a production processor design, we noted many potential enhancements to the system that could improve prediction performance and efficiency for future designs. In the following subsections we describe the SDSE enhancements

Table 6.3: Macros for Tuning Learning-Based Algorithms

Macro	Datapath	Modeling Accuracy	# Gates
1	Yes	Gate-level	16K
2	Yes	Transistor-level	3K
3	No	Gate-level	13K
4	No	Transistor-level	18K

that we achieved since the second processor tapeout.

6.4.2 Learning vs. Base Algorithm Results

The new Learning algorithm provides a number of configuration settings that can be fine-tuned to improve its performance. In this subsection we compare various configurations against the Base algorithm on a set of representative medium-sized macros to find the top settings for the Learning algorithm. We then compare the winning Learning algorithm with the Base algorithm on larger macros.

Table 6.3 lists the set of macros used for the comparisons. These macros feature a mix set of datapath vs. non-datapath types and gate-level vs. transistor-level accuracy for cost modeling. They also come from different units of a processor core. Overall, these macros pose completely different challenges to the synthesis tool.

Our experiments consider a relatively balanced cost function in terms of timing, power, and congestion, by putting a weight of 4 (36%)to total power consumption, 4 (36%)to multiple timing metrics, and 3 (27%)to congestion (for routability). The multiple timing metrics include total negative slack, worst overall slack, and worst latch-to-latch slack.

Table 6.4 shows the configurations of the algorithms for comparison. We let each Learning-based algorithm run for three iterations in addition to the sensitivity test. Thus, with the same total iteration count (i.e., 4, which is the *latency* of the exploration processes), the total scenario count (excluding the sensitivity test) of each Learning-based algorithm can be $3 \times 20 = 60$, whereas the total scenario count of the Base algorithm is inherently restricted to $C_2^5 + C_3^5 + C_4^5 + C_5^5 = 26$. Note that in this experiment each algorithm explores 19 initial primitives.

Table 6.4: Algorithm Configuration

Algorithm	Survival/Potential Set Size	Combination Order	Cost Estimation
Base	5	2	None
Learning2	20	2	Coarse
Learning3	20	3	Coarse
Learning3+	20	3	Coarse + Fine

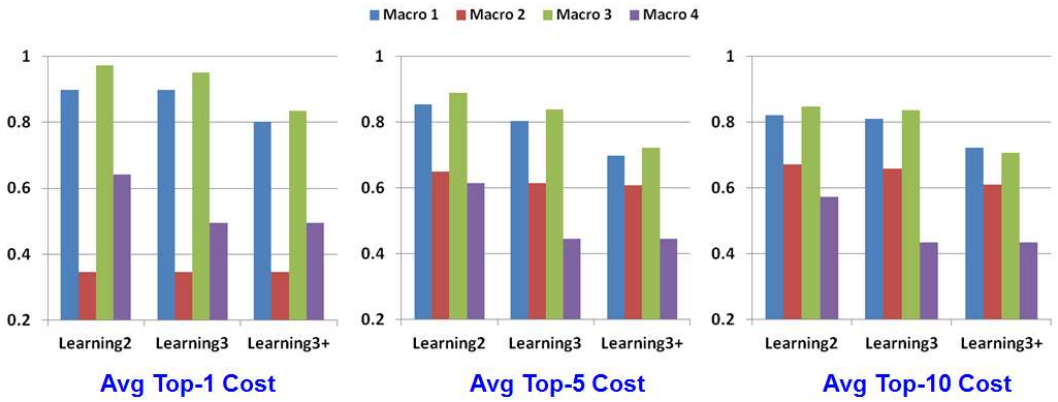


Figure 6.10: Results comparing the Base vs. Learning algorithms.

We compare these algorithms in terms of the average cost of their final top-N results, with $N \in \{1, 5, 10\}$. When comparing two algorithms, we put all explored results from both algorithms together and calculate the cost of each result according to the balanced cost function. Figure 6.10 shows the cost comparison (the lower the better), where Learning-based algorithms are each compared with Base (i.e., Base as 1.0). Overall, the winner is Learning3+, which consistently outperforms Learning3, which in turn outperforms Learning2. The Learning3+ victory over the other Learning algorithm variants can be conceptually justified with two hypotheses: 1) the higher combination order allows exploring scenarios that combine more primitives during earlier iterations and 2) the addition of the Fine-Cost component in the cost estimation does indeed improve prediction accuracy. Note that all the Learning-based algorithms outperform the Base algorithm regardless of the cost-prediction functions or combination setting. This result validates the idea of adaptive exploration, which differentiates the Learning-based algorithms from Base.

Furthermore, we compare the Learning3+ algorithm directly against Base on the largest test

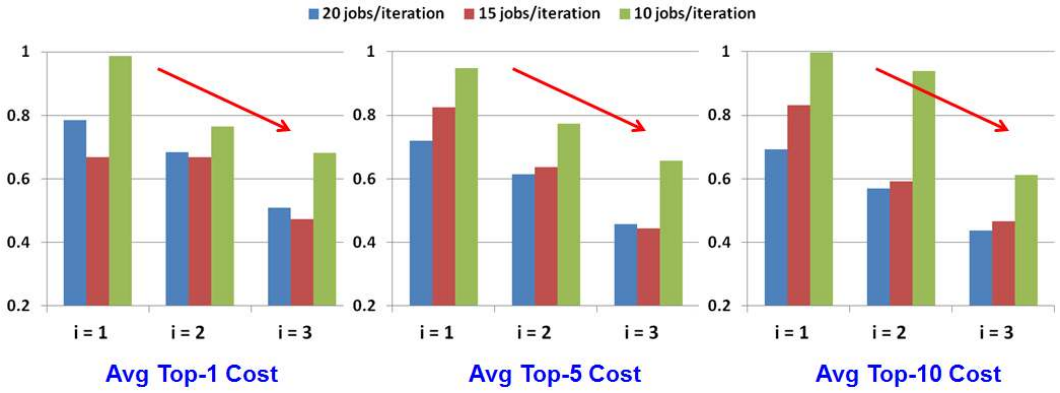


Figure 6.11: Iteration-by-iteration costs: Learning3+ vs. Base.

macro (Macro 4), and let Learning3+'s potential-set size vary in $\{20, 15, 10\}$. This means that the total scenario count (excluding the sensitivity test) for Learning3+ varies in $\{60, 45, 30\}$. The iteration-by-iteration cost comparison is shown in Figure 6.11, where the cost of Base is normalized as 1.0. We can see that Learning3+ outperforms Base even with an equal number of scenarios. Moreover, Learning3+ not only wins, but wins by a larger margin as the number i of iterations increases (see the trend indicated by the red arrows).

With the same balanced cost function, now we compare Learning3+ against Base on a test suite of 12 macros spanning a variety of sizes. These 12 macros range in size from 1K to 110K gates, with an average size of 31K gates, which is the same average macro size as the server processor. The goal of this study is to validate and quantify the Learning3+ improvements over Base. In order to push the exploration results further, we considered 49 initial primitives. In addition, Learning3+ ran for 5 iterations besides the sensitivity test, and Base considered the 6 best primitives for its survivor set. Thus, the total scenario count (excluding the sensitivity test) of the Learning3+ can be $20 \times 5 = 100$, whereas the total scenario count of Base is inherently restricted to $C_2^6 + C_3^6 + C_4^6 + C_5^6 + C_6^6 = 57$. Note, however, that the maximum scenario count per iteration of these two algorithms is the same (i.e., 20 or C_3^6), which is the max *throughput* of the exploration processes.

The exploration results are summarized in Table 6.5. For this study we did not route the macros and, therefore, we report only post-synthesis statistics. In lieu of routing, we include a routability metric called *route-score*, in which a lower value denotes less congestion and a more routable macro. Over all the metrics, the results reported in Table 6.5 are in the same average

Table 6.5: Comparison of Learning3+ and Base algorithms across a 12 macros test suite that is representative of a larger processor

	Timing			Power	Routeability
	Worst Slack	Latch to Latch Slack	Total Negative Slack	Total Power	Route Score
Improvement	(%)	(%)	(%)	(%)	(%)
Base	24	43	42	11	46
Learning3+	35	70	62	10	43
12 macro sum	(ps)	(ps)	(ps)	(a. u.)	(a. u.)
Default	-256	-105	-77505	1125	129
Base	-195	-60	-45104	1003	69
Learning3+	-167	-32	-29138	1007	74

improvement range for the Base algorithm as the results reported for the industrial processor design in Table 6.2. However, *the Learning3+ algorithm achieves an additional 20% total negative slack improvement over the Base results as well as significant improvements across all timing metrics.* On the other hand, the power and congestion improvements are essentially the same for the Base and Learning3+ algorithms. Note that if further power or congestion improvements were desired, increasing the weight of these metrics in the cost function is one avenue.

In terms of how results may differ across macros, Figure 6.12 provides a macro-by-macro breakdown of the percentage of change for each metric category in the cost function. In an attempt to simplify the plot, the improvements of the three timing components are averaged for a single timing improvement factor. The plot shows only one macro that suffers a degradation (macro H) with the Learning3+ algorithm, while three macros do so with the Base algorithm. Note that degradation in the route-score can often be tolerated, if the initial macro has low congestion.

To provide a visual representation of the explored design space during a SynTunSys run, we plot the design space of the two largest macros in Figure 6.13. These plots show total power (top row) and a route-score (bottom row) vs. total negative slack (all values normalized). The less the power/route-score/negative-slack, the better the quality of result (i.e., the optimization goal is the lower-right corner of the plots). We can see that Learning3+ effectively optimizes all the three metrics for macro K. The results also illustrate that Learning3+ indeed adapts to the more promising design space: the "i=5" dots are more focused on the lower-right region of the spaces

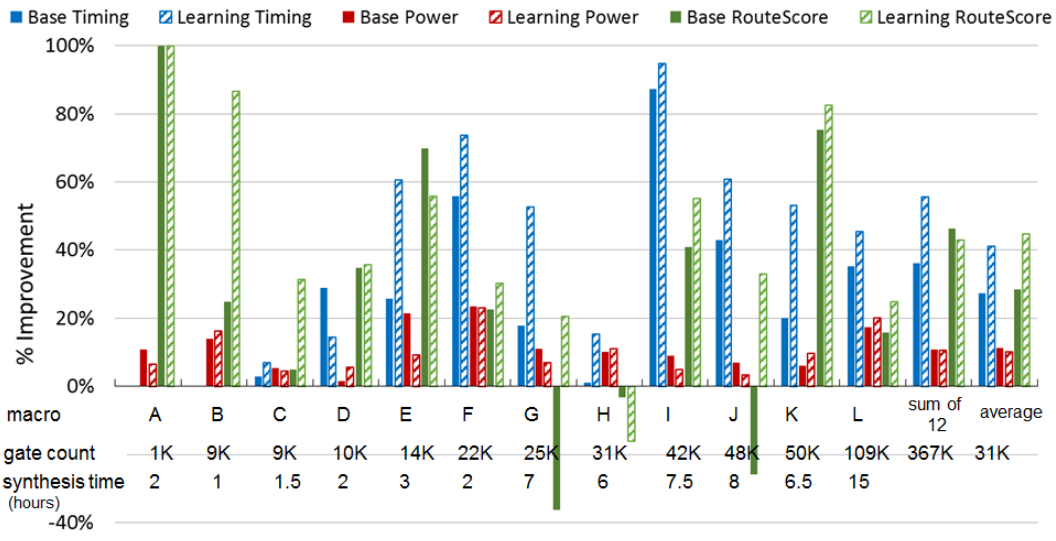


Figure 6.12: Macro-by-macro breakdown of Learning3+ vs. Base.

than the "i=3" dots. On macro L, Learning3+ adapts to optimizing the timing and power, whereas the Base algorithm finds solutions with better route score. Nevertheless, if we compare the *best* macro L based on the balanced cost function (Figure 6.12), Learning3+ actually outperforms Base in each metric, including route score. Overall, both algorithms achieve better results than the default scenario.

6.4.3 Jump-Start Sensitivity Test Exploration

Next, to explore the effectiveness of the Jump-Start sensitivity test, we use the same set of smaller macros listed in Table 6.3, and compare it against the standard sensitivity test. For this study we use the Base algorithm for the comparison; however, since the Base and Learning algorithm rely on the same initial sensitivity test, the results of this study are representative of both algorithms. The Jump-Start test is based on an archive of hundreds of high-effort sensitivity tests. By default, the low-effort and high-effort Base algorithms are given 19 and 54 initial primitives, and search in survival sets of size five and six (corresponding to total $19 + 26 = 45$ and $54 + 57 = 111$ scenarios to be explored), respectively.

Figure 6.14 shows the cost comparison between the default low-effort Base algorithm (whose cost is normalized as 1.0) and the low-effort Base algorithm with Jump-Start sensitivity test. Though with the same actual scenario count (45), the Base algorithm with Jump-Start can outperform

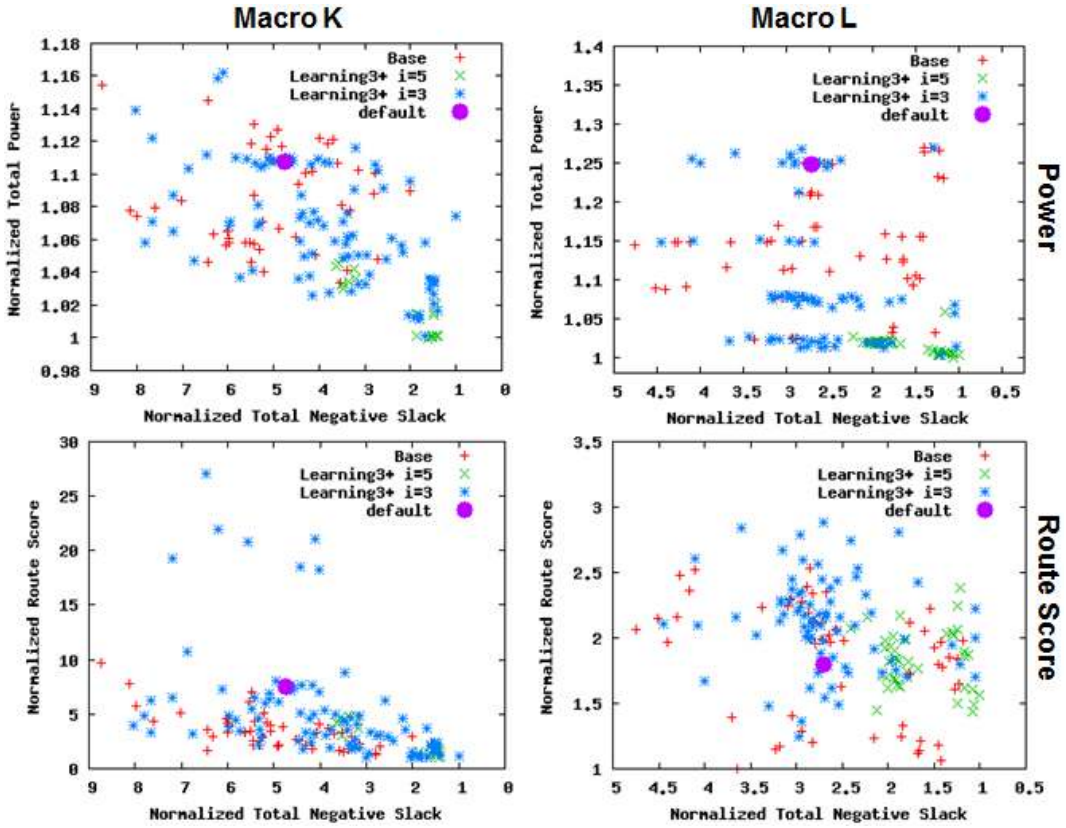


Figure 6.13: Power (top row) and Route-Score (bottom row) vs. Total Negative Slack for large macros K (left column) and L (right column).

the default Base algorithm, thanks to the high-effort primitives retrieved from the archive. The improvement does not require an actual high-effort tuning run, which needs 111 scenarios in total. The reduced effort (i.e. reduced turn-around time) is particularly useful for early design evaluation, e.g., when the logic design is still under significant change.

6.5 Remarks

Related Work. SynTunSys addresses a black-box parameter tuning problem for logic and physical synthesis. Black-box search is a common problem seen across a number of fields, e.g., Arcuri and Fraser look at the problem in terms of software engineering [10]. This problem has also been approached using a number of techniques, e.g., genetic programming [16] and Bayesian optimization [181]. Similarly, black-box parameter tuning problems have been explored for high-level

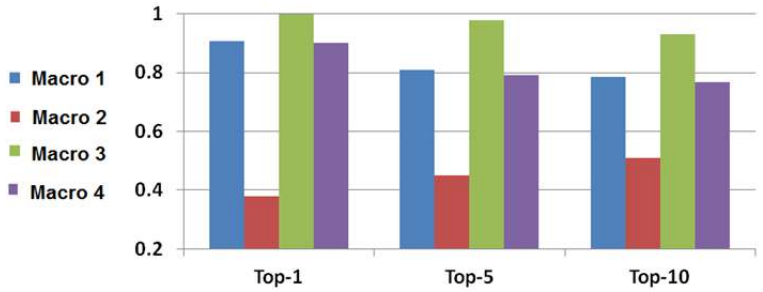


Figure 6.14: Average cost comparison between the Just-Start test vs. the Base test (Base as 1.0).

synthesis (HLS), e.g. [197]. However, to our knowledge, SynTunSys is the first self-evolving and autonomous parameter tuning system for logic and physical synthesis levels. Furthermore, whereas many problem formulations involve a sequential process for parameter tuning, SynTunSys performs both a parallel and iterative DSE to reach attractive solutions with relatively low latency.

Concluding Remarks. By taking over the process of tuning the input parameters and by learning automatically from the information of previous synthesis runs, SynTunSys realizes a new level of abstraction between designers and tool developers. The application of SynTunSys to an IBM 22nm high-performance server processor yielded on average a 36% improvement in total negative slack and a 7% power reduction. SynTunSys has been further enhanced with SDSE algorithms in preparation for future processor projects. This work opens new avenues of research such as automatic feature extraction of parameters and primitives for design-space reduction and enhanced mining of archived data to improve scenario combinations.

