UNIVERSITÉ DE MONTRÉAL

ACCURACY-GUARANTEED FIXED-POINT OPTIMIZATION IN HARDWARE SYNTHESIS AND PROCESSOR CUSTOMIZATION

SHERVIN VAKILI

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIÆ DOCTOR

(GÉNIE INFORMATIQUE)

AOÛT 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

ACCURACY-GUARANTEED FIXED-POINT OPTIMIZATION

IN HARDWARE SYNTHESIS AND PROCESSOR CUSTOMIZATION

présentée par : VAKILI Shervin

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de:

Mme NICOLESCU Gabriela, Doct., présidente

M. LANGLOIS J.M. Pierre, Ph.D., membre et directeur de recherche

M. BOIS Guy, Ph.D., membre et codirecteur de recherche

M. DAVID Jean Pierre, Ph.D., membre

M. ABDI Samar, Ph.D., membre

Dedicated to

my mother, Sima

and the memory of my father, Fariborz

ACKNOWLEDGEMENT

I would like to express my sincere and greatest gratitude to my supervisors, Dr. Pierre Langlois and Dr. Guy Bois. I am thankful for their constant support and encouragement and for their constructive advice and comments.

Special thanks to Mr. Gary Dare for helping to proof-read this thesis. I would also like to thank the faculty, staff and students of the department who helped me to expand my knowledge and expertise. I would also like to thank them for every bit of help they provided, each in their own way.

My eternal gratitude to my family and friends for their unconditional encouragement and support.

RÉSUMÉ

De nos jours, le calcul avec des nombres fractionnaires est essentiel dans une vaste gamme d'applications de traitement de signal et d'image. Pour le calcul numérique, un nombre fractionnaire peut être représenté à l'aide de l'arithmétique en virgule fixe ou en virgule flottante. L'arithmétique en virgule fixe est largement considérée préférable à celle en virgule flottante pour les architectures matérielles dédiées en raison de sa plus faible complexité d'implémentation. Dans la mise en œuvre du matériel, la largeur de mot attribuée à différents signaux a un impact significatif sur des métriques telles que les ressources (transistors), la vitesse et la consommation d'énergie. L'optimisation de longueur de mot (WLO) en virgule fixe est un domaine de recherche bien connu qui vise à optimiser les chemins de données par l'ajustement des longueurs de mots attribuées aux signaux.

Un nombre en virgule fixe est composé d'une partie entière et d'une partie fractionnaire. Il y a une limite inférieure au nombre de bits alloués à la partie entière, de façon à prévenir les débordements pour chaque signal. Cette limite dépend de la gamme de valeurs que peut prendre le signal. Le nombre de bits de la partie fractionnaire, quant à lui, détermine la taille de l'erreur de précision finie qui est introduite dans les calculs. Il existe un compromis entre la précision et l'efficacité du matériel dans la sélection du nombre de bits de la partie fractionnaire. Le processus d'attribution du nombre de bits de la partie fractionnaire comporte deux procédures importantes: la modélisation de l'erreur de quantification et la sélection de la taille de la partie fractionnaire. Les travaux existants sur la WLO ont porté sur des circuits spécialisés comme plate-forme cible.

Dans cette thèse, nous introduisons de nouvelles méthodologies, techniques et algorithmes pour améliorer l'implémentation de calculs en virgule fixe dans des circuits et processeurs spécialisés. La thèse propose une approche améliorée de modélisation d'erreur, basée sur l'arithmétique affine, qui aborde certains problèmes des méthodes existantes et améliore leur précision.

La thèse introduit également une technique d'accélération et deux algorithmes semi-analytiques pour la sélection de la largeur de la partie fractionnaire pour la conception de circuits spécialisés. Alors que le premier algorithme suit une stratégie de recherche progressive, le second utilise une méthode de recherche en forme d'arbre pour l'optimisation de la largeur fractionnaire. Les algorithmes offrent deux options de compromis entre la complexité de calcul et le coût résultant. Le premier algorithme a une complexité polynomiale et obtient des résultats comparables avec des

approches heuristiques existantes. Le second algorithme a une complexité exponentielle, mais il donne des résultats quasi-optimaux par rapport à une recherche exhaustive.

Cette thèse propose également une méthode pour combiner l'optimisation de la longueur des mots dans un contexte de conception de processeurs configurables. La largeur et la profondeur des blocs de registres et l'architecture des unités fonctionnelles sont les principaux objectifs ciblés par cette optimisation. Un nouvel algorithme d'optimisation a été développé pour trouver la meilleure combinaison de longueurs de mots et d'autres paramètres configurables dans la méthode proposée. Les exigences de précision, définies comme l'erreur pire cas, doivent être respectées par toute solution.

Pour faciliter l'évaluation et la mise en œuvre des solutions retenues, un nouvel environnement de conception de processeur a également été développé. Cet environnement, qui est appelé PolyCuSP, supporte une large gamme de paramètres, y compris ceux qui sont nécessaires pour évaluer les solutions proposées par l'algorithme d'optimisation. L’environnement PolyCuSP soutient l’exploration rapide de l'espace de solution et la capacité de modéliser différents jeux d'instructions pour permettre des comparaisons efficaces.

ABSTRACT

Fixed-point arithmetic is broadly preferred to floating-point in hardware development due to the reduced hardware complexity of fixed-point circuits. In hardware implementation, the bitwidth allocated to the data elements has significant impact on efficiency metrics for the circuits including area usage, speed and power consumption. Fixed-point word-length optimization (WLO) is a well-known research area. It aims to optimize fixed-point computational circuits through the adjustment of the allocated bitwidths of their internal and output signals.

A fixed-point number is composed of an integer part and a fractional part. There is a minimum number of bits for the integer part that guarantees overflow and underflow avoidance in each signal. This value depends on the range of values that the signal may take. The fractional word-length determines the amount of finite-precision error that is introduced in the computations. There is a trade-off between accuracy and hardware cost in fractional word-length selection. The process of allocating the fractional word-length requires two important procedures: finite-precision error modeling and fractional word-length selection. Existing works on WLO have focused on hardwired circuits as the target implementation platform.

In this thesis, we introduce new methodologies, techniques and algorithms to improve the hardware realization of fixed-point computations in hardwired circuits and customizable processors. The thesis proposes an enhanced error modeling approach based on affine arithmetic that addresses some shortcomings of the existing methods and improves their accuracy.

The thesis also introduces an acceleration technique and two semi-analytical fractional bitwidth selection algorithms for WLO in hardwired circuit design. While the first algorithm follows a progressive search strategy, the second one uses a tree-shaped search method for fractional width optimization. The algorithms offer two different time-complexity/cost efficiency trade-off options. The first algorithm has polynomial complexity and achieves comparable results with existing heuristic approaches. The second algorithm has exponential complexity but achieves near-optimal results compared to an exhaustive search.

The thesis further proposes a method to combine word-length optimization with application-specific processor customization. The supported datatype word-length, the size of register-files and the architecture of the functional units are the main target objectives to be optimized. A new

optimization algorithm is developed to find the best combination of word-length and other customizable parameters in the proposed method. Accuracy requirements, defined as the worst-case error bound, are the key consideration that must be met by any solution.

To facilitate evaluation and implementation of the selected solutions, a new processor design environment was developed. This environment, which is called PolyCuSP, supports necessary customization flexibility to realize and evaluate the solutions given by the optimization algorithm. PolyCuSP supports rapid design space exploration and capability to model different instruction-set architectures to enable effective comparisons.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AA	Affine Arithmetic
ADL	Architecture Description Language
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
BER	Bit Error Rate
CI	Custom Instruction
DCT	Discrete Cosine Transform
DFG	Data Flow Graph
FFC	Floating-point to Fixed-point Conversion
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FU	Functional Unit
FWL	Fractional Word-Length
GA	Genetic Algorithm
HDL	Hardware Description Language
HDR	High Dynamic Range
IA	Interval Arithmetic
IIR	Infinite Impulse Response
ILP	Instruction-Level Parallelism or Integer Linear Programming
IR	Intermediate Representation
ISE	Instruction Set Extension
IWL	Integer Word-Length

LISA	Language for Instruction-Set Architecture
MPSoC	Multi-Processor System on Chip
MAC	Multiply and Accumulate
MWL	Multiple Word-Length
PSNR	Peak Signal-Noise Ratio
QAA	Quantized Affine Arithmetic
RTL	Register Transfer Level
SA	Simulated Annealing
SoC	System on Chip
SPREE	Soft Processor Rapid Exploration Environment
SQNR	Signal-to-Quantization-Noise Ratio
TIE	Tensilica Instruction Extension
TM	Tone Mapping
UWL	Uniform Word-Length
UFB	Uniform Fractional Bit-width
VLIW	Very Long Instruction Word
WLO	Word-Length Optimization

CHAPTER 1 INTRODUCTION

1.1 Overview and motivation

Fractional computation is necessary in a vast amount of applications in the DSP and image processing domains. In digital arithmetic, a fractional value can be represented in fixed-point or floating-point. Arithmetic operators and their associated computational complexity are highly different for these two representations. Using the same number of bits, the floating-point representation supports a wider range of values compared to the fixed-point representation. However, in a hardware implementation, the complexity of a fixed-point realization is normally far less than the equivalent floating-point one [1]. As a result, a fixed-point computational circuit is usually more efficient than the corresponding floating-point circuit in terms of area, performance and power consumption. Hence, the fixed-point representation is commonly preferred for hardware implementation, particularly in embedded systems.

In a digital computational circuit, reducing the bitwidth of a signal can result in improvements in efficiency. The quantity of this improvement depends on the hardware components which are related to that signal, since the size of these components can be affected by the bitwidth reduction.

A fixed-point number is composed of integer bits and fractional bits. In a fixed-point-computing circuit, a fixed number of bits are allocated to the integer and fractional parts of each signal. The Integer Word-Length (IWL) of each signal must be wide enough to guarantee overflow/underflow avoidance. The minimum required width of the integer bits depends on the range of values that the signal may take. The Fractional Word-Length (FWL) determines the accuracy of the computations. A wider FWL means less introduced quantization error at the expense of consuming more hardware resources. Finding the appropriate bitwidths to allocate to the integer and fractional parts of the signal is a well-known research problem called word-length optimization (WLO). The objective of a WLO method is to optimize the efficiency of the hardware implementation while meeting the accuracy requirements of the application by adjusting the word-length of the signals.

Fixed-point WLO consists of two main activities:

- IWL optimization, which aims to find the minimum number of bits for the integer part of each signal that guarantees overflow/underflow avoidance. Range analysis is the usual method for IWL optimization.
- FWL optimization, which aims to maximize the efficiency of the design through adjusting the fractional word-lengths of the signals. Accuracy requirements must be met by the selected fractional word-lengths. Finite-precision error modeling and FWL selection algorithm are the main parts of this activity.

The WLO is the subject of numerous works in the literature. The objective of most of these works is to optimize the efficiency of the fixed-point computational circuits by allocating the best combination of word-lengths to the signals in the circuit. The signal word-lengths determine the bitwidth of the arithmetic, control and logic operators in the hardware implementation. In this field, research efforts normally focus on one of the three major components of any WLO approach: finite-precision error modeling, integer word-length selection algorithm and fractional word-length selection algorithm. Almost all of these works have targeted the hardwired circuits as the implementation platform. However, the WLO problem can be extended to other hardware platforms such as microprocessors.

1.2 Problem statement

This thesis introduces new methods to enhance and optimize the hardware implementation of the fixed-point computations in hardwired circuits and microprocessor. WLO for hardware synthesis and microprocessor platforms form a major part of this thesis.