Part III

Extending Supervised Design-Space Exploration

Chapter 7

Code Refactoring with Frequent Patterns

The SDSE frameworks presented in Parts I and II fully rely on synthesis tools for the DSE process. This requirement incurs the following limitations, especially for the HLS-based frameworks.

Limitation 1. HLS takes an IP specification as a fix input; thereby its HLS-explorable design space is intrinsically confined by the synthesis knobs that are applicable to the specification. However, there are few existing works on enhancing the IP reusability at the functional specification level, i.e., by restructuring the design specification to provide better DSE starting points.

Limitation 2. Most commercial HLS tools suffer from limited Quality of Result (QoR). The QoR is limited to what is achievable with the scheduling and binding on *primitive operations* (such as additions, multiplications, etc), as opposed to *composite operations* (aka *patterns*; a pattern is composed of multiple primitive operations). However, pattern-based optimization has long been proven successful at different abstraction levels, e.g., technology mapping [99] and logic transformation [107], thanks to the common structural regularity in digital design. In this regard, the HLS solution space covered by patterns turns out to be hardly reachable by most commercial tools.

Limitation 3. We have shown that commercial HLS tools suffer from long runtime in Chapters 3 and 5. Although SDSE frameworks can reduce the number of HLS jobs for DSE, the HLS tool runtime per se does not changed. Therefore, in the worst case the aggregated HLS runtime may still explode to make the DSE process infeasible.

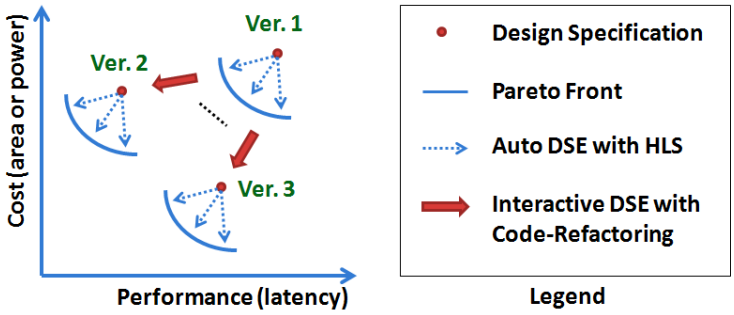


Figure 7.1: Enhanced DSE with HLS using automatic and interactive processes.

7.1 Problem Description

To address the aforementioned three limitations, we define the problem of Code Refactoring for Enhanced Design Exploration with High-Level Synthesis as follows.

Problem 5. *Given a synthesizable C-based specification, refactor it such that its HLS-explorable design space is enhanced for greater IP reusability, better HLS QoR, and less HLS runtime.*

Fig. 7.1 illustrates the motivations behind Problem 5. In the context of HLS, an *automatic-DSE* process (e.g. [98; 112; 160]) finds the best knob configurations for running HLS on a given design specification, in order to discover its Pareto front. Fig. 7.1 shows three such automatic-DSE processes in a cost-*vs.*-performance design space. Each of these processes starts with a unique version of a design specification. These versions (Ver. 1–3) differ in their code structure, but preserve the same functional behavior, thereby exposing different opportunities for knob applications. On the other hand, *code refactoring* [70] has long been a common practice in software engineering for improving software quality, but is now becoming popular for hardware description [172; 194]. By applying code-refactoring to DSE, we aim at pushing the design frontier (i.e. the IP reusability and the HLS QoR), e.g., from Ver. 1 to Ver. 2 to improve mostly the performance or from Ver. 1 to Ver. 3 to reduce mostly the cost. In addition, Ver. 1 and 3 should each require less HLS runtime compared with Ver. 1.

The fundamental novelty of Problem 5 with respect to related research problems is that we focus on the optimization of the design specifications as opposed to the synthesis engines. After all, it is the optimized specifications that are reused across SoCs/design-teams/system-companies,

not the (so many) HLS tools. As a major contribution, we point out this critical missing piece for enhancing DSE with HLS and promoting soft-IP reuse. Other contributions are listed as follows.

- We propose an *interactive-DSE* methodology to enrich the design space that HLS tools can explore. Our methodology does so by *suggesting code-refactoring options* to designers, so that the starting point of the DSE process can improve, without changing design functionalities and HLS engines.

- As a concrete realization of our interactive methodology, we present a code-refactoring design flow based on *automatic pattern discovery*. In the flow, frequent patterns are to be regrouped into custom C function calls. By doing so, HLS engines can be unleashed to schedule and bind each of these function calls as a composite operation. Moreover, we present a pattern-discovery tool (whose prototype implementation is described in Appendix A.4) to identify the top-k patterns that appear frequently in a given design.

- By optimizing the design specification, our methodology makes the optimization result available in a soft-IP form, which can work seamlessly with most C-based commercial HLS tools. In fact, there were up to 17 commercial HLS products available in 2014 [1] and most of them do *not* offer in-synthesis pattern discovery. Therefore, an HLS-agnostic methodology like ours can have a general impact.

7.2 A Code-Refactoring Design Methodology

Fig. 7.2 shows a simple example to motivate the idea of code refactoring. The original specification is shown in Fig. 7.2(a) and we assume that each primitive operation ('$*$', '$+$', and '$\gg$') takes one clock cycle. For the original specification, a heuristic scheduling algorithm such as the List algorithm [52] may generate a schedule as the one shown in Fig. 7.2(b). In this schedule, although the multipliers in cycles C1 and C2 can be shared to save circuit area, this sharing would require a multiplexer to choose the right input ($x*2$ or $2*m$) for the adder that is shared in cycles C3 and C5. Instead, if the multiplier in cycle C1 were scheduled to cycle C2, we could remove the multiplexer because the multipliers for "$x * 2$" and "$2 * m$" would become the same one by safely overwriting the register that stores the multiplication results. Note that the total numbers of multipliers and

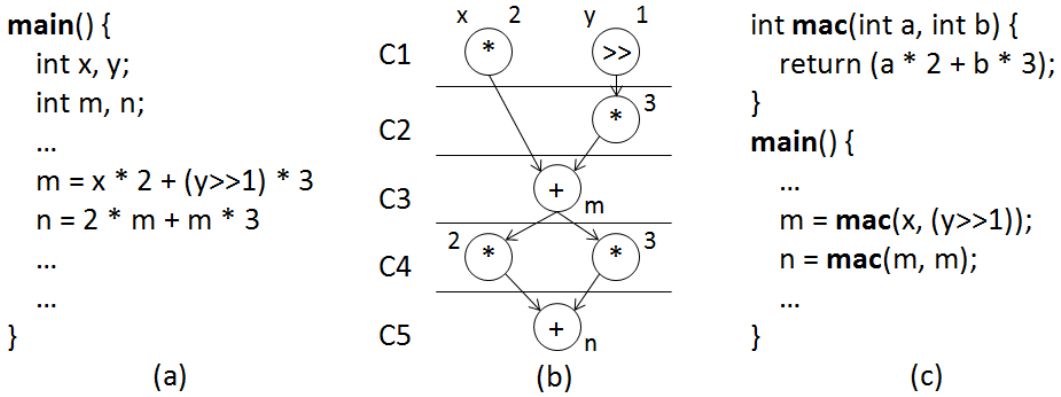


Figure 7.2: (a) Example original specification. (b) A sub-optimal schedule for the original specification. (c) Suppose that a frequent pattern consisting of two multipliers plus an adder is selected. Accordingly, we can refactor the specification with a customized function call `mac`, which is essentially a composite operation if `mac` is *not* inlined during HLS.

adders are not changed for these two different schedules. The later (better) schedule, however, requires the prior knowledge of a pattern that consists of the two multipliers and the adder. To achieve this better schedule, we can guide the HLS heuristic by customising a function call *mac* that is essentially the pattern composed of two multipliers and an adder, and then refactor the original specification by calling *mac* properly, as shown in Fig. 7.2(c). Thus, the HLS heuristic could avoid the local optima by first synthesizing *mac*, followed by the synthesis of the code that calls *mac* using a reduced number of multiplexers (**Advantage 1**).

The refactoring can also bring other advantages in addition to area saving. First, the refactoring may help HLS escape from local optima in terms of design performance (e.g. clock periods) as well (**Advantage 2**). The potential performance improvement is again caused by the heuristic nature of HLS algorithms. The change of the code structure corresponds to perturbing the input to the HLS, i.e. jumping to different starting points from a local search perspective [44]. Although the performance improvement is not guaranteed, the refactoring does not deteriorate the design at all, because the original specification can be completely recovered by inlining the custom functions. Second, the refactoring with patterns could also save HLS runtime (**Advantage 3**). The potential runtime saving comes from hierarchical synthesis that treats patterns as composite operations [27]. With hierarchical synthesis, the total number of operations to be scheduled can be reduced, i.e.

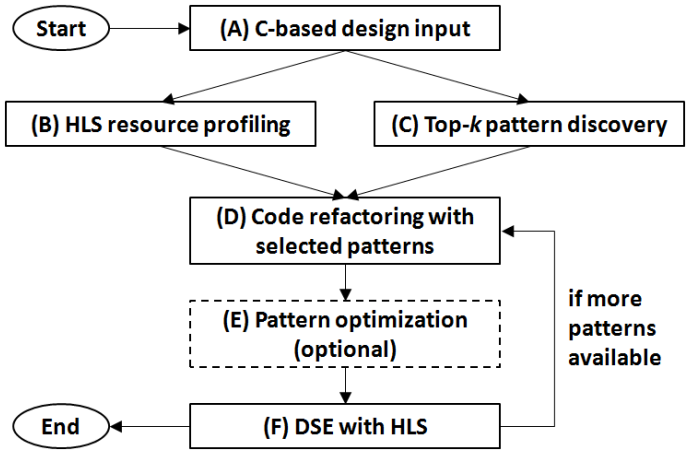


Figure 7.3: A code-refactoring design methodology for enhanced DSE with HLS.

the problem size for HLS scheduling can be shrank. For example, the six multiplication/addition operations shown in Fig.7.2(a) become two *mac* operations for the *main* function in Fig. 7.2(c). Note that the reduction applies to wherever *mac* is called, so the aggregated reduction can be significant. Finally, Advantages 1 and 2 combined could result in a better Pareto front as already illustrated in Figure 7.1. However, with the current HLS technology the Pareto-front optimization is not *portable* unless all the HLS tools feature in-synthesis pattern discovery (which is patent-protected [93]) or there is a dominating HLS tool with that feature (which is currently not true [1]). On the other hand, *code refactoring can provide an alternative solution for preserving and spreading the desired optimization*, since a properly-refactored design specification can be reused to expand the HLS-tool explorable design space (**Advantage 4**). To our knowledge, there is no prior work on improving the IP reusability with high-level specification languages.

To maximize the odds of the aforementioned advantages, we need to identify *frequent patterns* in the original specification and use them to refactor it properly. Based on this principle, we propose a code-refactoring methodology, as shown in Fig. 7.3, which consists of the following steps.

(A) The starting point is a C-based design specification of the IP core that can be synthesized by HLS tools. An HLS tool takes the specification and generates an intermediate representation that is used for resource allocation, scheduling, and binding. A common intermediate representation is a set of data flow graphs (DFGs) [52].

(B) A set of trial HLS runs are done for profiling the to-be-mapped resources in terms of their

implementation costs (area/power/etc.) under tight and loose timing constraints. The goal is to obtain the extreme operating conditions of each type of resources. The conditions are quite dependent on the implementation platforms (ASIC vs. FPGA), IP libraries, and technology nodes.

(C) The top-k frequent patterns (composite operations) are identified. The selection criterion is based on the number of DFGs that include a pattern. A pattern that appears in more DFGs has more chances to be shared across the DFGs, thereby reducing its total implementation costs. A list of k pattern suggestions with their appearances in the source code is presented to the designer.

(D) The designer selects patterns from the top-k list, considering their potential return of investment based on the analysis from Step B. If a less frequent pattern needs larger or more power-hungry resources, then the designer should re-rank the pattern to a higher position for potentially better cost reduction via resource sharing. We present other selection guidelines later in Section 7.2.1. At the end of this step, each of the selected patterns forms a customized function call. The original design specification is then refactored with these function calls as shown in the example of Fig. 7.2. The customized function calls will be treated as composite operations if the calls are *not* inlined during HLS, thereby enabling coarse-grained optimization and hierarchical synthesis.

(E) The selected patterns can be separately optimized via aggressive custom design or mapped to existing in-house or third-party IPs. This step is optional, depending on the available design efforts. Alternatively, the patterns can still be implemented via HLS.

(F) The final DSE process can be done via exhaustive search in a knob-configuration space on small/medium designs or via advanced techniques (e.g. [8; 11; 18; 77; 85; 98; 112; 145; 165; 160]) on medium/large designs. If there are more patterns available for alternative or further refactoring, the designer can go back to Step D; otherwise, stop here.

The methodology leverages the expertise of designers whose insights are critical for pattern selection as we discuss in Section 7.2.1. For the actual code refactoring, there exist automatic C-function-extraction tools such as Klockwork [2], but even manual refactoring with a pattern takes

only minutes for practical designs (see other manual refactoring examples in [172] for reference on the refactoring effort). For Step C, we developed an automatic pattern-discovery tool for assisting the analysis, which would otherwise become the productivity bottleneck in the methodology. Our pattern-discovery algorithm is presented in Section 7.3, following the presentation of the pattern-selection guidelines in the next subsection.

7.2.1 Pattern Selection

By applying our methodology to many experiments, we have learned the following practical guidelines for pattern selection.

(1) Through trial HLS runs, we observed that multipliers and adders contribute the most to the overall implementation area, whereas other resources are smaller than multiplexers. Since to share resources incurs additional multiplexers, sharing resources that are way too small leads to suboptimal micro-architectures. Hence, the selected patterns should include at least a multiplier, an adder, or a complex composition of other resources.

(2) It is not worth considering patterns with mostly constant inputs because modern HLS front-end can perform aggressive resource optimization via constant propagation and variable life-time analysis. For instance, consider a pattern `ROT`(x, n) that rotates a 32-bit integer x by n bits: $(x << n)|(x >> (32-n))$. If this pattern are only called as either `ROT`(x, 5) or `ROT`(x, 30), the parameter n is essentially constant and therefore can be optimized away. However, patterns that include constant operands internally are generally fine, as HLS can still optimize their implementation. For instance, the pattern `ROT`(x, n) is fine if both its parameters are unknown at the compile time.

(3) It is not worth regarding two patterns as identical if their function is the same but their input bit widths are very different (for example, differ by more than 8 bits). In this case, pattern sharing would force redundant area/power for unnecessary bits.

(4) It is worth prioritizing patterns that reside in major loop bodies. A major loop is usually the computation bottleneck in an application, and the manipulation of the loop (by breaking, unrolling, and pipelining) can generate a rich set of micro-architecture alternatives that offer

various degrees of resource sharing for the loop body [112; 165; 160]. In this case, a pattern sharing across the loop iterations can be especially effective in HLS QoR/runtime improvements.

(5) Non-trial refactoring opportunities can help. Fig. 7.4 shows a non-trial refactoring example from a double-precision floating-point application (DFSIN, one of the benchmarks used in our experiments). The example includes a function *func* in the original specification as shown in Fig. 7.4(a). Suppose that we are given a pattern that is represented by the *top2* function in Fig. 7.4(b) and that the *top2* pattern is already frequent *outside* the function *func*. At first glance, the "if" statement of function *func* in Fig. 7.4(a) seems unable to be refactored with *top2* because *(i)* the less-then and the less-than-or-equal-to comparators are not equivalent and *(ii)* the left-shifting results are taken in different orders by the comparators, which are not commutative operations. However, a refactoring with *top2* turns out to be possible if we apply the following transformation steps: *(i)* negate the predicate of the "if" statement in Fig. 7.4(a) as shown by the comment S1 below the function *func*, *(ii)* refactor the negated predicate with the *top2* function as show by S2 below *func*, and then *(iii)* remove the negation and restructure the if-else statements to produce the refactored *func* as in Fig. 7.4(b). Thus, the sharing opportunity of *top2* can be increased without incurring extra operations (such as negation) or dramatically changing the control flow. Our experimental results for this benchmark show significant QoR improvements by the refactoring with *top2*.

(6) Complex patterns are not necessarily better. The more complex is a pattern, the less frequent it appears in the whole design. Therefore, the sharing opportunities of a complex pattern can be less than those of a simpler pattern. Conversely, a simpler pattern can offer more effective optimization opportunities than a complex pattern. The final effectiveness should account for other guidelines such as 1 to 5. Our experimental results also confirm this observation.

We argue that there does not exist a universal cost function that can model all the above guidelines with perfect weighting factors. Especially in our problem setting, which is general and HLS-tool agnostic for maximal IP reusability, the designer's expertise would be extremely valuable for the pattern selection process. In this regard, our methodology lets the refactoring-recommendation tool narrow the design choices down to a human-manageable subset and then allows the design expert to make the final selection.

```
func(int a) {
  ...
  int count = 0;
  if (a < (1 << 32))
    count += 32;
  else
    a >>= 32;
  ...
}
// S1: if (!((1<<32) <= a))
// S2: if (!top2(1, a))
```

(a)

```
bool top2(int x, int y) {
  return ((x<<32) <= y);
}
func(int a) {
  ...
  if(top2(1, a))
    a >>= 32;
  else
    count += 32;
  ...
}
```

(b)

Figure 7.4: (a) Example original specification. Under `func` body, we list two transformation steps S1 and S2 that derive equivalent "if" statements to the original "if" in `func`. (b) A refactoring of `func` with `top2`.

7.3 Pattern Discovery Algorithms

A common intermediate representation of HLS is a set of data-flow graphs (DFGs), where a graph node represents a primitive operation (e.g. addition, multiplication, etc.), and graph edges denote the sequences of operations translated from a given design specification [52]. *Top-k Pattern Discovery* is the task of identifying the k subgraphs which correspond to those patterns that are present in the highest numbers of DFGs. The rationale is that the more frequent a pattern is, the more likely it can be shared across multiple DFGs by properly multiplexing the inputs to the pattern. The pattern-discovery task is similar to the following *Frequent Subgraph Mining* task taken from Data-Mining research [188]:

Task 1. *Given a set $D = \{G_0, G_1, \ldots, G_n\}$ of n graphs, **support**(g) denotes the number of graphs (in D) in which g is a subgraph. Find any subgraph g such that **support**(g) is at least as big as a given minimum support threshold.*

To reduce the top-k pattern-discovery task to Task 1, we can take the following naive procedure: *(i)* set the minimum support threshold to be equal to two, *(ii)* sort all the returned subgraphs by their support values, and *(iii)* pick the top-k subgraphs as the patterns. However, since task 1 is computationally intractable [188], this naive procedure is inherently inefficient. In practice, in the field of Data Mining there exist various algorithms working well for task size $|D|$ up to

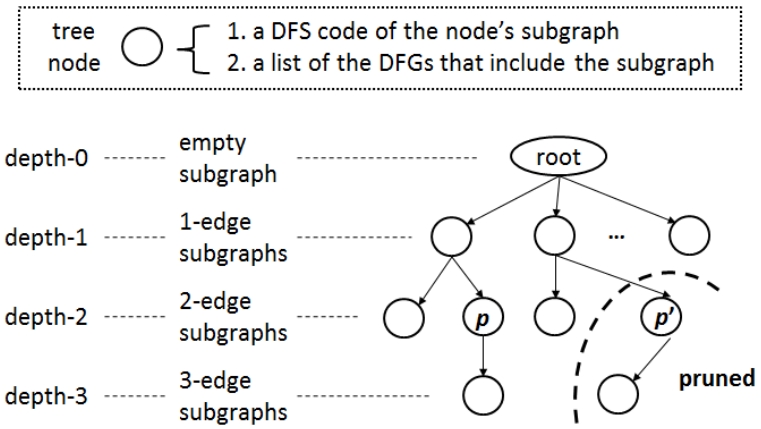


Figure 7.5: An example DFS-Code Tree.

several hundreds, with each $G_i \in D$ including several tens of nodes and edges. Among the popular algorithms, gSpan [188] stands out for several good properties, which can be leveraged to further enhance the algorithm for fast top-k pattern discovery.

7.3.1 The gSpan Algorithm

When performing a depth-first search (DFS) on a connected graph, the discovery times of the graph nodes forms a linear order [44]. As a result, a connected graph can be encoded with this order, aka *DFS code* [188]. But there can be multiple DFS codes for one particular connected graph. Hence, gSpan introduces a *DFS lexicographic order* among subgraphs and maps each subgraph to a *minimum DFS code* as its canonical representation. Two subgraphs are isomorphic if and only if their minimum DFS codes are identical [188].

Using the DFS-code representation, gSpan searches in a *DFS-Code Tree* for frequent subgraphs. An example DFS-Code Tree is illustrated in Fig. 7.5. Each node in the tree contains a DFS-code representation and a list of DFGs that include the DFS code as a subgraph. In addition, a node at depth i corresponds to a subgraph with i edge(s). Thus, the root node contains an empty subgraph and a list of all the DFGs; each depth-1 node corresponds to a one-edge subgraph and a list of DFGs that include the one-edge subgraph. By the DFS lexicographic order, gSpan can systematically enumerate valid child nodes for each parent node and then generate their lists of DFGs accordingly.

Property 1. *[188] The pre-order traversal of DFS-Code Tree follows the DFS lexicographic order.*

According to Property 1, as long as a tree node p' is not minimum (in terms of the DFS lexicographic order [188]), p' (and all its descendant nodes) can be pruned away during the traversal, because there must exist a minimum node p that is isomorphic to p' and has been traversed/enumerated (see Fig. 7.5). The combined enumeration and pruning scheme avoids the Subgraph Isomorphism Test, which is NP-complete, thereby making the gSpan search very efficient.

At each tree node g, gSpan counts $\mathbf{support}(g)$ by the size of g's list of DFGs and then outputs subgraph g if its support value meets the minimum threshold[1].

7.3.2 Enhanced gSpan

To find the top-k patterns directly, we propose an *enhanced* gSpan algorithm, which does not need to first find out and then sort *all* the frequent patterns. This characteristic is particularly appealing, as our design methodology prefers small k's in order to reduce the number of iterations in Fig. 7.3. Moreover, due to the recursive nature of the gSpan algorithm, the enhanced gSpan can scale very well in terms of both CPU time and memory usage. We enhance the gSpan algorithm based on the following property.

Property 2. *[188] The support value of a parent node is greater than or equal to that of its individual child node.*

Therefore, during the pre-order traversal, the enhanced gSpan maintains a heap [44] of size k that stores the current top-k support values. At each tree node g, if $\mathbf{support}(g)$ is less than the current minimum support stored in the heap, then the node is pruned. Thus, only the top-k patterns will be returned at the end of the traversal. The DFS-Code Tree searched by the enhanced gSpan is guaranteed to be a sub-tree of the tree searched by the original gSpan. As a result, the enhanced gSpan is even more efficient.

Property 3. *The support value of any node is upper bounded by the total number of DFGs.*

[1]When counting support values, we use the terms *node* and *subgraph* interchangeably.

Table 7.1: Benchmark Characteristics

Benchmark	Application	Input Token	Output Token	top1 Pattern	top2 Pattern
ADPCM	DSP	2 16-bit int	1 16-bit int	Multiplier & Shifter	Multiplier & Comparator
AES	Security	64 16-bit int	32 16-bit int	3 XORs	Shifter & AND & array access
BLOWFISH	Security	40 8-bit char	40 8-bit char	2 Adders & 2 XORs & AND	6 Shifters & (6 ORs or 8 ANDs)
DFSIN	Arithmetic	1 64-bit int	1 64-bit int	2 Adders	Shifter & Comparator
GSM	DSP	160 16-bit int	8 32-bit int	Multiplier & Adder	Multiplier & Adder & Shifter & AND
JPEG	DSP	128 16-bit int	64 16-bit int	8 Multipliers & 4 Adder & 4 Shifters	6 Multipliers & 4 Adders & 4 Shifters
MIPS	Processor	2 32-bit int	1 32-bit int	Multiplier & 2 ANDs & Shifter	2 Adders & Shifter
MOTION	DSP	13 16-bit int	8 16-bit int	2 Adders & Shifter	2 Adders
SHA	Security	16 32-bit int	5 32-bit int	4 Adders	(1) 2 XORs (2) 3 ANDs & 2 ORs (3) 2 ANDs & OR & NOT

This property can be proved by combining Property 2 and the fact that the root node of DFS-Code Tree contains a list of all the DFGs. Therefore, during the pre-order traversal, if *(i)* the heap has already stored k support values and *(ii)* the current minimum value among them equals the total number of DFGs, then the algorithm can stop the traversal immediately and directly return the current top-k patterns.

Our code-analysis tool based on the enhanced gSpan algorithm also outputs all the line numbers (of the original specification) where a given pattern appears, so that designers can quickly locate the code regions that are interesting for refactoring.

7.4 Experimental Results

We applied our methodology and tool to the CHStone suite [84] that includes nine benchmarks[2]: (1) adaptive differential pulse code modulation encoder for voice compression (ADPCM), (2) encryption function of advanced encryption standard (AES), (3) encryption function of data encryption standard (BLOWFISH), (4) double-precision floating-point sine function involving addition, multiplication, and division (DFSIN), (5) linear predictive coding analysis of global system for mobile communications (GSM), (6) JPEG image decompression (JPEG), (7) simplified MIPS processor (MIPS), (8) motion vector decoding of MPEG-2 (MOTION), and (9) transformation function of secure hashing algorithm (SHA). These benchmarks span over four application domains (see Table 7.1

[2] The DFSIN benchmark contains three other sub-benchmarks DFADD, DFMUL, and DFDIV.

and [84] for more details).

For top-k pattern discovery, we adapted the LLVM front-end to output DFGs and feed them to the enhanced gSpan algorithm. The primitive operations in the LLVM-generated DFGs were annotated with their locations in the source code for easy lookup. For each of the benchmarks, the enhanced gSpan can discover the top-20 patterns within just a few seconds.

In our experiments, we explored a two-objective *area vs. effective-latency* design space. The *effective latency* is defined as the product of the number of clock cycles and the length of the clock period. In other words, an effective latency is the total elapsed time for: reading an input token, processing the token according to the application, and generating an output token. The token sizes of each benchmark are listed in Table 7.1.

To trade-off between area and effective latency, we applied various loop manipulations (breaking, partial/full unrolling with different numbers of loop body, and pipelining with different initiation intervals and pipeline stages) on the major loops of each benchmark and selected the Pareto-dominating micro-architectures for each. Loop manipulation has been shown to be very effective for generating rich design alternatives [112; 165; 160]. Each of the selected micro-architectures was further swept by running HLS with several target clock-period settings. We refer to the synthesized implementations that cannot be further improved in area without sacrificing effective latency (or vice versa) as the Pareto implementations. A Pareto implementation in the "area vs. effective latency" space is also called a Pareto point. A set of Pareto points is called a Pareto front.

The DSE results are plotted in Fig. 7.6, where `base`, `top1`, `top2`, and `top1_2` stand for the design specifications in its original form, refactored with the top-1 pattern, with the top-2 pattern, and with both the top-1 and -2 patterns, respectively. These patterns are also listed in Table 7.1. Take the MOTION benchmark in Fig. 7.6 for example: its `top1` Pareto front covers effective latencies (ns) in [1.8, 2.2] and areas (um^2) in [3940, 4042] and its `top2` Pareto front covers effective latencies in [1.8, 5.7] and areas in [4192, 5163]. Across the nine benchmarks, the Pareto points render a design space with an average 1.6X span in area and an average 2.9X span in effective latency. We regard the Pareto points in the *(i)* upper-left, *(ii)* close to the origin, and *(iii)* lower-right regions in the design space as *(i)* performance-critical, *(ii)* area-performance-balanced, and *(iii)* area-efficient implementations, respectively.

All the synthesis results were obtained using a state-of-the-art commercial HLS tool with an

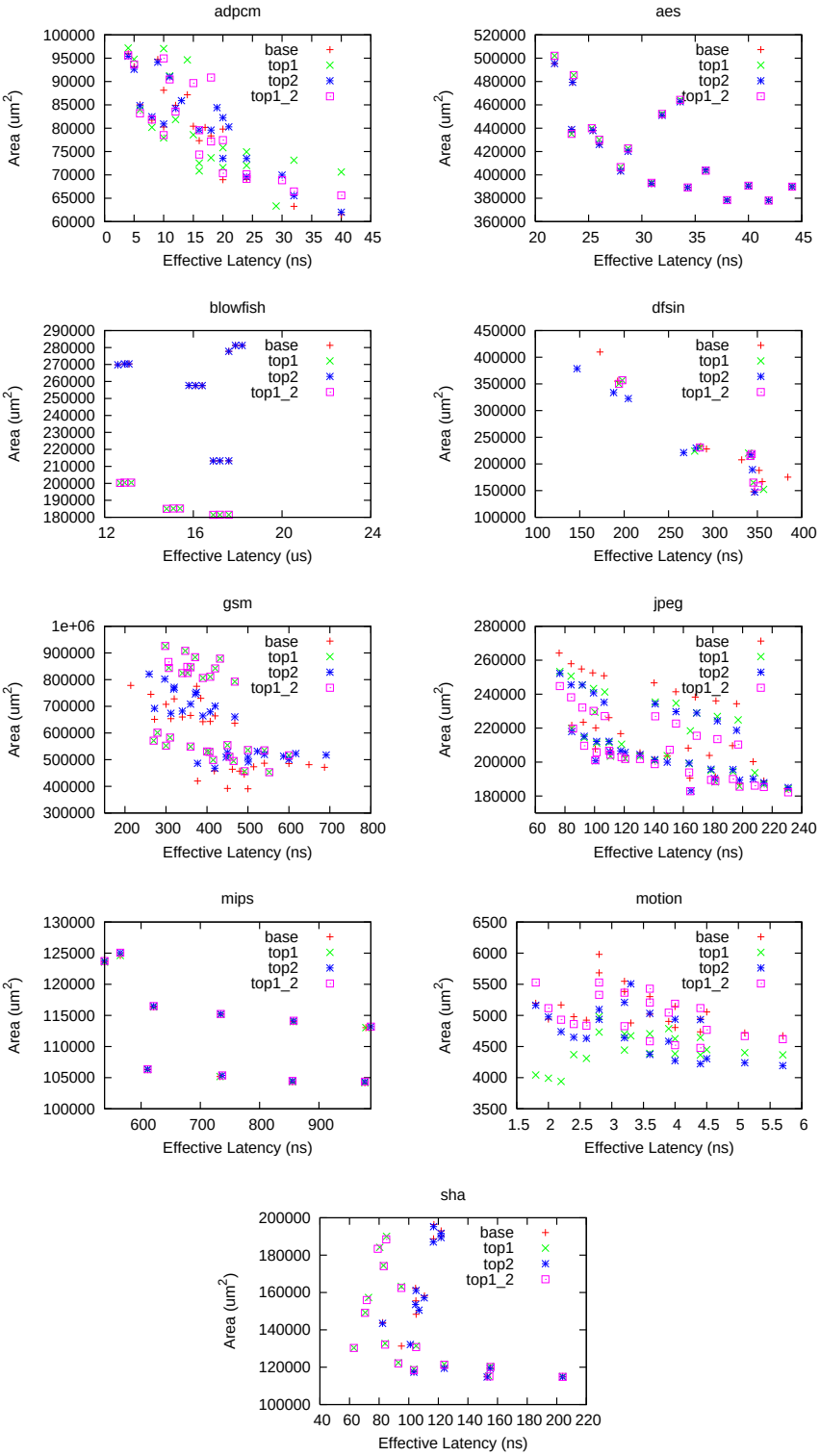


Figure 7.6: DSE results on the CHStone suite. Each point represents a distinct implementation synthesized by HLS.

industrial 45nm standard-cell library. Note that this HLS tool is unable to discover frequent patterns within its synthesis engine, but is able to treat custom functions as composite operations and synthesize the functions hierarchically.

7.4.1 DSE Result Improvements

We refer to the Pareto points from `base` specifications as the *original* Pareto points, the ones from refactored specifications as the *new*, and the two combined as the *final*. We evaluate the DSE result improvements with respect to three metrics:

M1: the percentage of the new Pareto points contributing to the final. Take the ADPCM benchmark in Fig. 7.6 for example: eight out of the eleven (73%) final Pareto points come from the refactored specifications: four from `top1`, two from `top2`, and two from `top1_2`.

M2: the maximum area/latency improvements given by the new Pareto points with respect to their closest original. Take the GSM benchmark in Fig. 7.6 for example: its `top1` Pareto point with effective latency around 270ns is 12% smaller than the closest `base` Pareto point from above and also 29% faster than the closet `base` Pareto point on the right.

M3: the minimum achievable area/latency by the new Pareto points over the original. Take the SHA benchmark in Fig. 7.6 for example: its minimum effective latency is 62ns from `top1`. Compared with the `base` minimum effective latency of 82ns, the improvement is 25%.

Intuitively, higher M1 suggests better design specifications for DSE; larger M2 indicates deeper Pareto fronts; and greater M3 provides broader usage of the design as an IP across various system-level design requirements (i.e. a metric to measure the IP reusability). All the improvements are summarized in Table 7.2, where the "-" mark stands for "no change". Next, we discuss the DSE result improvements.

M1: the size of the final Pareto set ranges from 3 to 10 points with an average of 6.11. Out of the final points, 68% on average come from the refactored specifications. Four benchmarks (BLOWFISH, JPEG, MOTION, and SHA) are even 100% improved by the refactoring. At the low end, the MIPS from CHStone suite cannot be improved because of its very simple behavior and structure. Note that the MIPS is described as a simple switch statement that performs primitive operations based on the opcodes. It has none of the advanced features that are common to modern processors, such as branch prediction, hazard detection, out-of-order execution, simultaneous multi-threading,

Table 7.2: DSE Result Improvements

	M1		*M2*		*M3*	
Benchmark	—Final—	% New	Area	Latency	Area	Latency
ADPCM	11	73%	10%	25%	-	-
AES	7	14%	1%	0%	-	-
BLOWFISH	3	100%	12%	8%	15%	-
DFSIN	8	88%	9%	15%	-	15%
GSM	8	38%	12%	29%	-	-
JPEG	7	100%	8%	0%	4%	-
MIPS	5	0%	0%	0%	-	-
MOTION	3	100%	20%	8%	14%	-
SHA	4	100%	9%	25%	-	25%
Average	6.22	68%	9%	12%	4%	5%

etc. However, in general we expect the refactoring to be effective for optimizing complex processor designs.

M2: the maximum area improvement by refactoring is up to 20% with an average of 9%, and the latency improvement is up to 29% with an average of 12%. Overall, refactoring successfully improves eight out of the nine benchmarks in terms of area and six out of the nine benchmarks in terms of latency. Notably, the improvements apply to performance-critical or area-performance-balanced implementations, as we see very marginal area-latency trade-offs for area-efficient implementations (the long tails in Fig. 7.6).

M3: refactoring improves the minimum achievable area of three benchmarks (BLOWFISH, JPEG, and MOTION) by 4%–15% and the minimum latency of two benchmarks (DFSIN and SHA) by 15%–25%. The area improvement is due to course-grained resource sharing. On the other hand, the latency improvement is due to the shorter clock periods that HLS can achieve. Note that these improvements also imply greater IP reusability as these refactored designs can sustain more stringent requirements and thus support more system-level designs.

7.4.2 HLS Runtime Reduction

As described in Section 7.2, the function customization with patterns enables hierarchical synthesis, which could reduce the HLS scheduling problem size in terms of the total number of operations to be scheduled, ultimately reducing the HLS runtime. In order to analyze the average runtime per

Table 7.3: Reduction of Average Initial Operations and Average HLS Runtime (Minute)

	avg_op							*avg_cpu*						
Benchmark	base	top1	diff	top2	diff	top1_2	diff	base	top1	diff	top2	diff	top1_2	diff
ADPCM	193	193	0%	193	0%	193	0%	30	29	-3%	29	-3%	28	-6%
AES	74	74	0%	74	0%	74	0%	27	27	0%	27	0%	27	0%
BLOWFISH	358	182	-49%	358	0%	182	-49%	371	320	-14%	353	-4%	332	-11%
DFSIN	221	226	2%	253	14%	233	5%	182	171	-6%	184	1%	174	-4%
GSM	836	616	-26%	790	-5%	620	-25%	97	58	-40%	112	15%	58	-40%
JPEG	362	311	-14%	320	-11%	269	-25%	25	26	4%	27	8%	28	12%
MIPS	17	17	0%	17	0%	17	0%	16	16	0%	16	0%	16	0%
MOTION	18	14	-22%	17	-5%	16	-11%	4	4	0%	3	-25%	3	-25%
SHA	82	46	-43%	94	14%	58	-29%	8	8	0%	8	0%	8	0%
Average			-16%		0%		-14%			-7%		0%		-8%

HLS run (*avg_cpu*), we also measured the average number of initial primitive operations (*avg_op*) that are generated by the HLS tool from an input specification. During scheduling and binding in HLS, the operations may be duplicated and/or combined for optimization. Therefore, although *avg_op* is a key factor that may affect HLS runtime, the actual runtime also depends on other factors such as the complexity of the control-data flow graph, the synthesis algorithm, the timing constraint, etc.