For hardwired circuit design, this thesis introduces new techniques and algorithms to improve existing word-length allocation methods. There are a large number of related works in the literature that aim to improve different parts of the WLO. These works normally compete on efficiency of the circuits that they produce and the execution time that they require. In this work, we introduce a finite-precision error modeling approach that amends and promotes a widely-used Affine-Arithmetic-based approach. The proposed modifications address a significant hazard in the existing method and improve the error modeling accuracy. The latter improves the efficiency of the FWL selection by increasing the error estimation accuracy while the former prevents false

FWL allocation. FWL selection is the other significant problem in WLO that was targeted in many existing works. These works typically proposed heuristic search algorithms to solve this NP-hard problem [2, 3]. In this work, we introduce an acceleration technique to reduce the searching time of the FWL selection algorithms. Moreover, we present two new heuristic algorithms to solve this problem more efficiently.

Microprocessors are the other platforms that are targeted in the contributions of this thesis. Over the last few decades, microprocessors have been the most popular choice to implement computational algorithms. Programmability and high degree of flexibility are the main advantages of microprocessor-based designs. Processor-centric architectures are commonly used in embedded systems, which is the largest production area of digital computers. However, general purpose processors are not normally able to meet the high demand of computationally intensive real-time applications in the multimedia, communication and signal processing domains. At the other extreme, maximum feasible performance is offered by hardwired circuits widely known as ASICs (Application-Specific Integrated Circuits). However, ASICs offer the lowest flexibility in terms of programmability and reusability.

In the processor domain, this research introduces the idea of combining the fixed-point word-length allocation with the processor customization, for the first time. This idea proposes use of word-length determination in fixed-point custom processor design process in order to customize the quantity and the format of the data types supported by the processor architecture. The objective is to improve the implementation efficiency (in terms of hardware cost, performance and offered accuracy) of the processor architecture by customizing the bit-length of the processor datapath and corresponding microarchitecture components. Based on this idea, we introduce a method for accuracy-guaranteed optimization of the processor word-length for fixed-point point applications.

Processors use functional units (FUs) to realize arithmetic functions. FUs are major performance bottlenecks particularly in computation-intensive applications and they typically demand significant hardware area. The complexity of each arithmetic function and the technique and architecture employed to implement them determine the performance and hardware cost of the corre-

sponding FU. Selection of the appropriate architectures for the functional units forms a new optimization problem.

Signal and image processing is selected as the target application domain for this research. Therefore, the developments and the evaluations will be accomplished based on selected applications from this domain.

1.3 Objectives

The main objective of this research is to develop methodologies to enhance the word-length allocation process for hardwired circuit design and to combine fixed-point optimization and processor customization into an integrated design environment. The processor customization environment will generate an optimized application-specific fixed-point processor as well as the corresponding executable code based on the given application.

The following specific objectives are identified for this research plan:

- Developing a custom processor design environment, which provides enough capability to implement and evaluate the ideas of the next objectives such as functional-unit architecture selection and datapath word-length configuration.
- Introducing an enhanced finite-precision error modeling approach that addresses the problems in existing approaches and improves their accuracy.
- Introducing an acceleration technique and two new semi-analytical word-length selection algorithms to improve fixed-point circuit design.
- Proposing an automatic word-length allocation methodology to be integrated into the custom processor design process. This methodology is based on design space exploration and aims to optimize cost-accuracy trade-offs.
- Proposing a method to enhance processor efficiency through functional unit architecture customization. This process is tightly connected to the word-length allocation solution and, therefore, must be integrated in the word-length allocation process to form a comprehensive optimization algorithm.

1.4 Thesis organization

This thesis is divided into 7 chapters. Chapter 2 reviews the important background material and related works that are used in this thesis. Chapter 3 describes a new processor design environment that is used to realize the processor customization method introduced in Chapter 6. Chapter 4 presents an improved finite-precision error modeling approach which is used in subsequent chapters. Chapter 5 introduces new algorithms and techniques for WLO in hardware synthesis. Chapter 6 presents a new method to customize the processor architectures based on fixed-point word-length optimization. Chapter 7 concludes the thesis.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we review the previous work that forms the background material of this thesis. Two topics, which are relevant to our research, were selected for this literature review: application-specific processor customization and fixed-point word-length optimization. First, we review various existing custom processor design methodologies and important related works. Second, we survey significant work in research areas related to fixed-point word-length optimization. This includes a review of the important IWL and FWL optimization methods. In FWL optimization section, we consider finite-precision error-modeling approaches and FWL selection techniques.

2.1 Custom processor design

The complexity of embedded SoC (System-on-Chip) designs has increased enormously in recent years. Programmable processor cores are increasingly used in such systems to shrink the design turnaround time through high-level language programming and high degree of code reusability [4]. However, general purpose processors are not usually able to meet high computational requirements of complex embedded applications in multimedia, communication systems, and signal processing domains. Application-specific processor customization is a well-known approach that has emerged in the past few years to close the gap between programmable processors and hardwired hardware implementation in ASICs (Application-Specific Integrated Circuits) [5-7]. The objective of processor customization is to enhance the efficiency of the processor by specializing specific elements of the processor architecture based on requirements of the target applications.

2.1.1 Customization categories

Processor customization approaches can be divided into two categories: micro-architectural tuning and instruction-set customization. Important existing works in these two categories are reviewed in this section.

The objective of micro-architectural tuning is to find the most efficient trade-offs for datapath elements such as pipeline depth, processor word-length and functional unit implementation. In micro-architectural tuning, there are typically some configurable parameters related to the pro-

cessor architecture, which are influential in efficiency. The objective is to find the combination of parameter values that represent the best trade-off among efficiency factors (i.e., performance, hardware cost, and occasionally power consumption). Henkel [8] divided micro-architectural tuning into two types of configurations: inclusion/exclusion of hardware functional units (e.g., hardware multiplier) and parameterization of components such as cache memories. Major existing works in this area that were reviewed for related parts of this thesis are summarized below.

Yiannacouras *et al.* [9, 10] developed a processor design environment, called SPREE, and used it to explore the impact of microarchitecture tuning on efficiency metrics of the processor. Their exploration covered four micro-architectural parameters: (1) optional hardware multiplication support; (2) choice of shifter implementation; (3) pipeline depth; and (4) cycle latency of multicycle paths. The presented results demonstrate significant impacts of microarchitectural tuning on the efficiency of the processors. The results also confirm the capability of SPREE for rapid design space exploration.

Dimond *et al.* [11] presented a Field Programmable Gate Array (FPGA) implementation of a parameterizable core, named CUSTARD, supporting the following options: different number of hardware threads and types, Custom Instructions (CI), branch delay slot, load delay slot, forwarding, and register file size. CUSTARD offers generation of single and multithread architectures. Customizations must be performed manually in CUSTARD since no automatic search algorithm was integrated in this tool.

Sheldon *et al*. [12] introduced a methodology for fast application specific customization of parameterized FPGA soft cores. In this work, two search approaches were considered for microarchitectural customization. One uses a traditional CAD approach that does an initial characterization using synthesis to create an abstract problem model and then explores the solution space using a knapsack algorithm. The other uses a synthesis-in-the-loop exploration approach. The methodology was evaluated on Xilinx MicroBlaze soft-core processors taking into account four inclusion/exclusion and cache parameterization configurations. The results demonstrate that the introduced approaches can generate customized processors that are 2× faster than the base soft-core, reaching within 4% of the optimal. The execution time of these approaches is 1.5 hours on average, compared to over 11 hours for the exhaustive search.

Padmanabhan *et al.* [13] formulated microarchitecture customization as a multi-objective nonlinear optimization problem to find Pareto-optimal configuration for the LEON processor. This approach is linear in the number of reconfigurable parameters, with an assumption of parameter independence. This assumption highly simplifies the optimization problem. The results show that the approach was able to achieve a performance gain within 0.02% difference from the exhaustive solution and with 1% reduction in LUTs (chip resource cost).

Saghir *et al.* [14] presented a new development tool to design and evaluate VLIW (Very Long Instruction Word) architecture with a set of customizable microarchitectural parameters. They compared the impacts of parameters on the efficiency factors of the design. The results show that this method can achieve significant gains in efficiency over XILINX MicroBlaze using a combination of processor customization and instruction-level parallelism. Obviously, this method takes advantage of the inherent superiority of the VLIW architectures in performance and, therefore, it cannot be directly compared with most of the related works that only focus on single-instruction architectures.

Automatic microarchitectural tuning is the other well-studied topic in literature. The related works commonly propose integrated design flows that consist of a search algorithm to explore the design space for the optimal microarchitectural configuration and a processor generation process to realize the best-found solution. Diverse search algorithms were proposed for this purpose in the literature. Hebert *et al.* [15] and Kuulusa *et al.* [16] adopted exhaustive search for the exploration of the architecture parameter space while Fitcher *et al.* [17] proposed heuristic approaches for this problem. Exhaustive search can usually yield the best possible results at the expense of significantly slower execution. The methodology presented in [15] involves an analysis of the resources of the processing core used by the target application. Then, a series of optimizations based on the analysis results are performed on an optimizable model of the processor core. The proposed exploration algorithm in [17] uses Pareto-dominance tests to prune non-optimal parts of the design space. A VLIW template design is used as the base architecture and obtained Pareto-optimal points for a number of DSP-like benchmark programs are presented. Table 2.1 summarizes some of the major works mentioned in this section.

Table 2.1 Comparing previous works related to micro-architecture tuning

	Supported architectures	**Parameters**	**Exploration method**
Yiannacouras *et al.* [9, 10]	Simple pipeline (based on MIPS I)	• Hardware vs. software multiplication • Shifter implementation • Pipeline depth, organization, and forwarding.	Manual
Dimond *et al.* [11]	Hardware multi-thread	• Multi-threading support: number of threads, threading type • Forwarding and interlock architecture: branch delay slot, load delay slot, forwarding: enable/disable • Register file: number of registers, number ports	Manual
Sheldon *et al.* [12]	Simple pipeline (Microblaze)	• Inclusion/exclusion of hardware FUs for: multiplier, barrel shifter, divider, floating-point unit • Data cache configuration	Automatic • Knapsack • Impacted ordered trees
Padmanabhan *et al.* [13]	Simple pipeline (LEON2)	• Data cache configuration • Instruction cache configuration • Integer Unit: multiplier, divider, register window size, fast jump	Automatic Integer Linear Programming
Saghir *et al.* [14]	VLIW	• Multiplier • Data forwarding paths • RAM block depth and word-length	Manual

Instruction-set customization, which is also known as Application-Specific Instruction-set Processor (ASIP), aims to adapt the processor's instruction set to a given application to achieve improvement according to a chosen metric. Automatic instruction-set customization is defined as a process to generate CIs from an application in order to improve intended efficiency metric(s). This activity can be categorized into two main approaches [5].

The first approach offers complete customization in which the whole instruction-set is selected based on application requirements [18]. The second approach is partial customization, also known as instruction-set extension (ISE), which involves adding a limited number of CIs to a pre-existing instruction-set architecture [19, 20]. The ISE process generally starts with the source code of the target application written in a high-level programming language such as C. The process typically consists of three significant steps [4]: (1) application profiling and characterization; (2) automated ISE identification; (3) ISE realization.

Application profiling is used to identify the computational hotspot areas of the target application. The objective of the ISE identification step is to find combinations of operations in hotspot areas, which can be integrated into a single instruction aiming for improvement in performance. Finally, the ISE realization step involves techniques to synthesize and add the selected custom instructions into the processor architecture.

Most of the research in this field has focused on automatic ISE identification. The introduced methods are commonly based on data flow graph (DFG) analysis. The custom instruction selection process can be formulated as a convex sub-graph identification problem [21]. Each convex sub-graph of a DFG is a potential CI. The selection process typically takes into account hardware cost, architectural constraints and achievable improvements of the candidate CIs. Examples of such constraints include limited bandwidth of custom functional units and general purpose register file [4].