Table 7.3 lists the *avg_op* and *avg_cpu* for each benchmark with various specifications. Since each benchmark has three refactored versions (`top1`, `top2`, and `top1_2`), there are 27 refactored entries across the nine benchmarks. We observed 13 out of 27 entries (48%) with less *avg_op* compared with `base` and 12 out of 27 (44%) with less *avg_cpu*; on the other hand, only 4 out of 27 (8%) with higher *avg_op* and 6 out of 27 (22%) with higher *avg_cpu*. In general, this trend meets our expectation that less *avg_op* could lead to less *avg_cpu*.

The average *avg_cpu* reductions for the nine benchmarks are 7%, 0%, and 8% with `top1`, `top2`, and `top1_2`, respectively. For the top-3 benchmarks that require the longest HLS runtime (BLOWFISH, DFSIN, and GSM), the average *avg_cpu* reduction is 20% and 15% with `top1` and `top1_2`, respectively, which is a significant improvement. For the benchmarks that turned out needing longer HLS runtime with refactoring (DFSIN, GSM, and JPEG), the HLS engine may have terminated early on the `base` specification after seeing little optimization potential [105]. In this regard, we do see in the previous subsection that the refactored versions can indeed produce better DSE results (at the cost of longer HLS runtime).

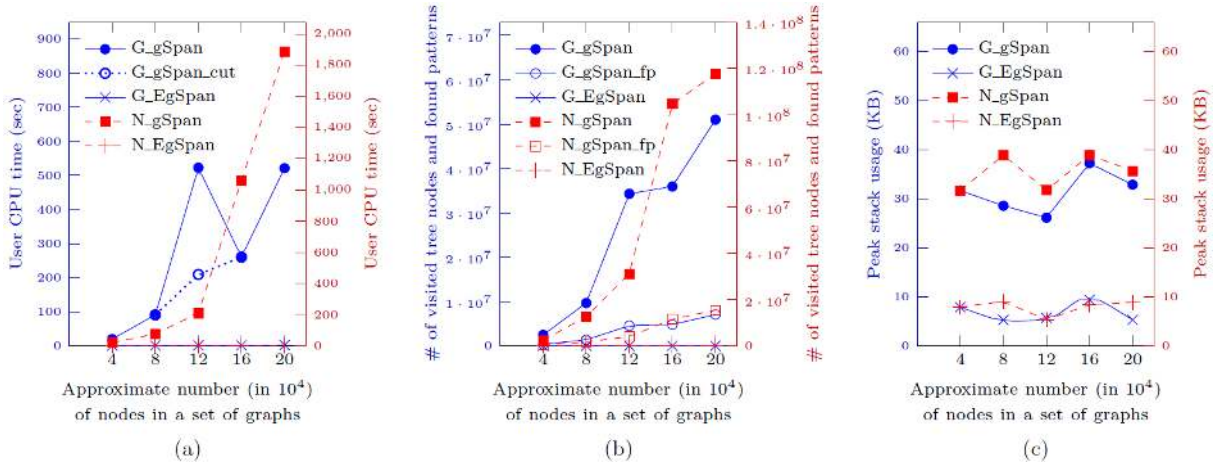


Figure 7.7: (a) User CPU time for the execution of gSpan and EgSpan. (b) Numbers of visited DFS Code Tree nodes and numbers of found patterns. (c) Peak stack memory usage. For all plots, the left vertical axis corresponds to the G group of input sets (where the number G of graphs in a set varies), and the right vertical axis corresponds to the N group of input sets (where the average number N of nodes in each graph varies.) The horizontal axes represent the size of an input set, which is equal to $G \cdot N$.

7.4.3 gSpan vs. Enhanced gSpan

To evaluate the scalability of the enhanced gSpan algorithm, we also performed experiments for gSpan and the enhanced gSpan with large randomly generated input graphs. We used TGFF [62] to generate an extensive collection of directed acyclic graphs. Then we observed trends in CPU time and memory usage for gSpan and the enhanced gSpan with respect to the size of an input set, as presented in Fig. 7.7.

The size of an input set depends on two main factors: the number G of graphs in the input set and the number N of nodes in each graph. In Fig. 7.7, the horizontal axes represent the total number of nodes in a set, which is equal to $G \cdot N$. We constructed two groups of the input sets, the G group (marked in blue) where N is averaged at about 200 and G varies from 200 to 1000 by 200-steps, and the N group (marked in red) where G is fixed at 200 and N varies from around 200 to 1000. There are 20 types of nodes (primitive operations), and they follow a Gaussian distribution with mean $(20 - 1)/2 = 9.5$ and standard deviation $20/5 = 4$.

Fig. 7.7(a) shows the average user CPU times (on Linux with 2.667 GHz cores) for the execution

of gSpan and the enhanced gSpan (referred to as EgSpan hereafter in the discussion). The left vertical axis represents the CPU time for the G group of input sets. Each marked point represents an average over ten input sets. Time for gSpan (filled circle) generally increases with the input size, except for the size of $120,000$ where $N \approx 200$ and $G = 600$. In fact, among the ten input sets of this size, one particular set takes about ten times longer than others. The empty circle, which follows the increasing trend, indicates the average CPU time excluding this outlier. Fitting a power law relation to this data (including the outlier) yields 'CPU_time $= 23.61 \cdot \text{input\_size}^{2.0396}$(+ constant)' with $R^2 = 0.8721$ (R^2 is the coefficient of determination for measuring the fitting quality. $R^2 = 1$ indicates a perfect fitting). This means that the CPU time for gSpan increases roughly at the rate of a square of the input size. On the other hand, EgSpan invocations with $k = 10$, 20, 30, and 40 all finish within 2.5 seconds on average. One with $k = 20$ is depicted with '$\times$' markers at the bottom of the plot area. A power curve fit for this data is 'CPU_time $= 0.4667 \cdot \text{input\_size}^{0.8631}$(+ constant)' with $R^2 = 0.8733$, implying a slower than linear increase in CPU time for EgSpan. Similarly, the N group's results are depicted with square markers for gSpan and '+' markers for EgSpan with $k = 20$, according to the scale of the right vertical axis. The power curves for gSpan and EgSpan are 'CPU_time $= 14.589 \cdot \text{input\_size}^{2.8795}$' ($R^2 = 0.9577$), and 'CPU_time $= 0.3659 \cdot \text{input\_size}^{0.8367}$' ($R^2 = 0.8793$), respectively.

The CPU time for gSpan increases more rapidly for the N group since its input sets have larger graphs (in the number of nodes), resulting in a generally larger DFS-Code Tree. Both gSpan and EgSpan search a DFS-Code Tree, as described in the Section 7.3, but EgSpan has an additional pruning condition (by Property 2) and an early termination condition (by Property 3). Fig. 7.7(b) illustrates the average numbers of visited DFS-Code Tree nodes and the patterns found by gSpan and EgSpan. The numbers of nodes visited by gSpan for the G group are marked by filled circles and the numbers of found patterns are marked by empty circles, according to the left vertical axis. The power curves are determined as 'DFS_nodes $= 2,675,199 \cdot \text{input\_size}^{1.9414}$' ($R^2 = 0.9642$) and as 'found_patterns $= 376,807 \cdot \text{input\_size}^{1.9113}$' ($R^2 = 0.9682$). In contrast to this phenomenon, the numbers of nodes visited by EgSpan with $k = 20$ are depicted with '$\times$' markers at the bottom, and the numbers of found patterns are constant at 20. The curve 'DFS_nodes $= 3969.9 \cdot \text{input\_size}^{0.2446}$' fits to this data with $R = 0.9982$, indicating a very slow increase in the number of visited nodes. This is due to the additional pruning effect, while an early termination is not observed for these

input sets. EgSpan terminates early, however, for randomly generated sets with even larger graphs or with smaller number of node (operation) types, for which gSpan takes much longer. Furthermore, unlike EgSpan that returns the top-k patterns, gSpan returns all the found patterns unsorted. Thus, in addition to the time gap shown in Fig. 7.7(a), extra time on the order of $n \log n$ is needed for the gSpan approach for sorting (where n increases at the rate of about 1.9). Similarly, the results for the N group are presented with filled and empty square markers for visited nodes and found patterns by gSpan, according to the right vertical axis. The exponents for the curve fitting are 2.5104 ($R^2 = 0.9853$) and 2.4147 ($R^2 = 0.9915$), respectively. These are greater than the exponents for the G group, which matches the CPU time results.

As gSpan searches a less pruned DFS-Code Tree, it requires more memory space to store the states. Fig. 7.7(c) compares the peak stack usage by gSpan and EgSpan. The usages were measured using Valgrind. For both the G group and the N group, gSpan utilizes three to five times larger stack memory space than EgSpan with $k = 20$. gSpan's peak heap memory usage is also $0 \sim 20\%$ greater for the G group and $7 \sim 12\%$ greater for the N group, ranging from 3.8 MB for small inputs to 20.0 MB for large inputs. For this experiment, both gSpan and EgSpan read in text files of size $1.0 \sim 5.7$ MB containing the input graphs information, which seem to take up the majority of the whole program memory space. Nonetheless, EgSpan, as demonstrated above, performs at a sustainable CPU-time cost despite a considerable increase in the input size, while gSpan fails to do so.

7.5 Remarks

Limitations. Our pattern-based refactoring methodology is effective in optimizing computa-tion-intensive applications thanks to their heavy requirements of arithmetic and logic operations and to the need for sharing/multiplexing these operations for resource optimization. However, it is less effective for applications with simple behavior simply because they have intrinsically few composite operations. The methodology is also less effective for control-/memory-dominated applications such as AES because the pattern-based refactoring cannot affect the data movement in the application.
Related Work. The closest related works lie in the fields of *(i)* code-refinement methodology and *(ii)* automatic DSE with CAD tools. We discuss first these two categories and then related works

on pattern discovery.

Code-refinement methodology is an emerging research topic [172; 194; 199]. Stitt *et al.* [172] introduced for the first time a set of C-based manual refinement guidelines for *performance* improvement (but not including pattern-based refactoring). Zeng and Huss [194] presented VHDL-AMS refinement techniques for making behavior models *synthesizable* while maintaining the model readability. Recently, Zuo *et al.* [199] proposed loop-transformation techniques for stencil/matrix applications, aiming for FPGA *area* minimization. Overall, our work is complementary to these works as it introduces pattern-based refactoring for area-performance co-optimization. Our refactoring technique can additionally reduce the average HLS tool runtime.

Automatic DSE with CAD tools is also an active area of research [8; 11; 18; 77; 85; 98; 112; 145; 160]. The value of automatic DSE is to minimize the number of invocations of synthesis/simulation tools for Pareto-front generation. However, the effectiveness of automatic DSE with HLS is restricted to the knobs provided by the HLS tool and their applications to the input design specification. Our work enables the extension of the tool-explorable design space through a code-refactoring technique to improve the quality of the input specification. In addition, the refactored specification requires less average HLS tool runtime, which is not achievable by automatic DSE.

Pattern discovery *within* HLS was first patented in 2007 [93] and later studied by Cong and Jiang [41] and by Hadjis *et al.* [80]. These HLS works focus on synthesizing a *single* RTL implementation that minimizes *area* under given timing constraints. Instead, our goal is to optimize a synthesizable C-based specification for area-performance co-optimization, soft-IP reuse, and HLS-runtime reduction. Moreover, for the pattern-discovery task, our work presents a different formulation that seeks directly the top-k patterns, without the need to list and sort all patterns satisfying either a frequency constraint [41] or a size constraint [80]. The latter two formulations do not fit into our methodology because a small minimum-frequency/size threshold may produce too many patterns for running multiple HLS (inefficient), while a large minimum threshold may end up finding very few patterns (ineffective). In practice, the users would not know a proper threshold in advance, whereas the selection of k is very intuitive.

Many other variants of the pattern-discovery task can also be found in the instruction-set extension literature [73]. However, these problem formulations generally require pattern input/output constraints [13; 191] or pattern size constraints [185]. Some algorithms only discover single-output

patterns [40; 140], search patterns in only a single DFG [19], or require heavy Subgraph Isomorphism Test [28; 40]. Although our top-k pattern-discovery task might be reduced to these existing problem formulations as outlined by the procedure in Section 7.3, we have empirically demonstrated the scaling advantage of a native top-k pattern-discovery algorithm, such as the enhanced gSpan, in terms of both CPU time and memory usage.

Concluding Remarks. In this chapter, we defined the problem of Code Refactoring for Enhanced Design Exploration with High-Level Synthesis. To solve this problem, we presented a pattern-based code-refactoring methodology, which brings advantages on expanding soft-IP reusability, improving HLS QoR, and reducing HLS-tool runtime. Our methodology is supported by a native top-k pattern-discovery algorithm, which empirically scales with large input graphs.

Chapter 8

Future Research Directions

Modern VLSI chips are embracing application-specific components (i.e. accelerators) for energy efficiency. Unfortunately, the hardware designers who know how to design optimized accelerators are often a distinct group from the software programmers who develop new applications [87]. By automatically supervising synthesis tools, SDSE has been able to reduce and simplify the learning curve of using synthesis tools for developing application-specific hardware. Looking forward, SDSE also paves the ways for *(i)* creating a platform for developing and deploying synthesis tools, *(ii)* promoting collaborative design exploration at an unprecedented scale, and *(iii)* exploring new classes of systems beyond VLSI chips and tools for electronic design automation.

8.1 SDSE as a Platform

SDSE can be implemented as a platform for in-house CAD teams to deploy and test new synthesis knobs and/or libraries on production designs, as Figure 8.1 illustrates. SDSE allows a small number of experimental knob settings (and/or libraries) to be run as part of DSE, enabling a DevOps-like deployment pipeline [69]. These new experimental knob settings can be either provided to designers as part of the standard synthesis runs or run directly by the CAD team. In both cases SDSE should be able to collect synthesis-run statistics and keep a history of results for all designs. The results can be analyzed off-line to determine a knob setting's impact on design quality as well as trends across multiple knob settings and designs.

Another usage of SDSE is to deploy SDSE as a third-party platform. This SDSE platform

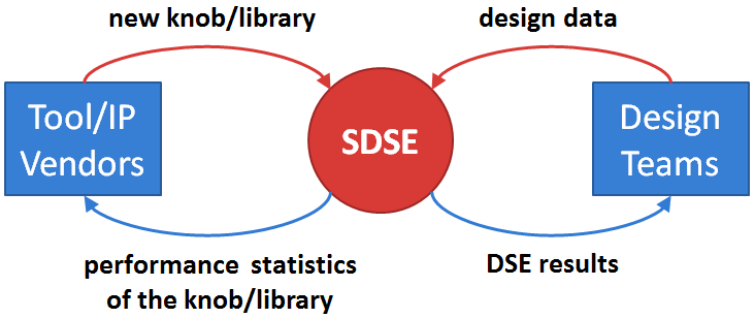


Figure 8.1: SDSE as a platform.

enables a new degree of collaboration between EDA vendors and design houses. Traditionally, the EDA vendors develop their tools based on in-house test designs, which typically do not capture well the characteristics of real designs. On the other hand, the design houses are reluctant to share their designs with the vendors, who have design know-how and also collaborate with other design houses. As a third-party platform, SDSE can resolve this concern by treating EDA tools as black boxes and reading only their QoR (e.g. performance metrics) but not necessarily their outcome (e.g. synthesized RTL). On this SDSE platform, EDA vendors can optimize *hidden knobs* (i.e. the parameters used within the tools but not exposed to the tool users) based on real customer designs. Also, design houses can use this SDSE platform for optimizing *design flows* that combine tools with different versions, features, and/or vendors.

Challenges. A designer-friendly platform can simplify the creation of application-specific hardware. To make SDSE a platform requires a standardized tool-user interface, i.e., application programming interfaces for both CAD tool developers and their users (i.e. the designers). Furthermore, in order to make SDSE a third-party platform that can serve multiple EDA vendors and design houses on limited computing resources, the platform should leverage cloud infrastructures for the submission, execution, and management of CAD-tool runs. Underlying the platform, the design-QoR data (a form of Big Data) will pose unprecedented analytics challenges (but also new opportunities) to both the CAD and design communities: *(i)* what makes a good knob? *(ii)* when and where to apply a knob to a given design? *(iii)* how to refine the design towards better QoR (this requires to locate design bottlenecks first). However, SDSE alone is not enough for maximizing the power of the envisioned platform. The next section sketches a complementary approach to SDSE.

8.2 Collaborative Design-Space Exploration

SDSE is fully automatic because it relies on synthesis tools for design optimization. However, the design space SDSE can explore and the Pareto set it can obtain are limited by the features that a synthesis tool can support. There exist many other design-optimization techniques that can be semi-automatically exploited, for example, by refactoring the design specification (Chapter 7), changing the underlying platform (ASIC vs. FPGA), and/or expanding the underlying libraries (with different functional units or technology nodes). All these optimizations are regarded as design *revision* techniques (or *virtual* knobs). Clearly, the revision-explorable space is larger than the synthesis-explorable space.

To progress from a synthesis-explorable space to another that permits better synthesis QoR, it is often necessary to apply multiple design revisions. Although some of the revision techniques may eventually be absorbed by synthesis tools, others are arguably hard to be due to unclear cost functions, high problem complexities, or limited computing resources [172]. Therefore, designer's expertise and involvement remain essential to achieve optimal design implementations. Moreover, collaborative engineering is essential for team-based design exploration, which is already the norm in the case of complex SoC designs. However, the tool support for inspiring a designer's creativity in revising the design has long been paid little attention.