Various methodologies and algorithms have been proposed and employed for DFG-based custom instruction identification. Kastner *et al.* [22] combined template matching and generation to identify sub-graphs based on recurring patterns. A novel algorithm was introduced in this work that profiles a dataflow graph and iteratively clusters the nodes based on a method, called edge contraction, to create the templates. The paper investigated how to target the algorithm toward the

novel problem of instruction generation for hybrid reconfigurable systems. In particular, this work targeted the Strategically Programmable System, which embeds complex computational units such as ALUs, IP blocks, and so on into a configurable fabric.

Clark *et al.* [23] and Sun *et al.* [24] proposed heuristic approaches to solve this problem. The methodology presented by Sun *et al.* [24], employs a two-stage process, wherein a limited number of promising instruction candidates are first short-listed using efficient selection criteria, and then evaluated in more detail through cycle-accurate instruction set simulation and synthesis of the corresponding hardware. Dynamic pruning techniques were also exploited to eliminate inferior parts of the design space from consideration. In the methodology proposed by Pozzi *et al*. [25], a binary tree search approach is employed to discover all potential CIs in a DFG first. Then the candidates who do not meet the predefined constraints (e.g., hardware cost and register file access bandwidth) are discarded to speed up the search process.

2.1.2 Existing methodologies

There are two basic trends in application-specific custom processor design: partial customization of a configurable processor and designing from scratch. Tensilica Xtensa [26], MetaCore [27] and SC Build [28] are some examples of partially customizable processor environments in which the main body of the processor is fixed, while a limited number of elements or components are left customizable. Xtensa is known as a configurable and extensible RISC processor core. Configuration options include the number and width of registers, memories, inclusion/exclusion of hardware units for some operations, etc. New instructions can be described using the Tensilica Instruction Extension (TIE) language and added to the baseline core. A complete software toolkit as well as synthesizable code can be generated automatically for Xtensa processors [26]. Moreover, automatic design space exploration for TIE-based instruction extension is offered by the provided development tools.

Architecture Description Languages (ADLs) such as PEAS III [18], LISA [29] and EXPRESSION [30] offer designing from scratch, which provides higher flexibility by allowing the designers to define their own ISA and datapath at the expense of more design effort. An ADL is normally accompanied by a corresponding tool-chain that allows automatic generation of the software toolkit (including compiler, assembler, simulator, and debugger) and Register Transfer

Level (RTL) code generator for the ADL processor model. The objectives of different ADLs may vary and their modeling complexity and generable outputs are not necessarily similar.

Mishra *et al.* [30] have classified existing ADLs according to two aspects: content and objective. ADLs can be classified into four categories based on the contents: structural, behavioral, mixed and partial. In structural ADLs, architecture of components and their connectivity must be explicitly defined. This category needs lowest abstraction level in modeling and consequently offers the highest flexibility. MIMALO and UDL/I are two well-known structural ADLs. In contrast, behavioral ADLs explicitly specify the instruction semantics and ignore detailed hardware structures. This means that the micro-architecture of the processors is not modeled in this approach. nML is one of the behavioral ADLs that captures instruction-set architecture and corresponding functional description in a hierarchal scheme [31]. The required structural description is limited to the information used by the instruction-set architecture (ISA). For example, memory and register units should be defined since they are visible to the instruction-set. The timing model is not supported for computations in nML. Sophisticated Instruction Level Parallelism (ILP) techniques such as those presented in superscalar processors cannot be modeled by nML.

Mixed ADLs capture both architectural details and behavioral function description of the processors. EXPRESSION is a mixed-level ADL primarily designed to generate software toolkits from the processor/memory description to enable compiler-in-loop design-space exploration [32]. Structural modeling in EXPRESSION includes three subsections: pipeline and data transfer path description, component specification and memory subsystem. Similarly, the behavioral model is composed of three subsections: operation specification, instruction description, and operation mapping. The components can be multi-cycle or pipelined units for storage elements, ports, and connections for which the timing behavior can also be specified. Automatic RTL generation from an EXPRESSION model is not supported by the original tool chain. However, a restricted template-based RTL generation method was proposed in [33]. LISA (Language for Instruction-Set Architecture) is the other mixed-level ADL that offers complete tool chain along with optimal RTL generation for design space exploration. LISA needs explicit modeling of the controlling process in the designs. This feature enables LISA to model complex control paradigms at the expense of a more complicated design process. However, LISA cannot model out-of-order-execution found in superscalar processors [34].

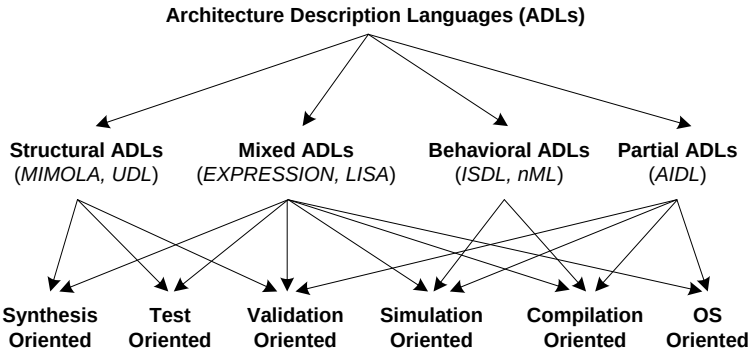


Figure 2.1 Taxonomy of ADLs

ADLs can also be classified into six categories based on their objectives: synthesis oriented, test oriented, validation oriented, compilation oriented, simulation oriented and Operating System (OS) oriented. Figure 2.1 demonstrates the taxonomy of ADLs based on the presented classification. This figure is extracted from [30].

As a result of the increasing complexity of SoC designs, in recent years, synthesis-based design space exploration has faced great challenges in search speed. An emerging trend to address this issue is to move toward simulation-based exploration in higher levels of abstraction. This trend is quickly becoming popular, particularly in Multiprocessor SoC (MPSoC) designs. Figure 2.2 compares the simulation speed of various abstraction levels of modeling.

Yiannacouras *et al.* [10] presented SPREE (Soft Processor Rapid Exploration Environment) that facilitates the design and exploration process using a high-level format for ISA and datapath description. In this text-based description, each instruction is defined as a directed graph of basic components provided as a library of available micro-operations, e.g., register files and instruction fetch units. Although using these encapsulated components significantly simplifies the design process, it also limits the describable architectures. For example, the internal structure of major components such as register files and fetch units are predefined and making any modification to these components entails direct HDL (Hardware Description Language) programming. Moreover, datapath and ISA should be designed separately from scratch and the designer is in charge of the consistency between datapath and ISA models. Yiannacouras *et al.* [9] employed the SPREE environment to explore the impacts of microarchitecture tuning on efficiency metrics of a soft-processor and compared the results with the Altera Nios II processor [35].

Nurvitadhi [28] introduced a transactional specification framework (T-space) that allows description of a pipelined processor as a state machine with a set of transitions. In this approach, the designer views the datapath as executing one transaction at a time, just like single-cycle designs. The T-space description is converted into the pipeline model using a specific synthesizer, called T-piper. The pipeline model allows concurrent execution of multiple overlapped transactions in different pipeline stages. The objective of this work is to facilitate pipelined processor design by automating the major parts of the design process such as pipeline parallelization and hazard prevention.

Dimond *et al.* [11] presented the FPGA implementation of a parameterizable core, named CUSTARD, supporting the following configurable parameters: number of hardware threads, CI, branch delay slot, load delay slot, forwarding paths and register file size.

Saghir *et al.* [14] presented a development tool to design and evaluate VLIW architectures with a set of customizable micro-architectural parameters. They compared the impacts of parameters on the efficiency factors of the design.

2.2 Fixed-point word-length optimization

For convenience in design and verification, most signal and image processing applications are initially developed in floating-point arithmetic. The applications are converted into fixed-point arithmetic for hardware implementation to achieve more efficient circuits. Fixed-point WLO is used to perform this conversion, automatically.

A WLO method is composed of the IWL and FWL allocation processes that aim to find the optimal values of IWL and FWL for each signal in the design. The efficiency of a word-length allocation method is measured by its execution speed and the efficiency of the resulting circuit. For complex designs, up to 50% of the design time may be needed for word-length allocation [1]. IWL and FWL allocation are naturally different problems while their complexity is categorized as NP-hard [2]. There are a significant number of previous works that propose various methods and algorithms to solve these problems. In the following subsections, we review some of the important existing works for each problem.

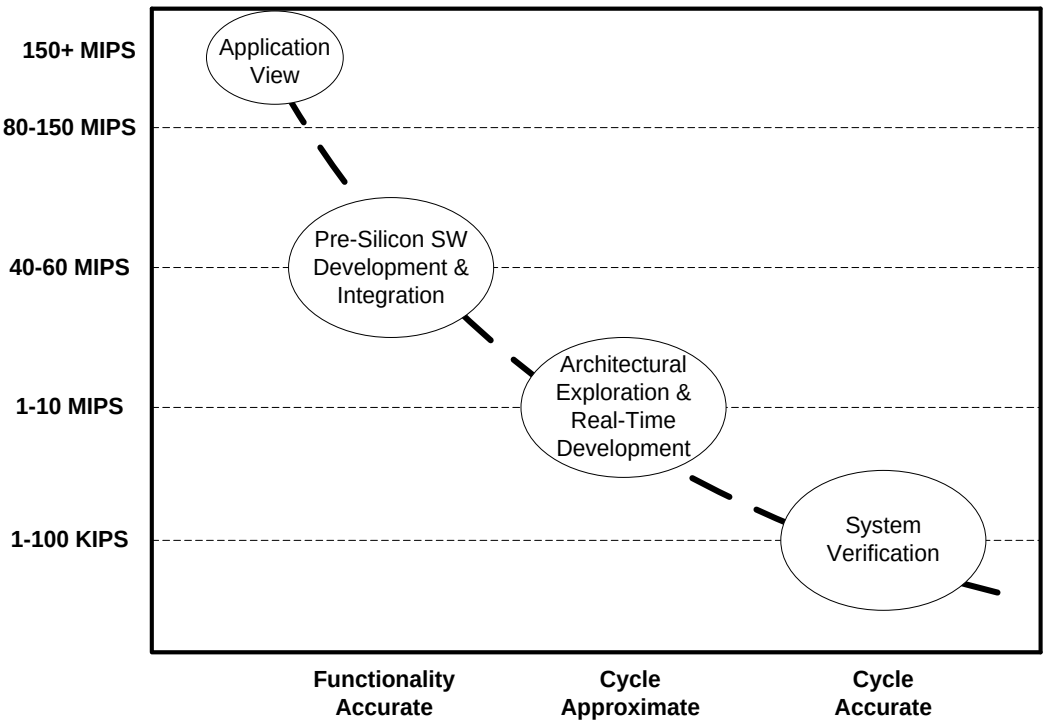


Figure 2.2 Simulation speed vs. abstraction levels.

2.2.1 IWL allocation

The allocated IWL to each signal of the circuit must be wide enough to guarantee overflow/underflow prevention during computation. There is an upper-bound in the range of values that each signal may take. The minimum required IWL is determined from this range of values. Hence, the main activity in IWL allocation is to find the range of values for each signal. A vast number of approaches have been introduced in literature to perform this activity, which is also known as range analysis. Range analysis approaches can be divided into two major categories: simulation-based approaches and analytical approaches.

Simulation-based approaches feed various input data to the algorithm to find out the variation range of each signal [36, 37]. These methods use statistical properties of the signals, such as the mean and the variance, and the maximum and minimum peak values obtained during simulations. Analytical approaches try to formulate the range analysis problem first and then employ appro-

priate methods to solve it [38]. Although the results of analytical approaches are commonly more conservative, their higher speed makes them more popular for complex designs [2, 39].

The methods based on Interval Arithmetic (IA), as well as those based on forward and backward propagation [40] usually overestimate scaling, while approaches based on multi-intervals [41] and Affine Arithmetic (AA) [38] may achieve better results by reducing overestimation. The methods based on AA and on the transfer function [42] are suitable for feedback systems. The methods based on the transfer function, however, are not able to handle nonlinear systems. The AA-based methods are modified in non-linear systems to reduce the computational complexity.