In order to *assist* (as opposed to *replacing*) designers, Collaborative Design-Space Exploration (CDSE) should include two ingredients.

- **Revision Ranking**: given a design specification, CDSE suggests a set of revision techniques associated with their estimated synthesis-QoR improvements. Revision ranking assists both novice designers with a portfolio of revision techniques and experienced designers with chances to escape from local optima.

- **Revision Planning**: given a design specification and a revision database that accumulates revision history of legacy designs done by experienced designers, CDSE suggests reference sequences of revision techniques. Revision planning assists designers with multiple revision techniques that may be applied together as a bigger step, in order to accelerate the entire revision process.

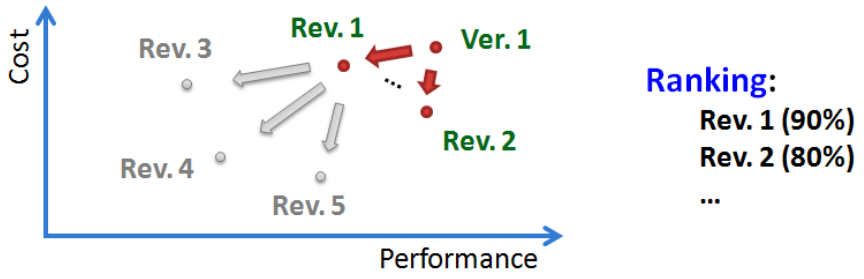


Figure 8.2: CDSE by revision ranking.

```
float coef[128];
void initCoef() { // initialize coef }
void fir() { // fir filter using coef }
void f() {
  initCoef();
  fir();
}
main() {
  // other code sharing coef
  f();
  for(int i=0; i<128; i++)
    r = r*coef[i] + pi;
  ...
}
```
(a)

```
// new function
int mac(float a, float b, float c) {
  return a * b + c;
}
// modified function
main() {
  ...
  for(int i=0; i<128; i++)
    r = mac(r, coef[i], pi);
  ...
}
```
(b)

```
// new function
void wrapper(const float c[128]) {
  fir(c);
}
// modified function
void fir(const float c[128]) {
  // fir filter using const coef
}
void f() {
  initCoef();
  wrapper(coef);
}
```
(c)

```
// new function
void g() {
  for(int i=0; i<128; i++)
    r = mac(r, coef[i], pi;
}
// modified function
main() {
  ...
  f();
  g();
  ...
}
```
(d)

Figure 8.3: CDSE example: (a) Ver. 1: initial design. (b) Rev. 1: pattern-based customization after Ver. 1. (c) Rev. 2: conversion to constants after Ver. 1. (d) Rev. 3: function/thread parallelization after Rev. 1.

8.2.1 Revision Ranking

CDSE with revision ranking is an *interactive* process, as illustrated in Figure 8.2. Starting with an initial specification (Ver. 1), CDSE suggests a ranked list of potential revision techniques (Rev. 1, Rev. 2, etc.) with an estimated QoR improvement for each (in percentage). Then, it is the designer's turn to pick up a suggestion and revise the specification accordingly. Suppose Rev. 1 is adopted. The interactive process repeats with a new list of suggestions (e.g., Rev. 3 to Rev. 5, as shown in gray in the figure). The process terminates if no more suggestions are available, or the designer is satisfied with the current QoR after applying SDSE to the revised specification.

Motivating Example. Consider the initial specification (Ver. 1) shown in Figure 8.3(a). The `main` function calls a function `f`, which does `fir` filtering after initializing an array of coefficients (`float coef[128]` is initialized in `initCoef`). Assume the content of `coef` is read-only in `f`. Following `f`,

`main` performs a sequence of Multiplication and Accumulation (MAC) operations to compute the value of variable `r`.

Given the initial design and a set of revision techniques, CDSE may suggest two top-ranked revisions: (Rev. 1) pattern-based customization (Chapter 7; the revised code shown in Figure 8.3(b)) and (Rev. 2) conversion to constants [172] (Figure 8.3(c)). Rev. 1 groups a set of frequent, non-primitive functional units, i.e. the MACs, into a new function `mac`, such that the HLS tools can bind and share the MAC, which is optimized separately and used as a new library component. On the other hand, observing that `coef` is essentially *constant* in the context of `fir` (but may be modified elsewhere in `main`, not shown in the example), Rev. 2 introduces a wrapping function `wrapper` for running `fir`, such that advanced HLS engines can utilize *constant propagation* for optimizing the implementation of `fir`. Overall, different revisions bring different cost-performance trade-offs; therefore divergent paths emerge in the revision process (Figure 8.2). After analysis, CDSE *estimates* that the overall QoR improvement of Rev. 1 would be greater than that of Rev. 2 (i.e. revision ranking). The ranking marks the end of the first iteration of CDSE.

The second iteration starts after the code is actually revised by the designer, say, Rev. 1 as in Figure 8.2. At this point, CDSE may propose another revision suggestion (Rev. 3) such as function/thread parallelization [74], an optimization that aims for a coarser granularity of resource sharing above the functional-unit level. Figure 8.3(d) shows Rev. 3 that combines functional-unit customization (Rev. 1) and function/thread parallelization. The parallelization is done by grouping the sequence of MAC operations into a new function `g`, which is independent from function `f`; thus, `f` and `g` can be scheduled to run in parallel for better performance. Similarly, CDSE may also provide other suggestions that follow Rev. 1 (e.g., Rev. 4, Rev. 5 and others that are not shown in Figure 8.2). Again, CDSE ranks these suggestions by their estimated QoR improvements, and then hands over the decision to the designer.

Challenges. The effectiveness of CDSE with revision ranking relies on the identification of commonly-effective revision techniques that are not integrated into general synthesis tools. Moreover, the analysis engines that estimate the revision QoR improvements need to be efficient and scalable, providing rapid suggestions during the interactive process. Examples include the engines for pattern-based customization (Chapter 7), communication channel reordering [58], and memory system configuration [144]. Stitt et al. discussed other revision techniques that are believed hard

to be automated for *implementation* [172]. However, from CSDE's perspective, the question of whether these techniques are hard to be *analyzed* remains open.

8.2.2 Revision Planning

A modern SoC is a complex system with a large number of heterogeneous components. Big engineering teams are involved in the exploration of its large design space. CDSE with revision planning aims at pioneering the design methodologies and tools that promote collaborative engineering. The planning should be based on *(i)* a *revision database* that can accumulate expertise on commonly-used design patterns over time and *(ii)* a *learning model* that can synthesize useful information from the revision database.

Motivating Example. Consider again the examples of Figure 8.3, assuming the same revision names as in Figure 8.2. Suppose the initial design is a DSP application that presents some design patterns that are similar to those that have been used by other engineers working on this project or on previous projects. For instance, DSP applications commonly: *(i)* include sequences of MAC operations and *(ii)* use lookup tables for storing coefficients and initialize them with standard function calls to compute sine or cosine values. Equipped with a learning model applied to the knowledge of similar DSP applications, CDSE can directly suggest the best revision sequence (e.g., Rev. 1 $\rightarrow$ Rev. 4 in Figure 8.4), which consists of "pattern-based customization" and "conversion to constants". Moreover, a common co-optimization with conversion to *floating-point* constants is to use a *fixed-point* library for better synthesis QoR. This latter revision is represented as Rev. 5 in Figure 8.4. Hence, CDSE will suggest also a revision tuple (Rev. 4, Rev. 5) because it *knows in advance (via learning)* that Rev. 5 is most frequently combined with Rev. 4. Overall, a designer can take all these suggestions together to revise the initial design in one bigger step. In general, the length of the sequence can be specified by the user: the shorter the sequence, the higher the estimation accuracy that can achieve. In the long run, the ability of CDSE to suggest revision will improve as more and more design specifications and designers' decisions are collected in the revision database.

Challenges. The effectiveness of CDSE with revision planning relies on the information stored in the revision database. The information should be modeled by an intermediate representation that is capable to capture interesting design patterns. Furthermore, an effective learning model

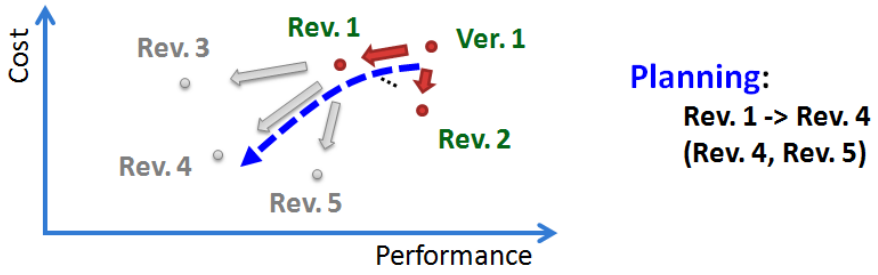


Figure 8.4: CDSE with revision planning.

(perhaps hybrid models are required) is crucial for the analysis of designs that come from different application domains, revised by different designers, implemented with different synthesis tools, and released at different stages in a design cycle. Also, a user-friendly interface for comparing the current design and the referenced designs is important for the designer to evaluate the credibility of revision planning.

8.3 Beyond Synthesis and SoC-Design

SDSE was invented for SoC design, especially for SoC's computation components that primarily rely on synthesis tools for implementation. Nevertheless, the SDSE approach should be extensible to other cases of design exploration, e.g. design explorations based not only on the use of synthesis tools but also *simulation* tools. Also, SDSE can be extended to discover the Pareto set of other SoC components such as the network on chip (NoC). Existing multi-objective NoC-exploration works (e.g. [109]) are based on local-search algorithms and do not consider the real network traffics generated from the software stack. The real traffics can be better captured using full-system simulators such as virtual platforms. Another SDSE extension is to consider non-traditional design objectives in exploring an SoC component, such as design manufacturability. There exist works that use Monte Carlo simulation to discover manufacturability-aware Pareto sets [96; 135; 177]. However, these works either use local-search algorithms [177], assume differentiable and convex cost functions [96], or treat the system design as a whole object [135]. Overall, the NoC and manufacturability simulations (like synthesis) are parameterizable and computation-intensive. Therefore, a compositional-exploration approach (like SDSE) that is assisted with learning models

Table 8.1: Computer-Science Applications of Pareto-Set Discovery.

Application (Publication Year)	Area	Design Objectives
Parameter Tuning (2012) [154]	Algorithm	Quality, Runtime
Mesh Simplification (2013) [30]	Graphics	Accuracy, Simplicity
Optical Flow Computation (2014) [55]	Vision	Accuracy, Runtime
Motion Control (2013) [5]	Animation	Effort, Height
Robot Design (2013) [176]	Robotics	Speed, Energy, Stability
Model Selection (2006) [174]	Learning	Accuracy, Complexity
Attack Detection (2013) [130]	Security	Risk, Surveillance, Cost
Resource Management (2012) [143]	Cloud	Energies, Performance
Anaphora Resolution (2011) [68]	Linguistics	Multiple Scoring Metrics
Community Detection (2014) [131]	Network	Heterogeneous Connection Types

for simulation-result estimation should be able to approximate the Pareto sets much more efficiently.

Beyond SoC design, Pareto-set discovery has been also becoming popular in general computer-science research. In particular, I surveyed ten recent such applications across different areas of computer science. These applications are briefly described below and summarized in Table 8.1.

- Parameter tuning in *algorithm design* is the optimization of meta-heuristics (such as the temperature scheduling of Simulated Annealing) for the trade-off between solution quality and computation runtime. Typically, the quality can improve at the cost of more runtime, but the return would diminish in a long run. The trade-off is therefore useful for choosing problem-specific meta-heuristics for solving computationally-intractable problems such as the Traveling Salesman.

- Mesh simplification in *computer graphics* aims to reduce the number of faces used in a 3D polygonal surface model while keeping the overall shape, boundaries, and volume. Obviously, the model accuracy and simplicity are conflicting goals: more faces preserve the model accuracy but incur more computation for model visualizations and simulations, and vice versa.

- Optical flow computation in *computer vision* is a technique for estimating displacement and

velocity fields. The computation requires several integration and differential equations, which ultimately require to be approximated by solving numerical methods. The numerical methods are iterative methods, which (like meta-heuristics) require trade-offs between their approximation accuracy and computation runtime.

- Motion control in *computer animation* requires the pre-computation of a family of motion controllers for real-time physics-based simulation. For a motion such as a jump, the solution family of interest is described by the Pareto set that defines the trade-off between jump effort and jump height.

- Robot design in *robotics* involves multiple competing objectives, e.g., performance metrics of the physical system such as speed and energy efficiency. Moreover, one problem when teleoperating a robot is that the large movement of the robot head, which houses a camera, can cause operators to become disorientated. Therefore, the stability is also a common robot-design objective.

- Model selection in *machine learning* is to select a learning model that has high classification accuracy with low model complexity (i.e. short model evaluation time). For instance, the model complexity of a Support Vector Machine (SVM) is expressed by its number of support vectors. An SVM that achieves high classification rates with many support vectors may be over-fitting its training examples and requires longer evaluation time for classifying unknown examples. The trade-off is considered crucial for applications such as bioinformatics and computational biology [83].

- Attack detection in *computer security* is based on distributed multi-agent cooperation for detecting malware or cyber attacks. Each agent can either scan or not scan the target under protection (e.g. a website), and every configuration of two agents is assigned a risk value and a surveillance value: both "scan" ("not scan") result in lowest (highest) risk but highest (lowest) surveillance. Also, the cost of scan varies agent by agent. Overall, an optimal agent configuration is a trade-off among risk (security), surveillance (privacy), and cost.

- Resource management in *cloud computing* makes the decisions of service migration and placement across data centers. It aims to balance the trade-offs among conflicting optimization

objectives such as renewable energy consumption, cooling energy consumption and response time performance.

- Anaphora resolution in *computational linguistics* (aka natural language processing) is a feature-selection problem. There is no generally accepted metric for measuring the performance of anaphora resolution systems. Systems optimized according to one metric tend to perform poorly with respect to other ones, making it very difficult to compare anaphora resolution systems. The multi-objective approach is to optimize the systems according to multiple metrics simultaneously.

- Community detection in *social network analysis* is based on the network-edge semantics, e.g. indicating friend relationships. However, there are heterogeneous types of connections, e.g. the correspondence in an email dataset vs. the semantic correlation in the email body. These distinct groups of edges are referred to as *layers* (i.e. the union of all layers is the original network). An example multi-objective analysis is to bisect the network into equal parts in a way that attempts to minimize the cut-size on each layer. Thus, the Pareto-optimal cut sizes across multiple layers might be useful in determining differences in structure between layers.

Across these ten applications the number of design knobs can be ten or more, and for most of the applications the time required for evaluating the knob-setting qualities (i.e. design qualities) is short (a few seconds). So far, all these applications are handled using local-search algorithms and they treat the design as a whole object. One exception is the robot-design application, whose design evaluation time can be long (tens of minutes to experiment with prototype robots), so its optimization has adopted learning models for estimating the robot qualities. Although machine-learning methods have not been extensively adopted with multi-objective design exploration, they have already become popular for individual design tasks in areas such as graphics [81], vision [147], security [35], cloud [86], linguistics [39], and network [54].

Following the evolution of computer system design introduced in Chapter 1, general scientific/engineering designs will eventually become more complex (i.e. powerful), featuring more components within the entire design and involving more interactions with the physical world. Therefore, the creation and evaluation of a design configuration will inevitably become more time-consuming because of the growing system complexity and physical constraints. I consider SoC a successful

forerunner of complex system design, and I believe that the principles of SDSE can benefit the design of future complex systems beyond SoCs.

Chapter 9

Conclusions

I have presented Supervised Design-Space Exploration (SDSE), a novel abstraction layer for orchestrating multiple timing-consuming synthesis jobs for component-based design. SDSE has been applied to the design of heterogeneous hardware accelerators as well as high-performance processor cores, which all rely on synthesis tools for automatic implementation. Besides performance, these designs demand cost-effective implementations in terms of both design quality (e.g. area or power) and time to market (e.g. synthesis effort). However, current decade-old synthesis-centric design methodologies suffer from: *(i)* long synthesis tool runtime, *(ii)* elusive optimal setting of many synthesis knobs, *(iii)* limitation to one design implementation per synthesis run, and *(iv)* limited capability of digesting only component-level designs as opposed to holistic system-wide synthesis. SDSE addresses these challenges by adapting dynamically to the most influential component designs and to the most promising knob settings for synthesizing them, ultimately approximating the system Pareto set by combining component-synthesis results.

For system-level design exploration, I have presented two SDSE frameworks (and their companion algorithms) for supervising High-Level Synthesis (HLS) and Logic Synthesis (LS) tools, respectively. These frameworks can autonomously approximate a system Pareto set given limited information: only lightweight component characterization is required, yet the necessary component synthesis jobs are discovered on-the-fly. The frameworks can work together by using HLS for exploring micro-architectures and using LS for determining clock periods. In parallel the HLS-based framework can work with designs initially specified in high-level languages, while the LS-based framework targets traditional RTL design entries.

For component-level design exploration, I have presented two SDSE frameworks (and their companion algorithms) for supervising HLS and Physical Synthesis (PS) tools, respectively. These frameworks can autonomously approximate a component Pareto set by iteratively refining the approximation, guided by synthesis-result estimation with learning models. The capability of adapting to promising knob settings can significantly improve the Pareto approximation quality given a limited budget of synthesis jobs. In particular, I have successfully integrated SDSE algorithms into an IBM PS tuning system, yielding 20% better macro performance than the original system on the design of a 22nm server processor that is currently in production.

SDSE is extensible with "virtual" knobs, such as specification restructuring, and opens new research revenues for synthesis-driven Design-Space Exploration (DSE). I have presented a pattern-based code-refactoring methodology for enhancing HLS-driven DSE. The methodology can enhance the Pareto approximation quality, increase design reusability, and reduce average HLS tool runtime. More generally, based on SDSE I have sketched future research directions to incorporate more "design revision" techniques beyond the refactoring methodology and to promote collaborative design engineering to a larger degree. If it turns out to be successful, I believe this theme of research will drive a new wave of innovative uses of computing, thanks to the reduced and simplified learning curve of electronic system design.

Even more generally, I regard the techniques developed and the lessons learned as a valuable asset for complex system design beyond VLSI. There are increasing interests in applying Pareto-set discovery to a diverse set of computer applications. These applications have not yet taken advantage of system compositionality and have rarely used learning models to guide the exploration process. In general, as computer applications are becoming more powerful and sophisticated, involving a greater number of components interacting in complex behaviors, I believe that SDSE has advanced the frontier of DSE research that can benefit general computer-aided design.

Part IV

Bibliography

Bibliography

[1] 2014 Wallcharts: SDA, Multi-Platform Based Design, ESL, CAE & CAD-CAM. *Gary Smith EDA*.

[2] Klocwork. *See demo at http://www.klocwork.com/resources/videos/extract-and-inline-function-refactoring-for-c*.

[3] Opencores. `www.opencores.org/`.

[4] Accellera. SystemC. *Available at http://www.accellera.org*, 2015.

[5] S. Agrawal and M. van de Panne. Pareto optimal control for natural and supernatural motions. In *Proc. of Motion on Games*, pages 7:29–7:38, 2013.

[6] AMD. Advanced Micro Devices, Inc. 2014.

[7] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the 1967 Spring Joint Computer Conference (AFIPS)*, pages 483–485, 1967.

[8] Y. Ando, S. Shibata, S. Honda, H. Tomiyama, and H. Takada. Fast design space exploration for mixed hardware-software embedded systems. In *Proc. of the International System-on-Chip Conference (SOCC)*, pages 92–95, Nov 2011.

[9] H. Angepat, D. Chiou, E. Chung, and J. Hoe. *FPGA-Accelerated Simulation of Computer Systems*. Morgan & Claypool, 2014.

[10] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[11] G. Ascia, V. Catania, A. G. D. Nuovo, M. Palesi, and D. Patti. Efficient design space exploration for application specific systems-on-a-chip. *Journal of Systems Architecture*, 53(10):733–750, 2007.

[12] M. Ashouei, J. Hulzink, M. Konijnenburg, J. Zhou, F. Duarte, A. Breeschoten, J. Huisken, J. Stuyt, H. de Groot, F. Barat, J. David, and J. Van Ginderdeuren. A voltage-scalable biomedical signal processor running ECG using 13pJ/cycle at 1MHz and 0.4V. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 332–334, Feb 2011.

[13] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of the Design Automation Conference (DAC)*, pages 256–261, June 2003.

[14] O. Azizi, A. Mahesri, J. P. Stevenson, S. J. Patel, and M. Horowitz. An integrated framework for joint design space exploration of microarchitecture and circuits. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 250–255, Mar. 2010.

[15] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer-Verlag, 2006.

[16] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *Proc. of the Congress on Evolutionary Computation (CEC)*, volume 1, pages 773–780, Sept 2005.

[17] H. Bauer, J. Veira, and F. Weig. Moore's law: Repeal or renewal? `http://www.mckinsey.com/insights/high_tech_telecoms_internet/moores_law_repeal_or_renewal`, 2013.

[18] G. Beltrame, L. Fossati, and D. Sciuto. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 29(7):1083–1095, July 2010.

[19] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1–6, April 2007.

[20] U. Bordoloi, H. P. Huynh, S. Chakraborty, and T. Mitra. Evaluating design trade-offs in customizable processors. In *Proc. of the Design Automation Conference (DAC)*, pages 244–249, July 2009.

[21] S. Borkar. Design perspectives on 22nm CMOS and beyond. In *Proc. of the Design Automation Conference (DAC)*, pages 93–94, 2009.

[22] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[23] J. Branke, K. Deb, K. Miettinen, and R. Slowinski, editors. *Multiobjective Optimization, Interactive and Evolutionary Approaches*, Lecture Notes in Computer Science. Springer, 2008.

[24] R. Brayton and J. Cong. Nsf workshop on EDA: Past, present, and future (part 2). *IEEE Design & Test*, 27(3):62–74, May 2010.

[25] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[26] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[27] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 318–325, Nov 1997.

[28] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, pages 262–269, 2002.

[29] D. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[30] B. R. Campomanes-Álvarez, O. Cordón, and S. Damas. Evolutionary multi-objective optimization for mesh simplification of 3D open models. *Integrated Computer-Aided Engineering*, 20(4):375–390, Oct. 2013.

[31] J. Campos and J. M. Colom. A reachable throughput upper bound for live and safe free choice nets. In *Proc. of the International Conference on Application and Theory of Petri Nets*, pages 237–256, Gjern, Denmark, June 1991.

[32] Y. Cao, J. Velamala, K. Sutaria, M.-W. Chen, J. Ahlbin, I. Sanchez Esqueda, M. Bajura, and M. Fritze. Cross-layer modeling and simulation of circuit reliability. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 33(1):8–23, Jan 2014.

[33] R. Caruana, N. Karampatziakis, and A. Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *Proc. of the International Conference on Machine learning (ICML)*, pages 96–103, 2008.

[34] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *Proc. of the Design Automation Conference (DAC)*, pages 789–794, 2002.

[35] P. K. Chan and R. P. Lippmann. Machine learning for computer security. *Journal of Machine Learning Research*, 7:2669–2672, Dec. 2006.

[36] S. Chandra, A. Raghunathan, and S. Dey. Variation-aware voltage level selection. *IEEE Trans. on Very Large Scale Integration Systems*, 20(5):925–936, May 2012.

[37] A. Cicalini, S. Aniruddhan, R. Apte, F. Bossu, O. Choksi, D. Filipovic, K. Godbole, T.-P. Hung, C. Komninakis, D. Maldonado, C. Narathong, B. Nejati, D. O'Shea, X. Quan, R. Rangarajan, J. Sankaranarayanan, A. See, R. Sridhara, B. Sun, W. Su, K. van Zalinge, G. Zhang, and K. Sahota. A 65nm CMOS SoC with embedded HSDPA/EDGE transceiver, digital baseband and multimedia processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 368–370, 2011.

[38] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[39] M. Collins. Tutorial: Machine learning methods in natural language processing. In B. Schlkopf and M. Warmuth, editors, *Learning Theory and Kernel Machines*, volume 2777 of *Lecture Notes in Computer Science*, pages 655–655. Springer Berlin Heidelberg, 2003.

[40] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *Proc. of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 183–189, 2004.

[41] J. Cong and W. Jiang. Pattern-based behavior synthesis for FPGA resource reduction. In *Proc. of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 107–116, 2008.

[42] 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*, 30(4):473–491, 2011.

[43] H. Cook and K. Skadron. Predictive design space exploration using genetically programmed response surfaces. In *Proc. of the Design Automation Conference (DAC)*, pages 960–965, 2008.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[45] C. Cortes and M. Mohri. On transductive regression. In *Advances in Neural Information Processing Systems (NIPS)*, pages 305–312, 2006.

[46] R. Courtland. The origins of Intel's new transistor, and its future. *IEEE Spectrum*, 2011.

[47] R. Courtland. Memory in the third dimension. *IEEE Spectrum*, 51(1):60–61, 2014.

[48] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test*, 26(4):8–17, 2009.

[49] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA multi-CPU and GPU system-on-chip. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 56–57, 2012.

[50] A. Darabiha, J. Rose, and W. J. MacLean. Video-rate stereo depth measurement on programmable hardware. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 203–210, June 2003.

[51] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[52] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[53] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6(2):182–197, Apr. 2002.

[54] M. Dehmer and S. C. Basak, editors. Wiley, 2012.

[55] J. Delpiano, L. Pizarro, R. Verschae, and J. Ruiz-del-Solar. Multi-objective optimization for characterization of optical flow methods. In *Proc. of the International Conference on Computer Vision Theory and Applications*, volume 2, pages 556–573, 2014.

[56] R. H. Dennard, J. Cai, and A. Kumar. A perspective on todays scaling challenges and possible future directions. *Solid-State Electronics*, 51(4):518–525, 2007.

[57] R. H. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[58] G. Di Guglielmo, C. Pilato, and L. P. Carloni. A design methodology for compositional high-level synthesis of communication-centric SoCs. In *Proc. of the Design Automation Conference (DAC)*, pages 128:1–128:6, 2014.

[59] I. Diakonikolas. *Approximation of Multiobjective Optimization Problems*. PhD thesis, Columbia University, 2010.

[60] I. Diakonikolas and M. Yannakakis. Succinct Approximate Convex Pareto Curves. In *Proc. of the Symposium on Discrete Algorithms (SODA)*, pages 74–83, 2008.

[61] I. Diakonikolas and M. Yannakakis. Small approximate Pareto sets for bi-objective shortest paths and other problems. *SIAM Journal on Computing*, 39:1340–1371, 2009.

[62] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 97–101. IEEE Computer Society, 1998.

[63] X. Dong, J. Zhao, and Y. Xie. Fabrication cost analysis and cost-aware design space exploration for 3-D ICs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 29(12):1959–1972, Dec 2010.

[64] H. Drucker, C. Burges, L. Kaufman, A. Smola, and V. Vapnik. Support vector regression machines. In *Advances in Neural Information Processing Systems*, volume 9, pages 155–161, 1997.

[65] EDAC. Electronic Design Automation Consortium. 2014.

[66] T. Eeckelaert, T. McConaghy, and G. Gielen. Efficient multiobjective synthesis of analog circuits using hierarchical Pareto-optimal performance hypersurfaces. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1070–1075, 2005.

[67] M. Ehrgott. *Multicriteria optimization*. Springer-Verlag, 2005.

[68] A. Ekbal, S. Saha, O. Uryupina, and M. Poesio. Multiobjective simulated annealing based approach for feature selection in anaphora resolution. In I. Hendrickx, S. Lalitha Devi, A. Branco, and R. Mitkov, editors, *Anaphora Processing and Applications*, volume 7099 of *Lecture Notes in Computer Science*, pages 47–58. Springer Berlin Heidelberg, 2011.

[69] F. Erich, C. Amrit, and M. Daneva. Cooperation between information system development and operations: A literature review. In *Proc. of the International Symposium on Empirical Software Engineering and Measurement*, pages 69:1–69:1, 2014.

[70] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[71] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 1991.

[72] J. H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38:367–378, 1999.

[73] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Trans. on Reconfigurable Technology and Systems*, 4(2):18:1–18:28, May 2011.

[74] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, June 2011.

[75] M. Geilen and T. Basten. A calculator for Pareto points. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 285–290, 2007.

[76] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[77] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip. *IEEE Trans. on Very Large Scale Integration Systems*, 10(4):416–422, Aug. 2002.

[78] J. B. Gosling. *Simulation in the Design of Digital Electronic Systems*. Cambridge University Press, 1993.

[79] A. Greenfield. *Everyware: The Dawning Age of Ubiquitous Computing*. Peachpit Press, 2006.

[80] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski. Impact of FPGA architecture on resource sharing in high-level synthesis. In *Proc. of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 111–114, 2012.

[81] P. M. Hall. Introduction to machine learning for computer graphics. In *ACM SIGGRAPH 2014 Courses*, pages 20:1–20:33, 2014.

[82] P. Hallschmid and R. Saleh. Fast design space exploration using local regression modeling with application to ASIPs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):508–515, Mar. 2008.

[83] J. Handl, D. B. Kell, and J. Knowles. Multiobjective optimization in bioinformatics and computational biology. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 4(2):279–292, Apr. 2007.

[84] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[85] C. Haubelt and J. Teich. Accelerating design space exploration using Pareto-front arithmetics. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 525–531, Jan. 2003.

[86] E. Hormozi, H. Hormozi, M. Akbari, and M. Javan. Using of machine learning into cloud environment (a survey): Managing and scheduling of resources in cloud systems. In *Proc. of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 363–368, Nov 2012.

[87] M. Horowitz. Computing's energy problem (and what we can do about it). In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 10–14, Feb 2014.

[88] C. Hu. 3D FinFET and other sub-22nm transistors. In *IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits*, pages 1–5, 2012.

[89] IBS. International Business Strategies, Inc. 2014.

[90] Intel. Intel Corporation. 2014.

[91] ITRS. International Technology Roadmap for Semiconductors. *Available at http://public.itrs.net*, 2009.

[92] H. Javaid, X. He, A. Ignjatovic, and S. Parameswaran. Optimal synthesis of latency and throughput constrained pipelined MPSoCs targeting streaming applications. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–84, 2010.

[93] M. Jensen. Data path synthesis apparatus and method for optimizing a behavioral design description being processed by a behavioral synthesis tool. US Patent 7,305,650.

[94] H. Jones. Why the technology road maps for the semiconductor industry will be different. In *SEMI Industry Strategy Symposium (ISS)*, Jan 2014.

[95] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

[96] S. Jung, J. Lee, and J. Kim. Yield-aware Pareto front extraction for discrete hierarchical optimization of analog circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1437–1449, Oct 2014.

[97] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, Mar. 2004.

[98] J. Keinert, M. Streubuhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. on Design Automation of Electronic Systems*, 14(1):1–23, Jan. 2009.

[99] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Proc. of the Design Automation Conference (DAC)*, pages 341–347, 1987.

[100] R. Keyes. The impact of Moore's Law. *IEEE Solid-State Circuits Society Newsletter*, 11(5):25–27, 2006.

[101] H.-E. Kim, J.-S. Yoon, K.-D. Hwang, Y.-J. Kim, J.-S. Park, and L.-S. Kim. A 275mW heterogeneous multimedia processor for IC-stacking on si-interposer. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 128–130, Feb 2011.

[102] Y. Kim, L. John, I. Paul, S. Manne, and M. Schulte. Performance boosting under reliability and power constraints. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 334–341, Nov 2013.

[103] P. Kollig, C. Osborne, and T. Henriksson. Heterogeneous multi-core platform for consumer multimedia applications. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1254–1259, Apr. 2009.

[104] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic performance-constrained pipelining in high-level synthesis. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1–6, Mar. 2011.

[105] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Exploiting area/delay tradeoffs in high-level synthesis. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1024–1029, 2012.

[106] W. Kruijtzer, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin. Industrial IP integration flows based on IP-XACT™ standards. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 32–37, 2008.

[107] T. Kutzschebauch and L. Stok. Regularity driven logic synthesis. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 439–446, Nov 2000.

[108] L. Lavagno, G. Martin, and L. Scheffer. *Electronic Design Automation for Integrated Circuits Handbook*. CRC Press, Inc., 2006.

[109] G. Leary, K. Srinivasan, K. Mehta, and K. Chatha. Design of Network-on-Chip architectures with a genetic algorithm-based technique. *IEEE Trans. on Very Large Scale Integration Systems*, 17(5):674–687, May 2009.

[110] T. Leemhuis. What's new in Linux 3.10. `http://www.h-online.com/open/features/What-s-new-in-Linux-3-10-1902270.html`, 2013.

[111] R. Leupers, F. Schirrmeister, G. Martin, T. Kogel, R. Plyaskin, A. Herkersdorf, and M. Vaupel. Virtual platforms: Breaking new grounds. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 685–690, 2012.

[112] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proc. of the Design Automation Conference (DAC)*, June 2013.

[113] H.-Y. Liu, I. Diakonikolas, M. Petracca, and L. P. Carloni. Supervised design space exploration by compositional approximation of Pareto sets. In *Proc. of the Design Automation Conference (DAC)*, pages 399–404, June 2011.

[114] Y. Liu, M. Yoshioka, K. Homma, T. Shibuya, and Y. Kanazawa. Generation of yield-embedded Pareto-front for simultaneous optimization of yield and performances. In *Proc. of the Design Automation Conference (DAC)*, pages 909–912, 2010.

[115] C. A. Mack. Fifty years of Moore's Law. *IEEE Trans. on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[116] P. Madden. Dispelling the myths of parallel computing. *IEEE Design & Test*, 30(1):58–64, Feb 2013.

[117] J. Maggot. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters*, pages 7–13, 1984.

[118] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 37–48, 2012.

[119] D. Marculescu and A. Iyer. Application-driven processor design exploration for power-performance trade-off analysis. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 306–313, Nov 2001.

[120] G. Mariani, A. Brankovic, G. Palermo, J. Jovic, V. Zaccaria, and C. Silvano. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In *Proc. of the Design Automation Conference (DAC)*, pages 120–125, 2010.

[121] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano. OSCAR: An optimization methodology exploiting spatial correlation in multicore design spaces. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):740–753, May 2012.

[122] E. J. Marinissen, B. Prince, D. Keltel-Schulz, and Y. Zorian. Challenges in embedded memory design and test. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 722–727, Mar. 2005.

[123] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test*, 26(4):18–25, july-aug. 2009.

[124] A. Mathur, M. Fujita, E. Clarke, and P. Urard. Functional equivalence verification tools in high-level synthesis flows. *IEEE Design & Test*, 26(4):88–95, July 2009.

[125] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[126] S. Naffziger, B. Stackhouse, and T. Grutkowski. The implementation of a 2-core multi-threaded Itanium-family processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 182–183, Feb. 2005.

[127] U. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-Core 64-Thread 64b power-efficient SPARC SoC. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 108–590, 2007.

[128] D. Nenni and P. McLellan. *Fabless: The Transformation of the Semiconductor Industry*. CreateSpace Independent Publishing Platform, 2014.

[129] J. Oh, J. Park, G. Kim, S. Lee, and H.-J. Yoo. A 57mW embedded mixed-mode neuro-fuzzy accelerator for intelligent multi-core processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 130–132, Feb 2011.

[130] T. Okimoto, N. Ikegai, K. Inoue, H. Okada, T. Ribeiro, and H. Maruyama. Cyber security problem based on multi-objective distributed constraint optimization technique. In *Proc. of the Conference on Dependable Systems and Networks Workshop (DSN)*, pages 1–7, June 2013.

[131] B. Oselio, A. Kulesza, and A. Hero. Multi-objective optimization for multi-level networks. In W. Kennedy, N. Agarwal, and S. Yang, editors, *Social Computing, Behavioral-Cultural Modeling and Prediction*, volume 8393 of *Lecture Notes in Computer Science*, pages 129–136. Springer International Publishing, 2014.

[132] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[133] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Efficient system design space exploration using machine learning techniques. In *Proc. of the Design Automation Conference (DAC)*, pages 966–969, 2008.

[134] G. Palermo, C. Silvano, and V. Zaccaria. ReSPIR: A response surface-based Pareto iterative refinement for application-specific design space exploration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, Dec. 2009.