In addition to the fixed-point hardware design, the IWL allocation is also used for conversion of software code from floating-point to fixed-point. This activity is sometimes called Floating-point to Fixed-point Conversion (FFC) in the literature. The FFC commonly aims to convert a floating-point sequential code to a corresponding fixed-point one in order to be eventually executed in an embedded processor. Since fixed-point operators are the same as integer ones, integer arithmetic and data types along with appropriate scaling operators (typically realized by a shift operation) are adequate to implement a fixed-point application. Hence, a conventional integer processor is able to execute fixed-point applications produced by the FFC process.

Since the word-length is usually constant in conventional processors, the IWL allocation is sufficient for fixed point transformation. In literature, this scheme is also known as Uniform Word Length (UWL). When the IWL has been determined, the remaining bits are simply allocated to the fractional part. The output of this process is an integer code in which scaling operators are used to correct the values of the signals that have been converted from a floating-point representation. Figure 2.3 demonstrates this modification process on a simple Finite Impulse Response (FIR) filter. This example is highly extracted from [43]. In this figure, IWL_i represents the binary-point position in signal *i* from the left side. The dynamic range of the signals is identified by limited precision values. The original flow graph (Figure 2.3.a) illustrates that determining different binary-point positions for the signals (based on their dynamic range) may lead to unaligned operands for a single operation. In the IWL_u and the IWL_{acc} signals, the point is placed after the first and after the fourth bit, respectively. Thus, a scaling operation must be introduced between the multiplication and the addition to align the binary-point position before the addition.

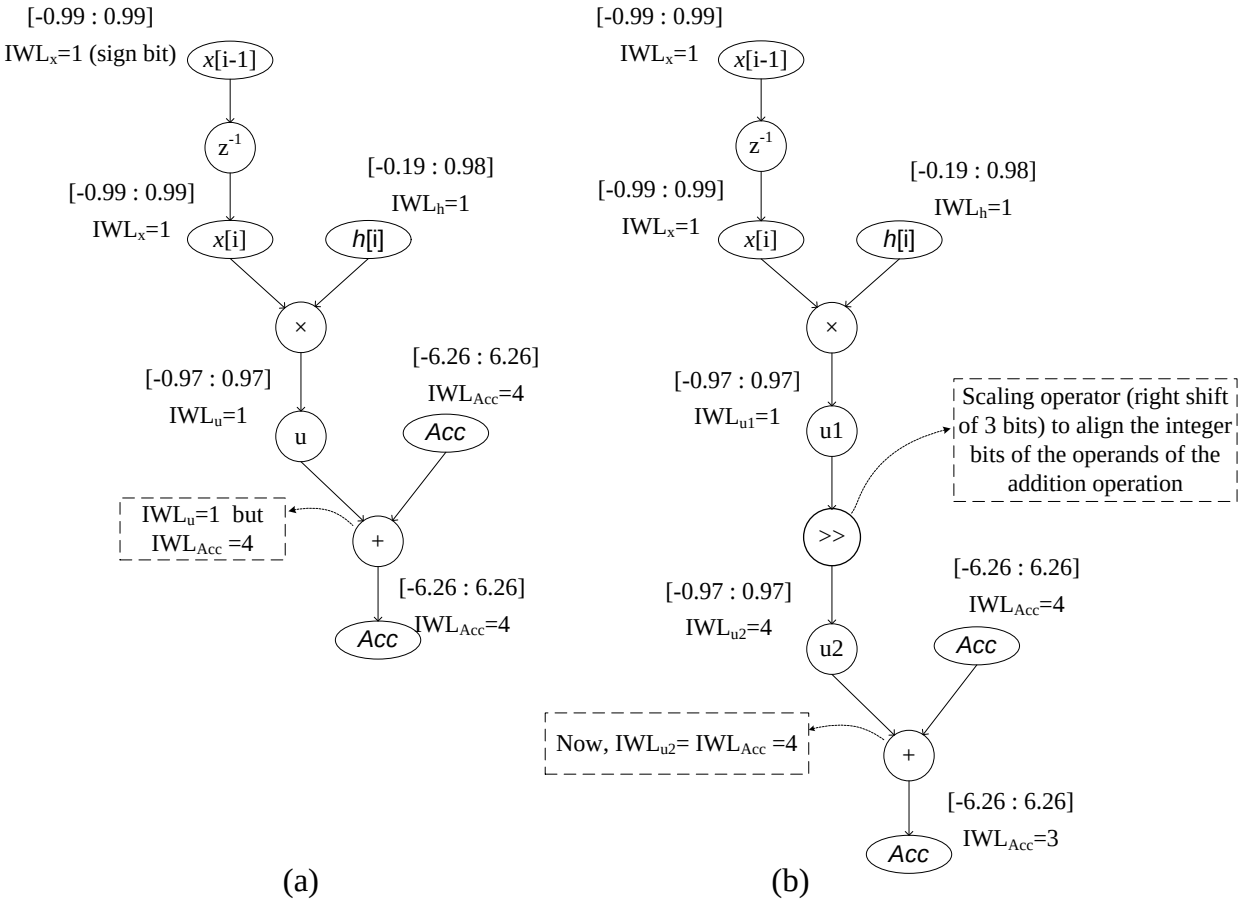


Figure 2.3 Using scaling operations for fixed-point alignment in FIR filter example.

Figure 2.3.b represents the DFG after the insertion of the scaling operation. *u*, *u1* and *u2* are intermediate signals. This example illustrates why scaling factors are necessary in fixed-point software development using the example of an FIR filter.

Earlier works in this category mostly proposed code conversion methodologies for particular DSP processors. In [44], a methodology which generates fixed-point code for the TMS320C25/50 DSP is proposed. The output code cannot be used in other architectures. Automatic tools to transform floating-point C source code into an ANSI code with integer data types are proposed in [45] and [46]. These methods use control flow graph representations for their analysis. Moreover, they apply an optimization process to minimize the number of required scaling operators in the output code.

2.2.2 FWL allocation

FWL allocation is the other important part of WLO. The FWL of each signal determines the amount of finite-precision error that it introduces in computations. This error can propagate through the next stages of the operations and eventually show up at the output as the finite-precision inaccuracy. Allocating wider FWLs results in more accurate computations at the expense of more hardware resources and higher latency.

FWL allocation aims to optimize the efficiency of hardwired hardware designs (on ASIC or FPGA) by adjusting the bitwidth of the fractional part of the signals. The accuracy requirement is the main consideration that must be satisfied by the FWL allocation solution. Since each operation in hardwired circuits can have a dedicated hardware unit, each intermediate signal can have a different word-length. This scheme is known as Multiple Word Length (MWL) in the literature. It is widely accepted that MWL is potentially able to lead to much more efficient implementations in terms of hardware cost compared to UWL in hardwired circuit design [47].

In both MWL and UWL approaches, the tolerable output inaccuracy of the design is normally measured in terms of the worst-case output error bound or Signal-to-Quantization-Noise Ratio (SQNR) for DSP systems. However, it can be also measured through other application specific metrics, such as the Bit Error Rate (BER) or Peak Signal-Noise Ratio (PSNR) in wireless systems and image/video processing designs, respectively. The quantization error must be estimated for every point in the search space that is examined during FWL selection [48].

The FWL allocation process normally consists of two important parts: finite precision error modeling and FWL selection algorithm. Each of these parts is the subject of a large number of researches.

Finite-precision error modeling is used to estimate the amount of quantization noise that is introduced at the outputs from the FWL of the input and intermediate signals. Analyzing the word-length effects on the precision of the computation is sometimes called precision analysis in literature. IA is again a widely-used method for finite-precision error modeling [49]. One drawback of IA is that it ignores the correlation among signals [50, 51]. AA is a preferable approach that addresses the correlation problem by taking into account the interdependency of the signals. In AA, each signal is represented as a linear combination of certain primitive variables, which stand in

for sources of uncertainty. Fang *et al.* [51], Lee *et al.* [38] and Osborne *et al.* [52] introduced word-length optimization methods based on AA. In Chapter 4 of this thesis, we introduce an enhanced AA-based error modeling approach. This method is used for the error estimation throughout the thesis

FWL selection methods can be categorized into optimal and heuristic methods. The optimal methods mainly employ exhaustive search or Integer Linear Programming (ILP). Both approaches are highly computational intensive. Heuristic methods aim to reach quasi-optimal result in a more reasonable amount of time. Some of the proposed heuristic methods are based on gradient-descent. These methods follow one of the following strategies to approach the optimal result: (1) starting from an infeasible point (due to unacceptable precision) and improving the accuracy [53]; and (2) starting from a feasible point and reducing the cost by degrading the precision [54]. The advantage of these methods is their relatively fast convergence. Their main drawback is their potential to fall in local optima. Simulated Annealing (SA) is another algorithm used to solve the word length selection problem [38]. SA-based approaches use stochastic properties of the search space to find the solution. Since SA algorithms use random point selection and exploration, they tend to be able to jump out of local optima.

Although the main objective of the word-length optimization process is to reduce the implementation cost (i.e. area, latency, power, etc.), many of the proposed methods do not take into account hardware costs directly [55]. However, some of the recent methods embody cost estimation techniques and consider the estimation result as the fitness value of the candidate solutions. These latter methods typically obtain results closer to the optimum. Most of the previous works considered the hardware area as the cost value [42]. However, latency, power consumption, or a combination of these metrics is taken into account in some researches [56]. Some of the important related works in error modeling and FWL selection are reviewed below.

López *et al.* [57] introduced a new methodology to represent statistical parameters of the quantization noise using AA. In this method, each quantizer of the realization is first modeled by an independent affine form. The constant value and the uncertainty factors are calculated as functions of the mean and variance of the noise source. Afterwards, the noise models are propagated using an AA-based simulation. The results show that, although this approach is more accurate

than analytical AA-based error modeling methods, it is still far slower in terms of computational time.

Caffarena *et al.* [58] proposed an AA-based method to estimate SQNR. This method combines the analytical AA calculations and simulation-based noise model parameterizations. The experimental results show a significant speed-up compared to simulation-based methods, at the expense of a negligible estimation error.

In another related work, López *et al.* [59] presented a non-linear adaptation of AA, called Quantized Affine Arithmetic (QAA), that offers tighter interval estimation compared to the traditional AA. In this method, only the uncertainty factors which are associated with the input signals appear in the affine representation of any signal of a design. In other words, QAA prevents introduction of new uncertainty factors in the propagation process. This significantly reduces the complexity of the affine expressions. However, the QAA fails to provide guaranteed worst-case error bounds as in the traditional interval-based computations.

Kinsman *et al.* [60] used Satisfiability-Modulo Theories to refine the range results given by IA and AA. The main drawback of this method is that its runtime grows rapidly with application complexity.

Zhang *et al.* [61] employed Extreme Value Theory for both range and precision analysis. They used a lightweight simulation to study the characteristics of extreme conditions. Although this approach is significantly faster than fully simulation-based approaches, the reported results demonstrate that it is still far slower than analytical methods particularly in large designs.

Boland *et al.* [62] recently introduced a polynomial algebraic approach using Handelman representations. When calculating bounds, this approach takes into account dependencies within a polynomial representing the range of a signal. Although this approach has shown promising results, it faces significant limitations in processing non-polynomial functions and scalability.

Cong *et al.* [50] made an extensive comparison of three significant static precision analysis methods using a set of experiments. The methods studied include AA, general interval arithmetic (GIA) and automatic differentiation. Cantin *et al.* [63] compared some pre-existing word length optimization algorithms through experimental evaluation with twelve DSP applications.