[135] G. Palermo, C. Silvano, and V. Zaccaria. Variability-aware robust design space exploration of chip multiprocessor architectures. In *Proc. of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 323–328, Jan 2009.

[136] M. Palesi and T. Givargis. Multi-objective design space exploration using genetic algorithms. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 67–72, 2002.

[137] C. Papadimitriou and M. Yannakakis. On the Approximability of Trade-offs and Optimal Access of Web Sources. In *Proc. of the Symposium on Foundations of Computer Science*, pages 86–92, 2000.

[138] J. Park, I. Hong, G. Kim, Y. Kim, K. Lee, S. Park, K. Bong, and H.-J. Yoo. A 646GOPS/W multi-classifier many-core processor with cortex-like architecture for super-resolution recognition. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 168–169, Feb 2013.

[139] Y. Park, C. Yu, K. Lee, H. Kim, Y. Park, C. Kim, Y. Choi, J. Oh, C. Oh, G. Moon, S. Kim, H. Jang, J.-A. Lee, C. Kim, and S. Park. 72.5GFLOPS 240Mpixel/s 1080p 60fps multi-format video codec application processor enabled with GPGPU for fused multimedia application. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 160–161, Feb 2013.

[140] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli. Automatic instruction set extension and utilization for embedded processors. In *Proc. of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 108–118, June 2003.

[141] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 184–185, Feb. 2005.

[142] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 157–162, 2015.

[143] D. H. Phan, J. Suzuki, R. Carroll, S. Balasubramaniam, W. Donnelly, and D. Botvich. Evolutionary multiobjective optimization for green clouds. In *Proc. of the Conference Companion on Genetic and Evolutionary Computation*, pages 19–26, 2012.

[144] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 18:1–18:10, 2014.

[145] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55:99–112, Feb. 2006.

[146] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 78(1):131–159, Jan. 2007.

[147] S. Prince. *Computer Vision: Models Learning and Inference*. Cambridge University Press, 2012.

[148] Qualcomm. Qualcomm Incorporated. 2014.

[149] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012.

[150] P. Ralph and Y. Wand. A proposal for a formal definition of the design concept. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 103–136. Springer Berlin Heidelberg, 2009.

[151] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Trans. on Software Engineering*, 6:440–449, 1980.

[152] P. Ramm, A. Klumpp, J. Weber, N. Lietaer, M. Taklo, W. De Raedt, T. Fritzsch, and P. Couderc. 3D integration technology: Status and application development. In *Proc. of the European Solid-State Circuits Conference (ESSCIRC)*, pages 9–16, 2010.

[153] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[154] J. Ries, P. Beullens, and D. Salt. Instance-specific multi-objective parameter tuning based on fuzzy logic. *European Journal of Operational Research*, 218(2):305–315, 2012.

[155] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, June 2006.

[156] Samsung. Samsung Electronics Co., Ltd. 2014.

[157] A. Sangiovanni-Vincentelli. The tides of EDA. *IEEE Design & Test*, 20(6):59–75, Nov 2003.

[158] B. C. Schafer. Hierarchical high-level synthesis design space exploration with incremental exploration support. *to appear in IEEE Embedded Systems Letters*, 2015.

[159] B. C. Schafer, T. Takenaka, and K. Wakabayashi. Adaptive simulated annealer for high level synthesis design space exploration. In *Proc. of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 106–109, Apr. 2009.

[160] B. C. Schafer and K. Wakabayashi. Design space exploration acceleration through operation clustering. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 29(1):153–157, Jan 2010.

[161] B. C. Schafer and K. Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *IET Computers Digital Techniques*, 6(3):153–159, May 2012.

[162] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[163] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proc. of the Design Automation Conference (DAC)*, pages 805–810, 1999.

[164] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, nov.–dec. 2010.

[165] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014.

[166] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, 1993.

[167] H. Shojaei, T. Basten, M. Geilen, and P. Stanley-Marbell. SPaC: a symbolic Pareto calculator. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 179–184, 2008.

[168] P. Singer. High cost per wafer, long design cycles may delay 20nm and beyond. `http://electroiq.com/petes-posts/2014/01/22/high-cost-per-wafer-long-design-cycles-may-delay-20nm-and-beyond/`, 2014.

[169] A. Singhee and P. Castalino. Pareto sampling: choosing the right weights by derivative pursuit. In *Proc. of the Design Automation Conference (DAC)*, pages 913–916, 2010.

[170] J. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[171] K. Smith, R. Everson, J. Fieldsend, C. Murphy, and R. Misra. Dominance-based multiobjective simulated annealing. *IEEE Trans. on Evolutionary Computation*, 12(3):323–342, June 2008.

[172] G. Stitt, F. Vahid, and W. Najjar. A code refinement methodology for performance-improved synthesis from C. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 716–723, Nov. 2006.

[173] L. Stok. The next 25 years in EDA: A cloudy future? *IEEE Design & Test*, 31(2):40–46, April 2014.

[174] T. Suttorp and C. Igel. Multi-objective optimization of support vector machines. In Y. Jin, editor, *Multi-Objective Machine Learning*, volume 16 of *Studies in Computational Intelligence*, pages 199–220. Springer Berlin Heidelberg, 2006.

[175] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 13 2012.

[176] M. Tesch, J. Schneider, and H. Choset. Expensive multiobjective optimization for robotics. In *Proc. of the International Conference on Robotics and Automation (ICRA)*, pages 973–980, May 2013.

[177] S. K. Tiwary, P. K. Tiwary, and R. A. Rutenbar. Generation of yield-aware Pareto surfaces for hierarchical circuit design space exploration. In *Proc. of the Design Automation Conference (DAC)*, pages 31–36, 2006.

[178] A. S. Tranberg-Hansen and J. Madsen. A compositional modelling framework for exploring MPSoC systems. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2009.

[179] C.-Y. Tsai, Y.-J. Lee, C.-T. Chen, and L.-G. Chen. A 1.0TOPS/W 36-core neocortical computing processor with 2.3tb/s Kautz NoC for universal visual recognition. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 480–482, Feb 2012.

[180] S. Vassilvitskii and M. Yannakakis. Efficiently computing succinct trade-off curves. *Theoretical Computer Science*, 348:334–356, 2005.

[181] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. de Freitas. Bayesian optimization in high dimensions via random embeddings. In *Proc. of the International Joint Conference on Artificial Intelligence*, 2013.

[182] J. Warnock, B. Curran, J. Badar, G. Fredeman, D. Plass, Y. Chan, S. Carey, G. Salem, F. Schroeder, F. Malgioglio, G. Mayer, C. Berry, M. Wood, Y.-H. Chan, M. Mayo, J. Isakson, C. Nagarajan, T. Werner, L. Sigal, R. Nigaglioni, M. Cichanowski, J. Zitz, M. Ziegler,

T. Bronson, G. Strevig, D. Dreps, R. Puri, D. Malone, D. Wendel, P.-K. Mak, and M. Blake. 22nm next-generation IBM System Z microprocessor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 1–3, Feb 2015.

[183] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S. Chu, S. Islam, and V. Zyuban. The implementation of POWER7: A highly parallel and scalable multi-core high-end server processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 102–103, 2010.

[184] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2011.

[185] C. Wolinski and K. Kuchcinski. Automatic selection of application-specific reconfigurable processor extensions. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1214–1219, March 2008.

[186] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano. A meta-model assisted coprocessor synthesis framework for compiler/architecture parameters customization. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[187] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano. SPIRIT: Spectral-aware Pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 34(1):155–159, Jan 2015.

[188] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *Proc. of the International Conference on Data Mining (ICDM)*, pages 721–724, 2002.

[189] B. Yu, J.-R. Gao, D. Ding, Y. Ban, J. seok Yang, K. Yuan, M. Cho, and D. Pan. Dealing with IC manufacturability in extreme scaling (embedded tutorial paper). In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 240–242, Nov 2012.

[190] K. Yu, J. Bi, and V. Tresp. Active learning via transductive experimental design. In *Proc. of the International Conference on Machine Learning (ICML)*, pages 1081–1088, 2006.

[191] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proc. of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, pages 69–78, 2004.

[192] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 264–266, Feb 2011.

[193] Y. Yuyama, M. Ito, Y. Kiyoshige, Y. Nitta, S. Matsui, O. Nishii, A. Hasegawa, M. Ishikawa, T. Yamada, J. Miyakoshi, K. Terada, T. Nojiri, M. Satoh, H. Mizuno, K. Uchiyama, Y. Wada, K. Kimura, H. Kasahara, and H. Maejima. A 45nm 37.3GOPS/W heterogeneous multi-core SoC. In *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 100–101, 2010.

[194] K. Zeng and S. Huss. Architecture refinements by code refactoring of behavioral VHDL-AMS models. May 2006.

[195] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar. Design space exploration of multiple loops on FPGAs using high level synthesis. In *Proc. of the International Conference on Computer Design (ICCD)*, pages 456–463, Oct 2014.

[196] M. Ziegler, R. Puri, B. Philhower, R. Franch, W. Luk, J. Leenstra, P. Verwegen, N. Fricke, G. Gristede, E. Fluhr, and V. Zyuban. POWER8 design methodology innovations for improving productivity and reducing power. In *Proc. of the Custom Integrated Circuits Conference (CICC)*, pages 1–9, Sept 2014.

[197] M. Zuluaga, A. Krause, P. Milder, and M. Püschel. "Smart" design space sampling to predict Pareto-optimal solutions. In *Proc. of Languages, Compilers, Tools and Theory for Embedded Systems*, pages 119–128, 2012.

[198] M. Zuluaga, A. Krause, G. Sergent, and M. Püschel. Active learning for multi-objective optimization. In *Proc. of the International Conference on Machine Learning (ICML)*, 2013.

[199] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving polyhedral code generation for high-level synthesis. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 15:1–15:10, 2013.

[200] V. Zyuban and P. Strenski. Balancing hardware intensity in microprocessor pipelines. *IBM Journal of Research and Development*, 47(5.6):585–598, Sept 2003.

Part V

Appendices

Appendix A

Prototype Tools

This appendix describes the following four prototype tools that I developed as part of my dissertation:

- *Synthesis Planning for Exploring ESL Design* (SPEED): a prototype tool that implements SDSE with High-Level Synthesis (HLS) for system-level DSE. The SPEED methodology and its companion algorithms are presented in Chapter 3.
- *Compositional Approximation of Pareto Sets* (CAPS): a prototype tool that implements SDSE with Logic Synthesis for system-level DSE. The CAPS algorithm and its companion heuristic modifications are presented in Chapter 4.
- *Learning-based Exploration with Intelligent Sampling, Transductive Experimental Design, and Refinement* (LEISTER): a prototype tool that implements SDSE with HLS for component-level DSE. LEISTER's underlying algorithms are presented in Chapter 5.
- *Enhanced gSpan* (EgSpan): a prototype tool for enhanced DSE with pattern-based code-refactoring. The code-refactoring methodology and the EgSpan algorithm are presented in Chapter 7.

Throughout the appendix, the term "effective latency" refers to the product of *(i)* the synthesized clock period of the circuit and *(ii)* the simulated clock cycle count to execute the entire task of the circuit. Moreover, a tutorial example is provided for each tool.

A.1 SPEED

This section describes the SPEED software package (`speed-p.tar.gz`), including its software requirements and file structure. A tutorial is also included to explain the usage of SPEED.

A.1.1 Software Requirements

The following software and libraries are required for the execution of SPEED. The version number enclosed by the parentheses denotes the software/library version that has been tested with SPEED.

- Cadence C-to-Silicon (CtoS) Compiler (14.10-p100, 64-bit): a commercial HLS tool that can synthesize RTL Verilog from SystemC.
- R (2.14.1): a free MATLAB-like software environment for statistical computing and graphics. R can be downloaded at `http://cran.r-project.org/`, which also provides 6K+ free statistical-computing packages written in R, such as the following two packages.
- SiZer (0.1-4): an R package for two-segment linear curve fitting.
- strucchange (1.5-0): an R package for multi-segment piecewise linear curve fitting.
- GLPK (4.45): SPEED uses the GLPK package for solving linear programming. GLPK can be downloaded at `http://www.gnu.org/software/glpk/`.
- g++ (4.6.3): The g++ compiler is used to compile SPEED's C++ code.

A.1.2 File Structure

The `speed-p.tar.gz` tarball includes the following directories.

- `prune` contains a C++ program `prune.cc` for component-level design-space pruning.
- `fit` features a bash script `fit.sh` for piecewise linear curve fitting based on the user's choice of curve-fitting methods.
- `plan` provides *(i)* a bash script `gen_tmg_template.sh` for the generation of Timed Marked Graph (TMG) models, *(ii)* a C++ program `gen_lp_model.cc` for the generation of linear

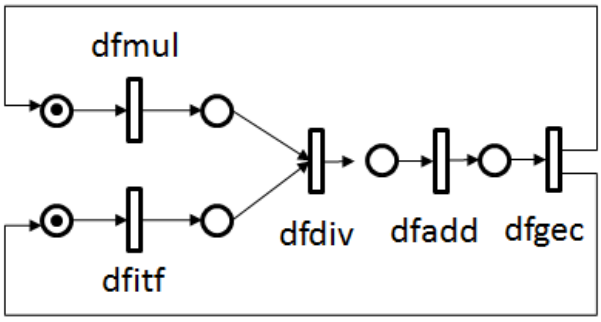


Figure A.1: The TMG model of `dfsin`.

programs given the TMG models, and *(iii)* a bash script `plan_hls.sh` for the planning of HLS jobs for each component.

- `dfsin` is an example of a system design that includes five component designs. Section A.1.3 uses `dfsin` to explain the usage of SPEED.
- `examples` contains more examples besides `dfsin`: *(i)* `mpeg`: an MPEG encoding/decoding system including fourteen components, *(ii)* `jpeg`: a JPEG decoding system including three components, and *(iii)* `rs`: a Reed-Solomon decoding system including five components.

A.1.3 Tutorial

`dfsin` is a double-precision floating-point sine approximation system with concurrent data flows, feedback paths, and multiple places with initial tokens. It is composed of five components: `dfadd`, `dfdiv`, `dfgec`, `dfitf`, and `dfmul`. The TMG model of `dfsin` is shown in Figure A.1.

A.1.3.1 Component-Level Exploration

For each component `com` of `dfsin`, take the following steps within the `com` directory for component-level DSE.

1. provide a CtoS template directory that includes four subdirectories: `src` (SystemC design), `tb` (test bench), `syn` (CtoS scripts), and `sim` (for clock-cycle-count simulation). An example template directory is provided as `_ks_template`, which can be instantiated with different knob settings, each corresponding to a distinct CtoS synthesis script.

2. run the script `./gen_test_dir.sh` to generate a test CtoS directory `ks_i_test` for each knob setting i.

3. run the scripts `./run_test_*.sh` to synthesize the component with all the knob settings. Based on the synthesis results, the user determines the minimum target clock period for each knob setting.

4. run the script `./gen_minmax_dir.sh` to generate CtoS directories with the minimum and maximum target clock periods. The example script applies a fix multiplication factor (e.g. 1.5) to the minimum period for setting the maximum period. In order to set a more accurate maximum period, the user can sweep the target clock periods with a fixed step or using binary search.

5. run the scripts `./run_min_*.sh` and `./run_max_*.sh` to synthesize the component with all the knob settings, each using either the minimum or the maximum target clock period.

6. run the script `./collect_results.sh` to obtain a Component Knob-setting Trade-off (CKT) library `com.ckt`. Each line of `com.ckt` follows the format:

$$id \quad min\_elat \quad max\_area \quad max\_elat \quad min\_area \quad lat$$

   where *id* is the knob setting ID, $min(max)\_elat$ is the minimum (maximum) effective latency, $min(max)\_area$ is the minimum (maximum) synthesized area, and *lat* is the simulated clock cycle count.

7. run the command `../../prune/prune com.ckt com.ilt com.fdt` to prune away the sub-optimal knob settings. This step generates Interval Lookup Table `com.ilt` and Curve Fitting Data `com.fdt`.

8. edit the script `../../fit/fit.sh` with the name of the component (e.g. `dfadd`) and the user choice of curve-fitting methods: `fit1`, `fit2`, or `fit3`. `fit1` does linear line fitting (for the case of only one knob setting available), `fit2` does two-segment linear fitting (sufficient for general cases), and `fit3` does multi-segment linear fitting (for the case of many knob settings available). Run the edited script `../../fit/fit.sh` to generate a (piecewise) linear fitting model `com.pwl`.

A.1.3.2 System-Level Exploration

After finishing the steps in the previous section for each component design, switch the work directory to `plan` for system-level DSE. Follow these steps:

1. edit the script `./gen_tmg_template.sh` with the name of the system design (`dfsin` in this example) and the full path to the system design directory. Run the edited script `./gen_tmg_template.sh` to generate a template TMG model file `system.tmg`.

2. edit the file `system.tmg` by replacing the "TO BE FILLED" line of each component with proper TMG connection and initial markings, following the format:

$$n \quad D_1 \quad M_1 \quad D_2 \quad M_2 \ldots D_N \quad M_n$$

   where n is the number of destination components, D_i is the name of the destination component i, and M_i is the initial marking from the current component to D_i.

3. edit the script `./plan_hls.sh` with the name of the system design and the desired characterization granularity δ (0.2 by default). In addition, turn off the option "thru_max" by setting it to 0 for *system-throughput-constrained area exploration*. Run the edited script `./plan_hls.sh` to identify the necessary component-synthesis jobs with their knob settings and target clock periods. The planning results will be returned as output files in the directory `hls_plan/`δ.

4. optionally, the user can run *system-area-constrained throughput exploration* by turned on the option "thru_max" (set to 1) in the script `./plan_hls.sh`. In addition, the user should provide a system-area range [`minarea`, `maxarea`] for the exploration. Run the edited script `./plan_hls.sh` again, and the planning results will be appended to the files in the directory `hls_plan/`δ.

After the synthesis planning, switch back to each component directory for additional component synthesis as follows.

1. edit the script `./gen_implement_dir.sh` with *(i)* the name of the component design, *(ii)* the target clock period used for testing the knob settings (i.e., the clock setting used in Step 2, Section A.1.3.1), and *(iii)* the system characterization granularity δ used in the previous Step 3. Run the edited script `./gen_implement_dir.sh` to generate CtoS synthesis directories.

2. run the script `./run_implement.sh` to synthesize the component with all the planned knob settings.

After finishing the planned synthesis jobs for all the components, switch again back to the `plan` directory for the final "plan vs. synthesis" comparisons. Run the script `./collect_hls_result.sh` after editing the system design name and the characterization granularity δ in the script. The comparisons will be returned as an output in the directory `hls_result/`δ`/`.

A.2 CAPS

This sub-appendix describes the CAPS software package (`caps-p.tar.gz`), including its software requirements and file structure. A tutorial is also included to explain the usage of CAPS.

A.2.1 Software Requirements

The following software and libraries are required for the execution of CAPS. The version number enclosed by the parentheses denotes the software/library version that has been tested with CAPS.

- Synopsys Design Compiler (I-2013.12-SP1): a commercial Logic Synthesis (LS) tool that can synthesize a netlist from RTL Verilog/VHDL.
- g++ (4.6.3): The g++ compiler is used to compile CAPS's C++ code.

A.2.2 File Structure

The `caps-p.tar.gz` tarball includes the following directories.

- `init` contains text files (`*.ini`), each describing the two extreme points of all the components in a system design.
- `data` is a directory that contains all the CAPS outputs.
- `vision` provides a tutorial example described in Section A.2.3.
- `examples` contains more examples besides `vision`: *(i)* `mpeg`: an MPEG encoding/decoding system including 26 components and *(ii)* `msoc`: an SoC based on the OR1200 procssor [3] including eight components.

A.2.3 Tutorial

This tutorial uses a system design for video-rate stereo-depth measurement (`StereoDepth`) to explain the usage of CAPS. `StereoDepth` is designed in RTL Verilog and consists of four components: `chip0`, `chip1`, `chip2`, and `chip3`.

To characterize `StereoDepth`, take the following steps.

1. under the directories `vision/chip*`, provide a template directory with the scripts to run Logic Synthesis for each component. An example template directory is provided as `template`, which can be instantiated with different target clock periods.

2. under the directory `init`, provide a text file `vision.ini` that describes the extreme points of the four components in the following format:

   ```
   min_x
   max_x0 min_y0 max_y0 Ninst0
   max_x1 min_y1 max_y1 Ninst1
   max_x2 min_y2 max_y2 Ninst2
   max_x3 min_y3 max_y3 Ninst3
   ```

   where *(i)* `min_x` is the minimum target clock period (ns) of all the components (and the system), *(ii)* `max_x`i is the maximum target clock period (ns) of `chip`i, *(iii)* `min_y`i and `max_y`i are the minimum and maximum total power (mW) of `chip`i, respectively, and *(iv)* `Ninst`i is the number of instances of `chip`i in the `StereoDepth` system (which is 1 in this example, i.e., no duplicated component). These extreme points can be obtained by running a sequence of trial Logic Synthesis up front with a coarse-grained sweeping of target clock periods. Based on the trial synthesis results, the minimum target clock period can be set to the minimum achievable clock period and the maximum target clock period can be set to the clock period whose returned circuit area starts to reach a plateau.

3. under the `caps-p` directory, run the command `./caps` ϵ `Niter init/vision.ini`, where ϵ constrains the approximation accuracy ($0 \leq \epsilon < 1$) and `Niter` is the constraint of the maximum number of synthesis jobs. Setting either ϵ or `Niter` to zero tells CAPS to ignore that constraint. See also the bash script `run_caps.sh` for command examples.

CAPS outputs the approximate Pareto sets in the text files `data/*.dat`. Each line of a `.dat` file shows the target clock period (ns) and the synthesized total power (mW). The file `data/approximation.dat` shows the system Pareto set and the file `data/error.dat` shows the approximation error per CAPS iteration.

A.3 LEISTER

This sub-appendix describes the LEISTER software package (`leister-p.tar.gz`), including its software requirements and file structure. A tutorial is also included to explain the usage of LEISTER.

A.3.1 Software Requirements

The following software and libraries are required for the execution of LEISTER. The version number enclosed by the parentheses denotes the software/library version that has been tested with LEISTER.

- Cadence C-to-Silicon (CtoS) Compiler (14.10-p100, 64-bit): a commercial HLS tool that can synthesize RTL Verilog from SystemC.
- R (2.14.1): a free MATLAB-like software environment for statistical computing and graphics. R can be downloaded at `http://cran.r-project.org/`, which also provides 6K+ free statistical-computing packages written in R, such as the following package.
- randomForest (4.6-10): an R package for classification and regression based on a forest of trees using random inputs.
- g++ (4.6.3): The g++ compiler is used to compile LEISTER's C++ code.

A.3.2 File Structure

The `leister-p.tar.gz` tarball includes the following directories.

- `gen_ks` provides a reference way for the generation of a knob-setting library for a given SystemC design. The library defines the design space that can be explored by perturbing HLS

knob settings. Since the explorable design space is application-specific, the utility scripts of `gen_ks` require a certain degree of customization for the application (SystemC code) at hand. The customization can also control the granularity of the design space to be explored.

- `train` is used for the training stage of LEISTER to generate an initial Pareto set.
- `refine` is used for the refinement stage of LEISTER to refine the Pareto-set approximation.
- `examples` contains the following examples: *(i)* `adpcm`: adaptive differential pulse code modulation encoder for voice compression, *(ii)* `aes`: encryption function of advanced encryption standard, *(iii)* `blowfish`: encryption function of data encryption standard, *(iv)* `dfsin`: double-precision floating-point sine function involving addition, multiplication, and division, *(v)* `gsm`: linear predictive coding analysis of global system for mobile communications, *(vi)* `jpeg`: JPEG image decompression, *(vii)* `motion`: motion vector decoding of MPEG-2, and *(viii)* `sha`: transformation function of secure hashing algorithm.

A.3.3 Tutorial

This tutorial uses a SystemC design of the Secure Hashing Algorithm (`SHA`) to explain the usage of LEISTER.

A.3.3.1 Knob-Setting Library Generation

To begin with, in the `gen_ks` directory an initial knob-setting library `sha.ks` and a final knob-setting library `sha.fks` for `SHA` need to be generated. For each initial knob setting, low-effort synthesis and cycle-accurate simulation are run, in order to *(i)* quickly filter out the non-synthesizable knob setting and *(ii)* obtain the clock cycle count required by the knob-setting derived micro-architecture. The clock cycle counts are then annotated with the initial library `sha.ks`. The annotated library is called the final knob-setting library `sha.fks`. Thus, in the later training and refinement stages, the remaining unsynthesized knob-settings are all synthesizable (unless there are problems caused by high-effort synthesis[1]). Moreover, cycle-accurate simulations are no longer required because a

[1] Compared with low-effort synthesis, CtoS performs additional analyses (and optimizations) such as using Logic Synthesis for evaluating detailed timing feasibility. It is possible (but not often) that a certain knob setting is synthesizable with low-effort synthesis but not with high-effort synthesis due to a too stringent clock-period constraint.

lookup into the final library `sha.fks` is sufficient to obtain the clock cycle counts. The following steps generate the initial and the final knob-setting libraries.

1. provide a CtoS template directory `_ks_template` that includes four subdirectories: `src` (SystemC design), `tb` (test bench), `syn` (CtoS scripts), and `sim` (for clock-cycle-count simulation). An example template directory is provided as `_ks_template`, which can be instantiated with different knob settings.

2. edit the script `./gen_ks.sh` according to the actual SystemC design, such that a user-desired set of knob settings can be parameterized. The example `gen_ks.sh` shows a way to generate an initial knob-setting library of `SHA` with 243 loop configurations, 4 function configurations, and 4 target clock periods, resulting in a design space of size 3,888. The parameterization also requires the corresponding editing of CtoS scripts `_ks_template/syn/syn_temp.tcl` and `_ks_template/syn/micro.tcl`. Run the edited script `./gen_ks.sh` to instantiate CtoS directories `ks_i` for `i` $\in \{1, 2, \ldots, 3888\}$, and to generate the initial knob-setting library `sha.ks`, which enumerates the 3,888 settings, each associated with a unique knob-setting ID.

3. edit the script `./get_sim_list.sh` with the name of the SystemC design (i.e. "sha" in this example). This script should be adapted from the script `./gen_ks.sh`, so that all the knob-setting IDs are consistent. Note that if the knob-settings only differ by, for instance, function-inlining and/or clock-period configurations, then only one of them is sufficient for the simulation of the clock cycle count, since all of them should have the same number of clock cycle count. Therefore, for the `SHA` example, the required number of simulations is only 243. Moreover, in `./get_sim_list.sh`, the user can define the number of simulation threads (`thread`) and the maximum number of simulation jobs per thread (`maxjob`). Run the edited `./get_sim_list.sh`, which will generate `thread` simulation scripts `sim_thread_*.sh`.

4. run in parallel the scripts `sim_thread_*.sh`, which launch low-effort syntheses and cycle-accurate simulations.

5. edit the script `./collect_lat.sh`, which again should be adapted from the script `./gen_ks.sh`

However, even if an HLS job is failed during the Pareto-refinement stage, LEISTER can analyze the HLS log to determine the exit status of the current HLS job and still progress to select the next HLS job without stopping.

for the consistency of knob-setting IDs. The script `./collect_lat.sh` will filter out non-synthesizable knob settings and annotate the simulated clock cycle counts for the synthesizable knob settings. Run the edited script `./collect_lat.sh` to generate the final knob-setting library `sha.fks`.

A.3.3.2 The Training Stage

Under the `train` directory, there are two subdirectories: *(i)* `ted` for training with the sequential TED algorithm and *(ii)* `rted` for training with the randomized TED algorithm. Both algorithms take the final knob-setting library `sha.fks` as an input and generate sample (training) knob-settings. At the end of the training stage, three files are generated:

- `sample.qor` is the training set of `sha.fks` yet additionally annotated with HLS QoR in terms of the synthesized clock period, the effective latency, and the circuit area,
- `pareto.qor` is an initial Pareto-set approximation in terms of area vs. effective latency (`pareto.qor` is also a subset of `sample.qor`), and
- `test.dat` is the set of knob settings whose HLS QoR are still unknown (i.e. `sha.fks` minus `sample.qor`).

Under the `ted` subdirectory, the training stage includes the following steps.

1. edit the R script `./_ted_temp.R` with the full path to the `ted` directory.
2. edit the script `./ted.sh` with name of the design ("sha" in this example), the number of knobs (8 for `SHA`), the number of training-sample size (20 by default), and the number of synthesis jobs per synthesis thread (5 by default). Run the edited script `./ted.sh` to generate $(20/5 = 4)$ `run_thread_*.sh` scripts.
3. run in parallel the `run_thread_*.sh` scripts to synthesize `SHA` with the sampled knob settings.
4. after all the `run_thread_*.sh` scripts have finished, edit the script `collect_qor.sh` with the number of knobs. Run the edited script `collect_qor.sh` to generate the final output files `sample.qor`, `pareto.qor`, and `test.dat`.

On the other hand, the training steps using randomized TED (in the `rted` directory) are basically the same, except for the need of editing the script `rted/rted.sh` in Step 2 by specifying the size of a random subset (`num_rted`, which is 200 by default). Repeat the above four steps in the `rted` directory with all the keyword `ted` being replaced with `rted`.

A.3.3.3 The Refinement Stage

Under the `refine` directory, there are two subdirectories: *(i)* `basic-ST` for Pareto-set refinement with Random Forests using exhaustive QoR prediction and *(ii)* `extreme-RT` for Pareto-set refinement with Random Forests using random-sampling for QoR prediction.

Under the `basic-ST` subdirectory, the refinement stage includes the following steps.

1. edit the R script `rf.R` with the full path to the `basic-ST` directory and the number of knobs (`num_knob`), which is this case is equal to 8.

2. edit the script `query.sh` with the name of the design ("sha" in this example).

3. setup the initial DSE results by creating a directory called `itr_0` and link the initial Pareto-set approximation obtained at the training stage. Run the commands `mkdir itr_0; ln -s ../../train/ted/FILE itr_0/FILE`, for `FILE` $\in$ {`sample.qor`, `pareto.qor`, `test.dat`}.

4. edit the C++ program `basic-ST.cc` by defining the number of knobs (`NUM_KNOB`, which is 8 for `SHA` in this example). Compile the edited program by running the command `make`.

5. start the Pareto-set refinement by running the command `./basic-ST itr_0/sample.qor itr_0/test.dat Niter`, where `Niter` is the user-desired number of refinement iterations (i.e. the number of HLS jobs, which in this case is equal to 20).

On the other hand, the refinement steps using the method `extreme-RT` are basically the same, except *(i)* in Step 1, the path should point to the `extreme-RT` directory, *(ii)* in Step 3, `ted` in the command should be replaced with `rted`, *(iii)* in Step 4, edit `extreme-RT.cc` by defining not only `NUM_KNOB` but also `RAND_SELECT_SIZE`, which is the size of the random subset for QoR prediction (59 by default), and *(iv)* in Step 5, the command becomes `./extreme-RT itr_0/sample.qor itr_0/test.dat Niter`.

For both methods `basic-ST` and `extreme-RT`, the final Pareto set will be output in the file `itr_Nitr/pareto.qor`. Each line of `itr_Nitr/pareto.qor` indicates one Pareto point in the following format:

$$ks\_id \quad elat \quad area$$

where ks_id is the knob-setting ID, $elat$ is the effective latency, and $area$ is the circuit area. Note that there is one Pareto-set approximation `pareto.qor` in each iteration directory `itr_Nitr` for `Nitr` being $0, 1, 2, \ldots$, so that the approximation can be tracked as opposed to being overwritten.

A.4 EgSpan

This sub-appendix describes the EgSpan software package (`egspan-p.tar.gz`), including its software requirements and file structure. A tutorial is also included to explain the usage of EgSpan.

A.4.1 Software Requirements

The following software and libraries are required for the execution of EgSpan. The version number enclosed by the parentheses denotes the software/library version that has been tested with EgSpan.

- LLVM (2.9): The LLVM compiler framework for the generation of Data-Flow Graph (DFG) files.
- LegUp (3.0): An open-source HLS tool that adapts LLVM as the C-to-Verilog synthesis front-end. Note that LegUp is only used for DFG generation and not for high-level synthesis.
- g++ (4.6.3): The g++ compiler is used to compile EgSpan's C++ code.

A.4.2 File Structure

The `egspan-p.tar.gz` tarball includes the following directories.

- `llvm-modify` provides two modified LLVM source files for generating the DFG files that can be processed by EgSpan.
- `raw2egi` contains a C++ program to convert the DFGs files to the EgSpan input format.
- `adpcm` is a tutorial example described in Section A.4.3.

- `examples` contains more examples besides `adpcm`: *(i)* `aes`: encryption function of advanced encryption standard, *(ii)* `blowfish`: encryption function of data encryption standard, *(iii)* `dfsin`: double-precision floating-point sine function involving addition, multiplication, and division, *(iv)* `gsm`: linear predictive coding analysis of global system for mobile communications, *(v)* `jpeg`: JPEG image decompression, *(vi)* `motion`: motion vector decoding of MPEG-2, and *(vii)* `sha`: transformation function of secure hashing algorithm.

A.4.3 Tutorial

This tutorial use an Adaptive Differential Pulse Code Modulation (`ADPCM`) encoder as an example to explain the usage of EgSpan. Before starting the tutorial, follow the LegUp manual (`legup-3.0/legup-3.0-doc.pdf`) to install LLVM (`clang+llvm-2.9-x86_64-linux.tar.bz2`) and LegUp (`legup-3.0.tar.gz`). The two modified LLVM source files (`egspan-p/llvm-modify/SchedulerDAG.*`) should replace the corresponding original LLVM versions in `legup-3.0/llvm/lib/Target/Verilog/` before starting the tutorial. These modifications can generate DFG files whose vertices (representing LLVM primitive operations[2]) are annotated with their locations in the original source code.

To generate the DFG files for (`ADPCM`), run `make` in the directory `legup-3.0/examples/chstone/adpcm`. The name of the generated DFG files should end with `raw`. Move all the generated DFG files (`*.raw`) to the directory `egspan-p/adpcm/raw/`. Change the work directory to `egspan-p/adpcm/raw/`.

EgSpan Input: EgSpan reads an input text file (`input.egi`) that contains at least one of the following section which represents a DFG:

```
t
v <vertex-index> <vertex-label>
...
e <edge-from> <edge-to> <edge-label>
...
```

where `t` denotes the beginning of a DFG representation, and each line starting with `v` (`e`) represents

[2] Documents of LLVM primitive operations can be found online at `http://llvm.org/releases/2.7/docs/LangRef.html#instref`.

a DFG vertex (edge). The `<vertex-label>` is the ID of the vertex's primitive operation. The `<edge-label>` is currently not used, but it can be adapted to model the edge characteristics if needed. All the indices start at zero, and the labels are unsigned integers.

EgSpan Output: EgSpan outputs three text files:

- `dfg_map` provides a list of the raw DFG file names. Note that in `dfg_map` the ID of the DFG file listed at "Line i" is "$i - 1$". The DFG IDs start at zero.
- `op_map` provides a mapping of primitive operations and their IDs. The primitive-operation IDs also start at zero.
- `output.ego` provides a list of frequent patterns (i.e. subgraphs), each following the format:

  `<pattern>`
  `<id>`*pid*`</id>`
  `<support>`*sup*`</support>`
  `<what>`
  `v <vertex-index> <vertex-label>`
  `...`
  `e <edge-from> <edge-to> <edge-label>`
  `...`
  `</what>`
  `<where>`*gid_list*`</where>`
  `</pattern>`

  where *pid* is the pattern ID, *sup* is the support count of the pattern, and *gid_list* is a list of IDs denoting the DFGs that contain the pattern. The subgraph (pattern) enclosed between `<what>` and `/what>` follows the same DFG format as in the input file `input.egi`.

Under the `adpcm` directory, edit the script `../run.sh` by specifying the value of k for discovering the top-k patterns. Run the edited script `../run.sh`, which will *(i)* convert the raw DFG fills into the input file `input.egi`, *(ii)* execute EgSpan, and *(iii)* generate the three output files. Use the three output files and the raw DFG files to look up the pattern locations in the original C source code.