Le Gal *et al.* [64], Constantinides *et al.* [65] and Menard *et al.* combined the word length optimization and high-level synthesis (HLS) problems. These works propose new HLS methodologies which take care of data word length in scheduling, allocation, and binding processes aiming at optimizing the hardware implementation.

Menard *et al.* [66] introduced a grouping algorithm to optimize the resource sharing paradigm for the operations. This process is followed by a WLO algorithm that optimizes the word length of each signal group. The WLO algorithm is composed of a greedy and a Tabu search procedures.

Nguyen *et al.* [67] also proposed a word length selection algorithm based on Tabu search. Contrary to the widely-used greedy search algorithms that are mono-directional, this method allows bidirectional movement in the solution space. To demonstrate applicability, the chapter provides experimental comparisons with three related works.

Fiore *et al.* [68] derived closed-form expressions for efficient word length allocation. They showed that these expressions can be effectively calculated in hardware. This allows the realization of adaptive word lengths that change in real-time as a function of the data. This idea can be particularly beneficial in specific applications such as adaptive filters.

Integer linear programming is a general optimization method that is believed to give optimal results for FWL selection problem. Due to the huge complexity of the ILP solving process, this method is not practical for large designs. However, it has been used to generate reference results for comparison [47, 69]. Custom heuristic methods [38, 52, 62] are the most widely used FWL selection algorithms in existing works.

Some researches focused on word length optimization in specific applications. Pang *et al.* [70] presented a technique for real-valued polynomial representation, such as Taylor series. This technique relies on arithmetic transforms and a branch-and-bound search algorithm for word-length allocation. Lee *et al.* [71] proposed an optimization methodology for piecewise polynomial approximation of arithmetic functions. Nguyen *et al.* [72] designed an optimized fixed-point WCDMA receiver using a combination of static and dynamic techniques.

The MWL word-length optimization can also be effectively used in High-Level Synthesis (HLS). HLS is a digital design trend, which has been widely studied since its introduction in the early

1990s. The main objective of HLS is to offer methodologies to convert a high-level description of an application to low-level RTL, automatically. HLS methods are mainly composed of three phases: scheduling, resource allocation and resource binding. Scheduling identifies timing period for the computation of each operation in the algorithm. Resource allocation involves the selection and integration of hardware components to realize operations. These resources include functional units, storage components, intercommunication circuits, and controlling logic. Resource binding is the process of assigning a resource to each operation.

As mentioned before, the MWL approach is efficient for hardware synthesis of fixed-point designs. It means that in such realizations, each variable in the algorithm may have a different word-length and precision. Consequently, operators of the same type may have different word-lengths. The resource allocation process is highly dependent on availability of a library of components on the target hardware platform. The timing properties, hardware area and power consumption of the components are normally provided in the library. These characteristics are used by the resource allocation process to identify the appropriate set of required components. Obviously, considering different word-lengths for the operators significantly increases the size of the component library. This leads to a vast growth in the complexity of the resource allocation process.

Early methodologies used the UWL scheme to simplify the library and the exploration process. However, MWL has been considered in more recent works. Contrary to the traditional approaches (i.e., UWL-based ones) which only consider the type of the resources in the allocation process, MWL-based methods take into account both the type and word-length of the components. This new model is based on the fact that an operation can be executed on a component only if its input word-lengths are equal or smaller than the component word-lengths. More than 50% area reductions are reported with the MWL approach. There are a few research works which combined HLS with MWL optimization [47, 65, 73-75]. Actual combination of these two processes provides a more comprehensive design space exploration.

Wadekar *et al.* [73] published one of the earliest works in this field. In their methodology, resources are fully shared among the operations of the same type. The word length selection looks for the lower bound of area cost, considering the cost estimation of the results. This cost estima-

tion is accomplished by a simple model taking into account provided information about component area of as well as their word-lengths. The latency of all FUs is assumed to be one cycle for simplification.

The work introduced in [74] proposes a methodology in which the word-length assignment (WLA) and HLS are carried out iteratively. The word-length optimization activity is preceded by HLS in each iteration. HLS integrates functional units which can be grouped together and WLA tries to reduce hardware cost by minimizing the word-length. This process is repeated until the improvements finish. The paper suffers from insufficient experimental results and comparisons with previous methods.

Constantinides *et al.* [65] proposed a new methodology in which a heuristic is employed to address the scheduling problem with incomplete word-length information. This methodology actually combines the resource binding and word-length selection processes. The heuristic follows a primary word-length selection step and aims to refine the word-length information regarding the scheduling and resource binding decisions. The provided results represent up to 46% of area reduction even for modest problem sizes.

Caffarena *et al.* [75] proposed a new framework that combines word-length optimization and FPGA-based synthesis. This method considers the embedded and logic-based multipliers in the resources binding step. Moreover, variable latency resource models are used in this work. A simulated annealing-based approach for the combined scheduling, resource allocation and binding tasks is presented. Compared to previous approaches, area improvements of up to 60% are reported.

Caffarena *et al.* [47] employed mixed integer linear programming to formulate the combined problem. Storage devices and control logic are not considered and FUs are assumed to have 1-cycle latency in order to simplify the problem. Therefore, this work does not meet real-world design criteria, completely. However, it is worthwhile to consider as initial research in this subject.

Table 2.2 summarizes the previous works on combined WLO and HLS problem. There are still many opportunities for new research on this subject. However, WLO has not been considered in custom processor designs so far.

Table 2.2 Summary of combined WLO and HLS approaches.

Approach	Tasks	Optimizations	Comments
Wadekar and Parker [73]	1. WLS* (considering hardware cost estimation) 2. HLS	Heuristic	1-cycle FUs
Herve *et al.* [74]	Loop: 1. Grouping the FUs 2. HLS 3. WLS	Heuristic	Var. FU latencies
Constantinides *et al.* [65]	Primary WLS Loop: HLS+WLS	Heuristic	Var. FU latencies
Caffarena *et al.* [75]	1. WLS 2. HLS	Heuristic: Simulated Annealing	Var. FU latencies
Caffarena *et al.* [47]	WLS+HLS	MILP**	1-cycle FUs

* WLS: Word-length Selection **Mixed Integer Linear Programming

2.3 Conclusion

In this chapter, we reviewed important related works that form the background knowledge used in this thesis. Some of the contributions of this thesis are built on the ideas introduced in these works. Furthermore, some of these works are used for comparison in several parts of the thesis. First, we reviewed important existing methodologies and environments for custom processor design. Then, we reviewed significant research on different analyses required in fixed-point WLO. We saw that the final objective, of most existing works, in this area, is to generate an optimal hardwired circuit for fixed-point computation. A smaller number of works focused on methods to convert floating-point software code to fixed-point, efficiently. Using WLO for processor customization, which is one of the new ideas proposed in this thesis, has not been considered in any previous work.

CHAPTER 3 POLYCUSP PROCESSOR DESIGN ENVIRONMENT

In this chapter, we present the Polytechnique Customized Soft Processor (PolyCuSP) design environment, which is a new processor design environment that is used for fast and easy custom processor generation. This thesis also introduces a new processor customization method for the fixed-point applications. This customization method is presented in Chapter 5. The basic goal for designing PolyCuSP was to have an environment that supports the required flexibility to realize new customizations proposed in this thesis and facilitates design space exploration in a large design area. The large number of customizable elements in PolyCuSP enables the generation of different processor architectures, which helps us to have more accurate comparisons with the related works. This chapter focuses on the general characteristics and the design flow of the PolyCuSP environment.

The contents of this chapter are largely extracted from our paper "Customised soft processor design: a compromise between architecture description languages and parameterisable processors," published in *IET Computers & Digital Techniques* in 2013 [76].

3.1 Introduction

Employing soft processors is increasingly becoming popular in FPGA-based embedded system design, as they offer rapid design process and high flexibility [4, 7]. However, general purpose processors are not usually able to meet the high computational requirements of complex embedded applications, particularly in the multimedia, communications and signal processing domains.

Application-specific processor customization is a well-known approach to close the gap between programmable processors and dedicated hardware implementation, while keeping post-fabrication flexibility [6]. In recent years, a large body of research has focused on different aspects of employing processor customization in the soft processor domain. Processor customization can be divided into micro-architectural tuning (or datapath customization) and instruction-set customization. The objective of microarchitectural tuning is to find the most efficient trade-offs for datapath elements such as pipeline depth, processor word-length and functional unit implementation. Instruction-set customization aims to adapt the processor’s instruction set to a given application for achieving improvement according to a chosen metric. The application-specific

instruction-set customization is widely known as ASIP design, in literature. Design space exploration is normally an essential part of a processor customization methodology. Rapid development process and high performance efficiency measurement are highly demanded in design space exploration [9].

Traditional RTL programming has proven to be inefficient for custom processor design mainly due to its high development cost. ADLs aim to reduce the design complexity by offering higher-level processor-specific description [30]. The main drawback of ADLs is their verbose format that can significantly slow down the development process and, consequently, the design space exploration. Moreover, the RTL generation capability in existing ADLs has shown poor quality in terms of efficiency in hardware synthesis [9]. A third option consists of using parameterizable and extensible processors. Such processors usually have a fixed core with a limited number of tunable microarchitectural parameters, and an Instruction-Set Architecture (ISA) with some extension capability. This approach constrains the designer to a narrow range of possibilities, which results in a smaller search space and faster design process. In general, it does not support specific customization techniques such as ISA subsetting. Altera Nios II [35] and Xtensa [26] are two well-known examples of parameterizable and extensible processors.

In this chapter, we explore a new area in the custom processor design space which lies between ADLs and extensible processors such as Xtensa. We introduce a new customized soft processor design environment, called PolyCuSP. The PolyCuSP environment combines the flexibility of the ADLs with the easy customizability of parameterizable processors. It allows ISA description from scratch for highest flexibility, while limiting datapath description to a predefined set of tunable parameters. This compromise follows from the observation that implementing usual micro-architectural customizations does not normally require high flexibility and is one of the main sources of design complexity in ADLs. In PolyCuSP, all microarchitectural elements are either directly defined by the tunable parameters or inferred from the ISA description.

The main difference between PolyCuSP and extensible processors like Xtensa is that PolyCuSP supports customization of the core processor datapath, which is normally fixed in extensible processors [77]. For example, the Xtensa core processor comes with 80 fixed instructions. Major datapath elements such as the processor word-length (bit-width of the registers and signals), in-

struction encoding formats and register-file size and access ports are fixed in the Xtensa core architecture. Other customizable processors such as Nios II present similar characteristics. However in PolyCuSP, the core processor datapath is mostly defined by the designer and hence, highly flexible. This flexibility facilitates evaluation of significant processor design techniques. For example, supporting multi-port memories enables PolyCuSP to realize related techniques such as the one proposed by Panda *et al.* [78], in a rapid way.

3.2 Processor description method

The PolyCuSP environment offers automatic generation of synthesizable RT-level VHDL code, assembler, and MATLAB simulation model from the given processor description. An overview of the development process is shown in Figure 3.1.

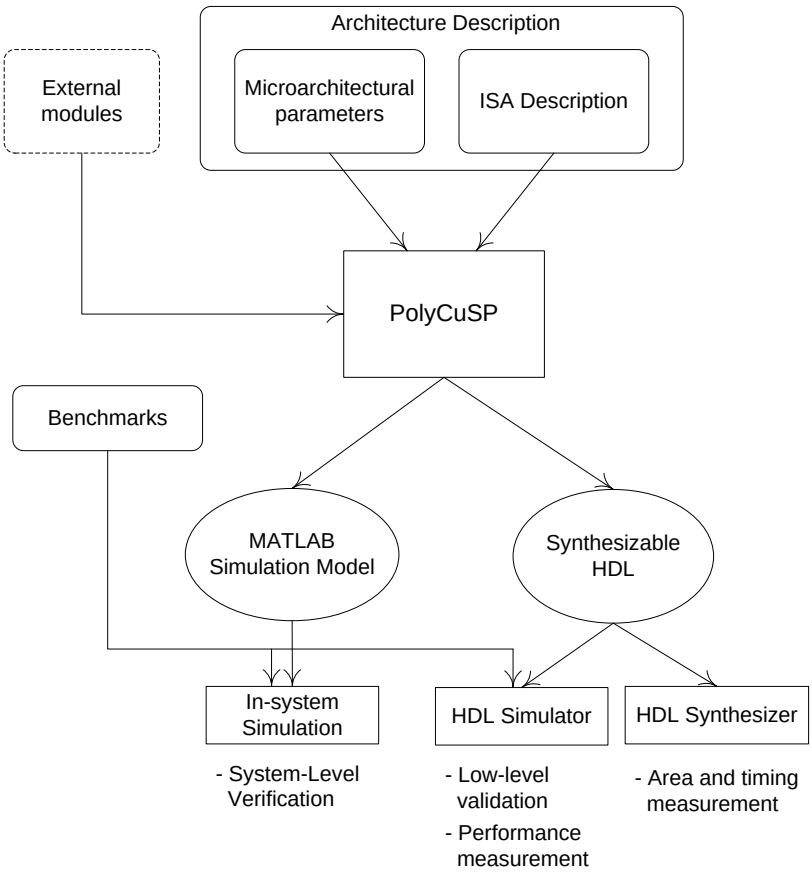


Figure 3.1 Overview of the proposed development process

The processor description in PolyCuSP is partially parametric and partially programming-based to allow design flexibility and rapid design exploration capabilities. The numeric parameters are used to define the size and bit-length of the components, select among limited number of options, and include or exclude specific elements. The datapath bit width, type of data memory (direct memory or cache-based) and the inclusion of forwarding paths are some examples of numerically defined elements in the processor description model.

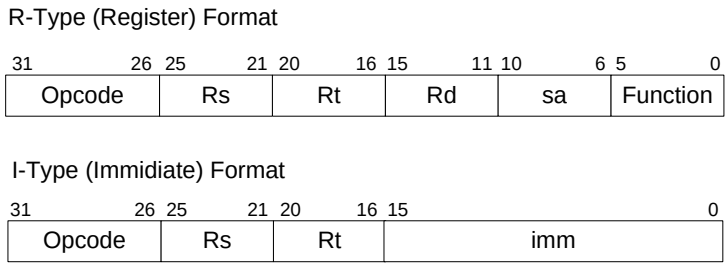
Keeping the datapath description consistent with the intended ISA is a very complex task for most existing ADLs and related environments such as SPREE [10]. In pipelined architectures, designers are normally responsible for balancing stage latency by appropriate distribution of the circuits among the stages. Improper distribution may lead to long critical paths and consequently lower clock frequencies. In PolyCuSP, the datapath shape and component distribution is fixed while the set of microarchitectural elements is customizable through numerically-tunable parameters and ISA description. This idea is supported by the fact that almost all previous works on microarchitectural design space exploration have limited their search space to a few elements. So, extensive flexibility provided by text-based datapath descriptions is not necessary for such limited explorations. Hence, compared to ADLs, the proposed method offers extremely low cost microarchitectural exploration while covering most of the significant parameters such as instruction encoding format. On the other hand, for the instruction-set architecture, the PolyCuSP environment offers full control to the designer. This allows the environment to realize different ISAs and well-known customization techniques such as instruction-set extension and subsetting.

The instruction-set description has a hierarchal structure and is stored in an XML file. The description is composed of the following elements:

1. Definition of general and special purpose processor registers and register-file(s). The identifier, depth and bit-width values must be indicated for each register.
2. Definition of instruction encoding formats.
3. Definition of instructions including assembly syntax, functional description and decoding data (the data by which the instructions can be distinguished from each other, e.g., opcode value in many ISAs).

Encoding formats are defined in two steps. First, the encoding fields and their corresponding bit-length are defined within the *Encoding* XML element. Then, a new encoding format can be defined by putting the appropriate set of fields together as an attribute within the *Encode_format* element. The order of the fields in the format definition must be the same as their order in the binary instruction word. Each encoding format has an identifier (ID) to enable easy connection to the instructions.

Figure 3.2 presents the definition of two major encoding formats (R-type and I-type formats) in MIPS ISA [79]. This convenient modeling approach allows the description of a wide variety of encoding formats including variable-length ones.




```
<Encoding>
    <field  name="Opcode"    bits="6"   />
    <field  name="Function"  bits="6"   />
    <field  name="imm"       bits="16"  />
    <field  name="Rs"        bits="5"   />
    <field  name="Rt"        bits="5"   />
    <field  name="Rd"        bits="5"   />
    <field  name="sa"        bits="5"   />
</Encoding>

<Encode_format>
    <!-- Instruction Format Definition -->
    <!--R-Type-->
    <format  ID=     "3registers"
             format= "Opcode Rs Rt Rd sa Ext"
    />
    <!--I-Type-->
    <format  ID=     "2registers_imm"
             format= "Opcode Rs Rt imm"
    />
</Encode_format>
```

Figure 3.2 Description of two major encoding formats of MIPS ISA

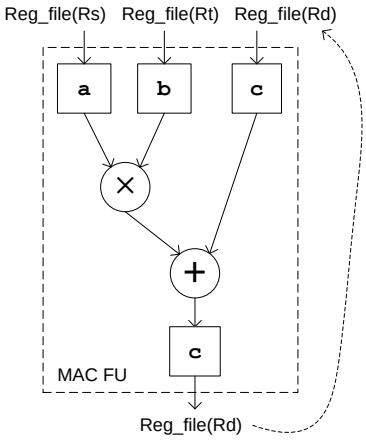
The functionality of ordinary functions can be described with code written in a simplified form of the C language. Special techniques are employed to detect and resolve sequential assignment conflicts in the hardware realization. In this approach, all variables assigned more than once are converted to multiple renamed signals in the final RTL code. Hence, sequential assignments to a single variable or port are supported in function descriptions.

More complex functions, such as multi-cycle and pipelined ones, can be modeled directly in HDL. The HDL modules, which normally reside in an external library of components, can be instantiated in the function description section of the instruction definition.

All temporary variables must be declared before being used inside the code. Two new data types including *bit* and *bitvec* are offered to facilitate the declaration of arbitrary-length variables. All identifiers declared for the registers and encoding fields are meaningful (and thus reserved words) in function description code. As an example, Figure 3.3 demonstrates the description of a MAC (Multiply and Accumulate) instruction.

One of the features of the PolyCuSP environment is its capability to automatically extract required interconnection signals (inputs and outputs of the components and their connections) from the function description of the instruction-set. The number of input and output ports of the register-file(s) and data memory is also identified automatically. For example, in a single-register-file processor, if all instructions have one or two operands, then a dual read-port register-file unit will be generated in the output RTL code and all interconnection signals (including pipeline buffers, input ports of the `EXE` unit, etc.) will be configured accordingly. Now, if a three-operand instruction such as MAC (Figure 3.3) is added to the ISA, the PolyCuSP environment will replace the register file(s) and all related interconnections to support triple concurrent operand reads. This feature significantly simplifies the design process by exempting the designers from explicit declaration of a large number of elements.

External modules can also be instantiated in function description code. This feature allows designers to develop complex CIs and function evaluation algorithms in low-level RTL. In such cases, direct RTL coding normally leads to more efficient designs. External modules can be single- or multi-cycle. A specific handshaking mechanism is defined for control and data exchange purposes.




```
<instruction     name=     "MAC"
                 format=   "3registers"
                 syntax=   "MAC Rd, Rs, Rt"
                 doc=      "Multiply and Accumulate operation"
                 Function= "bitvec[32] temp;
                            temp=reg_file(Rs)×reg_file(Rt);
                            reg_file(Rd)=reg_file(Rd)+temp;" >

                 <field   name="Opcode" value="001000" />
</instruction>
```

Figure 3.3 Description of Multiply and Accumulate (MAC) instruction.

For efficient execution of multi-cycle instructions, a smart interlocking mechanism has been implemented that checks data dependency between the instructions. When a multi-cycle instruction is being executed in a functional unit, the pipeline can continue to work as long as there is no data or resource dependency. Since the contents of external modules are unknown to the environment, a simulation model of these modules cannot be generated automatically and must be provided. The designers can also use encapsulated IP-cores as external modules in their designs.

3.3 Tunable parameters

This section summarizes tunable parameters currently supported in the PolyCuSP environment. The extent of these elements determines the size of the search space in design space exploration. A larger search space may lead to coverage of more meaningful design trade-offs at the cost of a longer design time. Table 3.1 lists significant tunable parameters currently supported by PolyCuSP. These elements can be classified into microarchitectural parameters and the elements determined by the ISA description.

Table 3.1 List of configurable elements in PolyCuSP

Category	Element	Options
Microarchitectural parameters	Pipeline depth	3, 4, 5
	Processor word length	Unlimited
	Forwarding paths	Enable/Disable
	Pipeline interlock control	Enable/Disable
	Branch delay slot	Enable/Disable
	Program memory size	8K-16M Words
	Data memory size	8K-16M Words
ISA Description	Number and functionality of the instructions	-
	Encoding formats	-
	Registers (number, size and bit-length)	Unlimited

Pipeline depth is a customizable parameter which indicates the number of pipeline stages. This parameter is realized by enabling or disabling pipeline buffers. Experimental results reported by Yiannacouras *et al.* [9] demonstrate the inefficiency of two- and seven-stage pipelines while each of the other examined intermediate options (including three, four and five stages) outperforms the others in some applications. Based on a simple pipeline with MIPS I ISA, we have implemented a similar architecture using PolyCuSP to obtain the results presented in this work. Hence, we have only considered the three most promising options in this work. Figure 3.4 illustrates the lay-out of these three options. This assumption is dependent to the target ISA and major datapath features; if either of them changes, then other pipeline depths should also be considered in the exploration process.

Processor word-length is the parameter that determines the size of the addressable words in data memory. The selected value for this option is automatically assigned to the reserved *Wordlength* macro which can be used as a number in the ISA description file. *Forwarding paths* and *pipeline interlocking* are two well-known techniques for data hazard avoidance. Two parameters are de-fined to allow designers to employ either of these techniques.

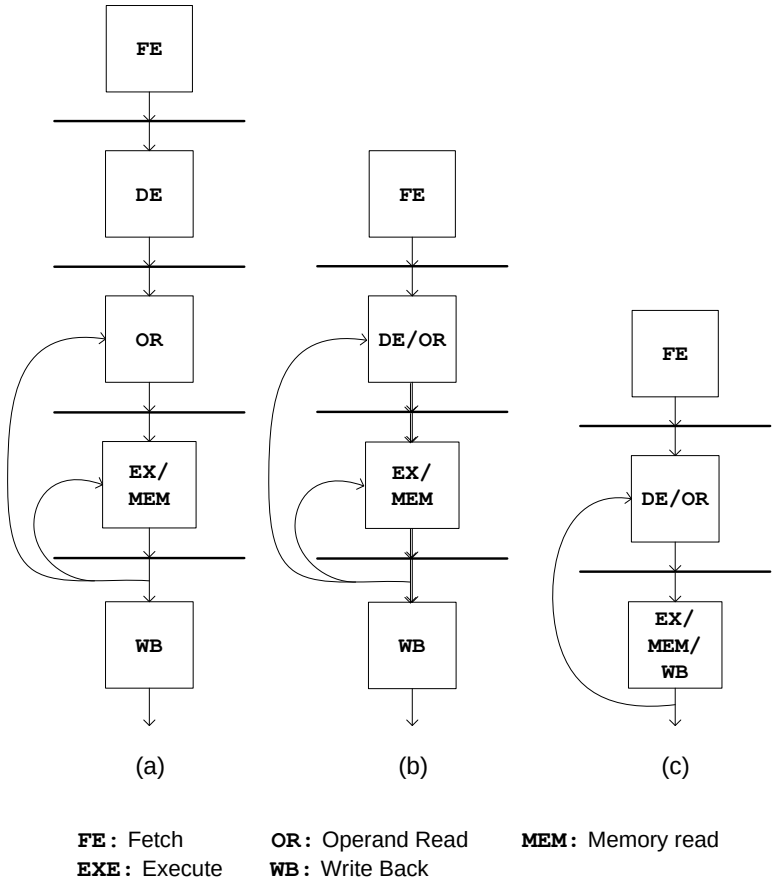


Figure 3.4 Pipeline layout in (a) 5-stage, (b) 4-stage, (c) 3-stage configurations.

When both parameters are disabled, the programmer is responsible for controlling the data hazard in application code. The program and data memory sizes are the other important microarchitectural parameters. They can be set to any power of two value between 2^{13} to 2^{24} words. Cache memories are not considered in the present version of the PolyCuSP environment.

3.4 Processor development process

The development process is composed of the design and verification phases. An extensive toolset was designed to facilitate these two phases. External CAD tools are also necessary for industry-standard HDL simulation and synthesis.

The development environment and the adopted verification method are briefly described in the following subsections.

3.4.1 Development environment

The development environment was implemented in MATLAB. The main reason of selecting MATLAB is its extensive facilities for developing exhaustive and heuristic search algorithms. Also, the main target application domain in this chapter is real-time embedded applications such as DSP and image/video processing. MATLAB provides a comprehensive library of functions and toolboxes which facilitate interfacing, monitoring and verification for such applications. Figure 3.5 illustrates the environment's design flow. Various components of this process are described as follows:

- The first step in a new design is to develop the ISA description and to define the microarchitectural parameters.
- The ISA description and the parameters are then fed to the environment where they are analyzed by the internal engine to extract all necessary information to produce the processor. This information is defined either in the list of parameters or inferred from the ISA description using specific procedures. The encoding formats that determine the decode stage circuits and the number of input and output ports of the register-file(s) are examples of elements that are obtained from the ISA description. Intermediate signals and pipeline buffers that support inter-stage data movements is another example of such elements. The obtained information in this step is passed to the HDL and simulation model generator procedures. The assembler and disassembler are also generated according to the defined assembly syntax of the instructions and the encoding format.
- The HDL generator creates the synthesizable HDL from the captured processor description. The output code can be directly simulated and synthesized by external tools.
- A stand-alone MATLAB simulation model can also be generated for the described processor. This cycle-accurate model greatly facilitates verification and in-system simulation, particularly when the application is initially developed in high-level MATLAB and intended to be implemented on embedded hardware. This is a widely used process in DSP and image/video processing designs. This feature enables performance and correctness evaluation during step-by-step migration from high-level algorithm to implementation.

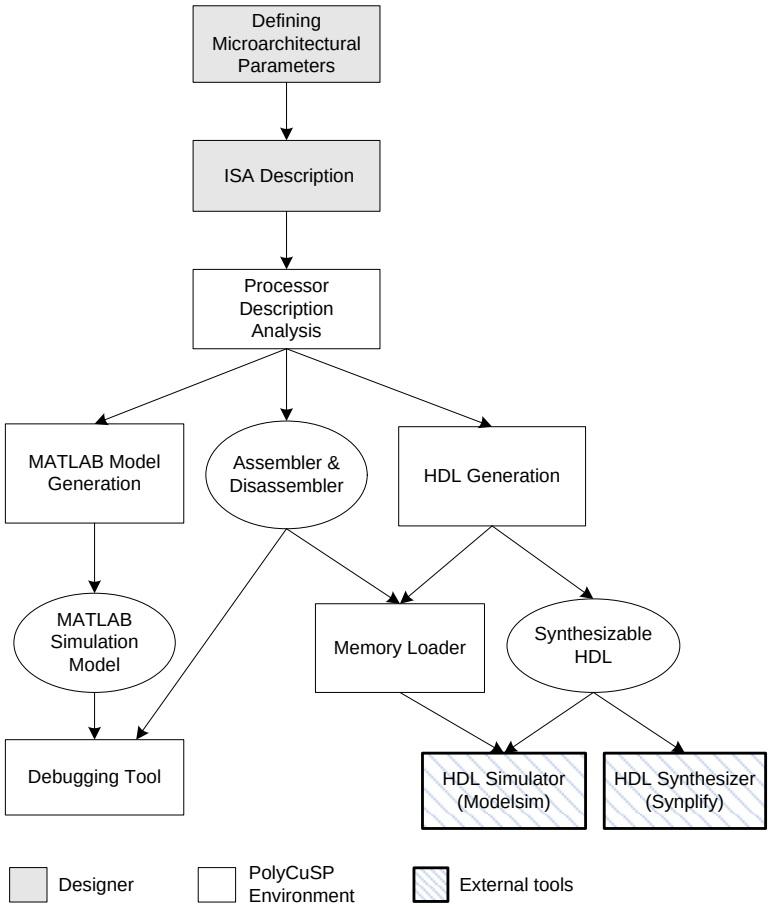


Figure 3.5 PolyCuSP design flow diagram.

- The application loader unit creates program and data memory content files from an input file in assembly or binary file. These files can be loaded into the corresponding memories during RTL and MATLAB simulations. Automatic high-level compiler generation (e.g., C compiler) for custom ISAs has not been supported in current version of PolyCuSP. However, adding this capability using retargetable compilers would be a highly useful future work.
- The development environment also provides a debugging tool to facilitate tracking and validation of application development in target processor. This tool exploits the MATLAB simulation model as the simulation engine.

3.4.2 Processor verification

The generated processors can be verified by examining functional correctness of the simulation and RTL models in a debugging tool and the external HDL simulator, respectively. In the experiments for this chapter, we employed Modelsim to observe RT-level tracing of the data and control signals. All instructions of the input ISA are examined one-by-one and pipeline controlling circuits such as data hazard avoidance techniques are verified by appropriate instruction sequences in the test applications. Other test cases must be developed manually in the current version of PolyCuSP. As a future work, an existing automatic test case generation method can be integrated into PolyCuSP to reduce the verification time and enhance the reliability of the designs.

When modeling a standard ISA, the synthesis results are expected to be comparable with existing implementations. This fact provides another useful verification approach particularly to validate functionality of microarchitectural configurations.

3.5 Experimental results

This section evaluates and compares the PolyCuSP environment in terms of development effort and efficiency of the soft processor it produces. It compares the design space and the design complexity of PolyCuSP with those of Nios II, SPREE and the LISA ADL.

3.5.1 Experimental set-up

The framework generates synthesizable VHDL for the processors. This code is evaluated using standard simulation and synthesis tools to quantify performance and hardware cost results. In this chapter, we have used Synplify 9.0 and Xilinx ISE 13.2 for synthesis and Modelsim for simulations. To enable meaningful comparison with previous works, we have synthesized the designs to Altera Stratix EP1S40F780C5 and Xilinx XC5VLX30 FPGAs. We developed three well-known image and video processing applications for the experiments: Sobel edge-detection operator, 3×3 convolution and Reinhard's tone mapping algorithm. The first two are used in general evaluations and trade-off explorations, while the latter serves as a case study. All applications were coded in assembly language. Despite different assembly syntax in other processors used for comparison, we tried to have the same code structure to ensure that the comparisons are fair.

3.5.2 Comparing with Nios II

We compared the processor generated by our environment with the commercial Altera Nios II soft processor. Since Yiannacouras *et al.* [10] has also compared the SPREE generated processors with Nios II, this allows a comparison between the PolyCuSP environment and SPREE. However, this comparison is not exact due to the different experimental applications used to measure their performance. The Sobel algorithm was adopted for performance measurements in this section.

Three unparameterized variations of Nios II were evaluated in this chapter: Nios II/e, a small six-cycle unpipelined architecture; Nios II/s, a five-stage pipeline with hardware multiplier; and Nios II/f, a large six-stage pipeline architecture with single-cycle multiplier and shifter, and dynamic branch prediction.

As mentioned in Section 4, we implemented the MIPS I ISA (except unaligned load and store operations) which is very similar to the Nios instruction-set. We set similar parameters to identical values to ensure that the comparisons were as fair as possible. For instance, the size of the data and instruction memories in all examined processors is identical and equal to 64 KB.

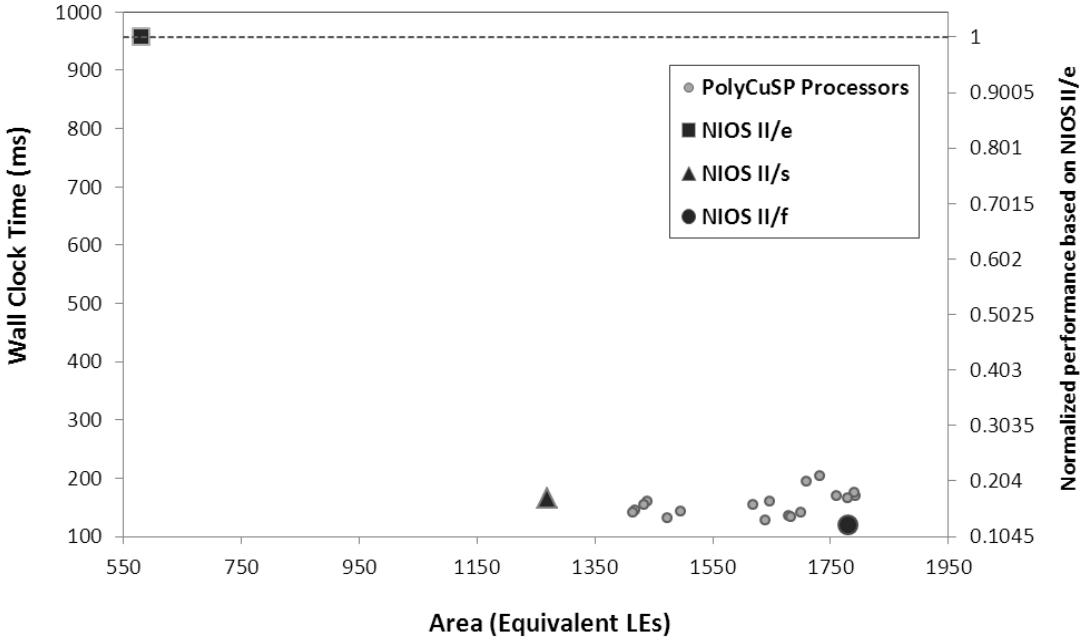
The Sobel application is small enough to reside in instruction cache memories used in Nios II/s and Nios II/f. As a result, the instructions are presumably moved to the on-chip instruction cache only once (for computation of the first pixel) while the same computation is repeated for all pixels of the input image. Hence, memory access time has negligible impacts on reported performance of Nios II/s and Nios II/f and our processors. However, since Nios II/e does not support cache memories, its performance results are subject to longer memory access time impacts. For the present experiments with PolyCuSP, the applications reside in on-chip memory to have a consistent comparison with SPREE and Nios II. Two 64-KB of RAM are used for separate instruction and data memories in both Nios II and PolyCuSP processors. The memories have not been included in area measurements. Memory hierarchy exploration, which requires larger benchmark applications, is out of scope of this thesis. Optional units such as JTAG interface and floating-point unit have not been taken into account in Nios II measurements.

Figure 3.6.a compares processors generated by our tool and Nios II variations in performance-area space. Differences among our processors are caused by changing micro-architectural param-

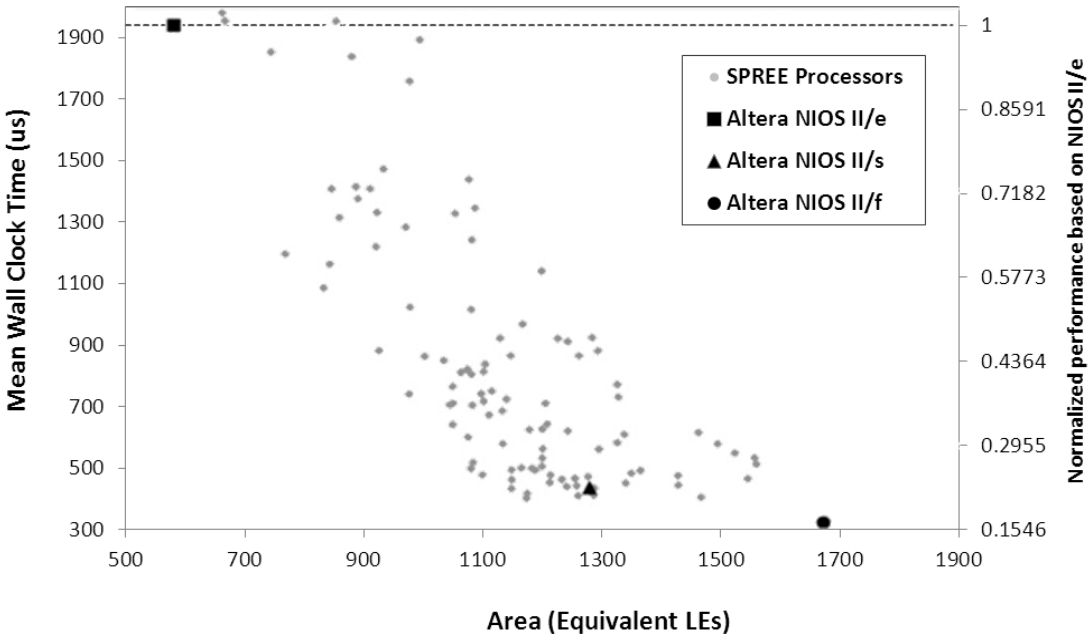
eters while the ISA is the same in all cases. The performance is represented by the wall clock time needed to compute Sobel on a 512×512 image. The most effective factor on the performance is the maximum clock frequency. While Nios II/s and Nios II/f achieve 128 and 136 MHz, respectively, our generated processors reach at most 115 MHz. The result also show a reduction of approximately 30 MHz in maximum frequency migrating from a 4-stage to a 3-stage (keeping other parameters) the same. The IPC (instructions per clock) of our processors in the best case is up to 52% higher than Nios II/s. Since the shift operation is widely used in Sobel algorithm, the single-cycle LUT-based barrel shifter employed in our processors is one of the effective factors on the IPC results.

Figure 3.6.b, which is borrowed from Yiannacouras' paper [10], presents a comparison of SPREE generated processors and Nios II variations. The performance is measured by calculating the average wall clock time of 20 embedded benchmarks. The processor of interest is an 80 MHz processor which is 9% smaller and 11% faster than Nios II/s. The right vertical axes in Figure 3.6.a and Figure 3.6.b illustrate normalized performance values based on Nios II/e as a common reference design. These axes facilitate the comparison between the design space of SPREE with PolyCuSP. Comparing the two diagrams demonstrates that the PolyCuSP design space is more oriented toward higher performance solutions. Furthermore, a precise look at the diagrams show that PolyCuSP offers slightly more efficient solutions near the high performance design space in some cases.

SPREE requires a description of both datapath and instruction-set. Each instruction is described in terms of data dependence graphs of basic operations coded in C++. The datapath description is composed of the list of the components (such as memories and arithmetic functions) and the interconnections between their ports. Individual components are described by RTL coding. In the open access MIPS I SPREE models [80], an average of 510 and 180 lines of C++ programs are needed for instruction and datapath descriptions, respectively. The RTL description of the components was not taken into account in this complexity assessment since they can be commonly used in other processor developments. The similar instruction-set was described using approximately 430 lines of XML code in the PolyCuSP environment, including functional description of the instructions. The microarchitectural parameters can be stored in fewer than 50 XML lines.



***(a)** PolyCuSP design space and Nios II variations running Sobel operator*



***(b)** SPREE design space and Nios II variations [10]*

Figure 3.6 Comparing Nios II with (a) PolyCuSP environment (b) SPREE environment.

Although a difference in benchmark applications prohibits a direct comparison with SPREE, the fairly constant position of the Nios II variations in performance-area space helps to make a valid assessment of the efficiency of the two environments. The results show that microarchitectural configuration in PolyCuSP covers a relatively smaller area of the design space compared to SPREE. This area is more oriented towards higher performance and more costly solutions. The main reason is that we gave a higher priority to performance in PolyCuSP development. Based on this priority, we eliminated the slower solutions from consideration. Moreover, the diversity of the solutions is more limited in PolyCuSP due to supporting a smaller number of configuration options. On the other hand, the efficiency of the PolyCuSP designs is comparable with the highest performance solutions of SPREE and NIOS II.

3.5.3 Processor customization techniques

Two customization techniques were evaluated in the PolyCuSP environment. In the first step, we explored microarchitectural trade-offs of the generated processors for two target algorithms: a Sobel edge detector and a 3×3 vector convolution. Pipeline depth, branch delay slot and data hazard avoidance techniques were covered in these experiments. Figure 3.7 illustrates the performance of possible trade-offs in terms of average instruction throughput of the two evaluated algorithms. The results show that four-stage pipelines generally yield the highest performance mainly due to the lower clock frequency in three-stage pipelines, and higher data dependency and branch penalties in five-stage pipelines. Pipeline interlocking circuits impose significant delays on the *Operand Read* stage. Hence, in the presence of the interlocking technique, the processor may suffer a reduction in maximum achievable clock frequency. The best performance is offered by four-stage pipeline with forwarding, and branch delay slot that yields more than 66 MIPS average instruction throughput.

In the second step, we examined application-specific ISA customization techniques for the Sobel algorithm. First, we applied an instruction subsetting technique through which all unused instructions were pruned from a default MIPS I instruction-set. Then, the pruned ISA was extended by two new custom instructions selected after a manual analysis of the Sobel algorithm (Figure 3.8). This operator is composed of two steps: a convolution of two 3×3 constant vectors and input image (or video frame); and post-processing of the convolution results to calculate the output image.

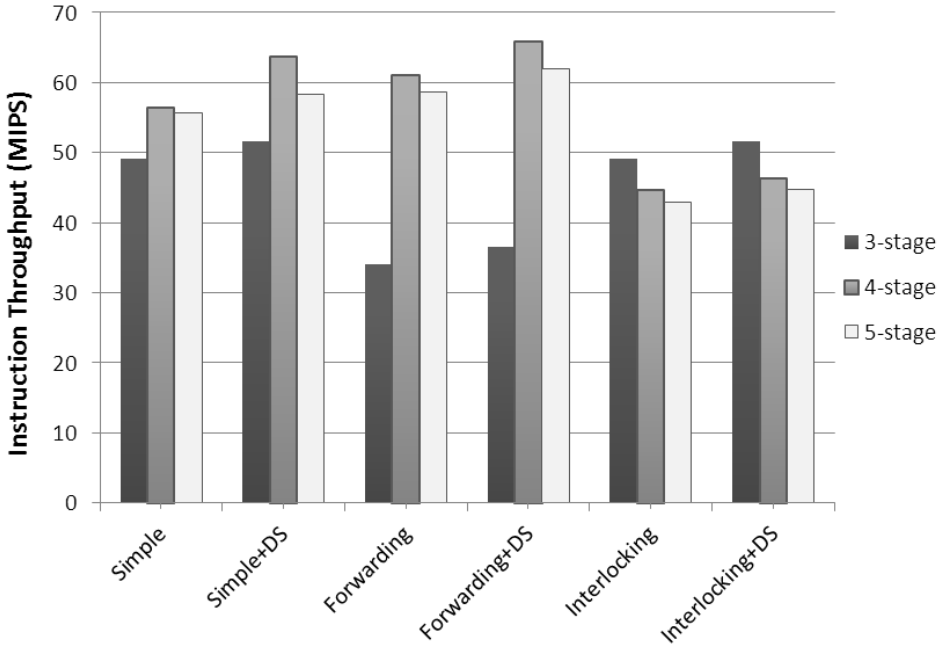


Figure 3.7 Impact of microarchitectural parameters on performance

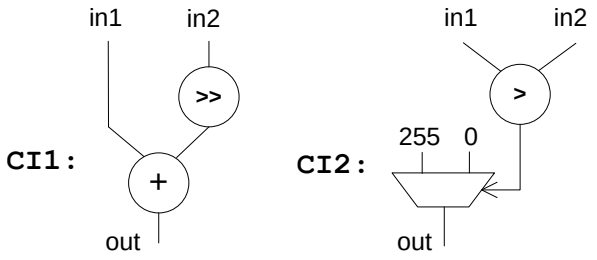


Figure 3.8 Selected CIs for the Sobel algorithm.

Figure 3.9 presents performance-area trade-offs after deploying ISA customization techniques. This diagram demonstrates significant improvement in both performance and area. The results show that the described ISA customization techniques lead to an average improvement of 19% and 35% in performance and hardware area, respectively.

In summary, the performance-to-cost ratio was improved by an average of 44% and 27% through microarchitectural tuning and ISA customization, respectively, targeting the Sobel algorithm.

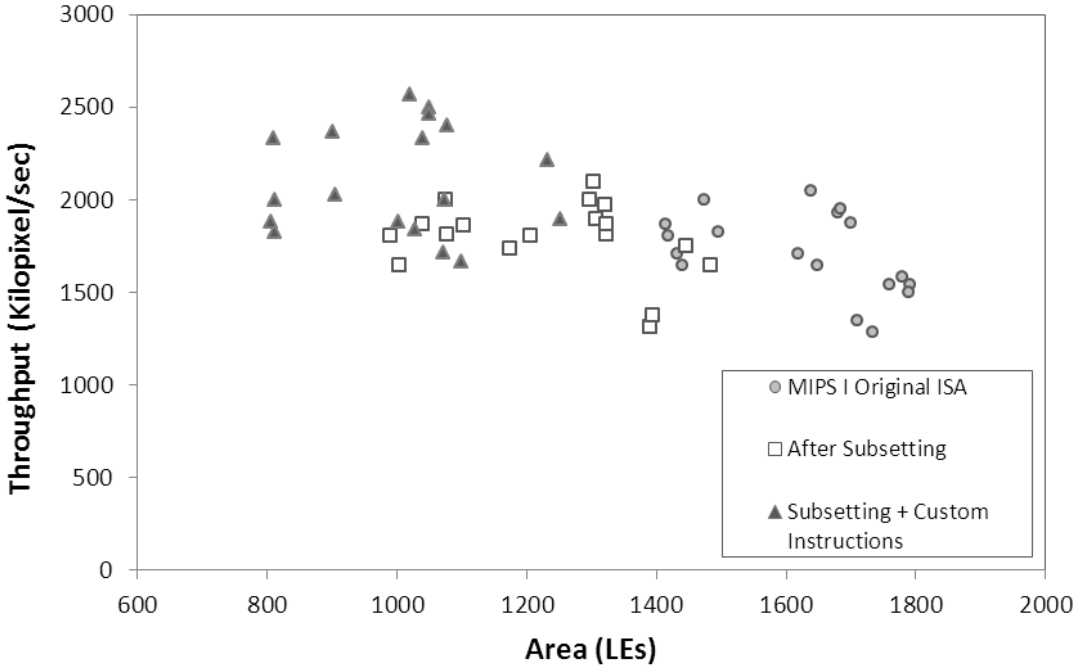


Figure 3.9 Performance-area trade-offs after ISA customization.

3.5.4 Case study of HDR tone mapping

This section evaluates the proposed environment through the implementation of a High Dynamic Range (HDR) tone mapping algorithm. In a previous work, we developed an ASIP for this algorithm using ISA extension [81]. The LISA ADL [29], was used for that research. In this section, we follow the same design process using the PolyCuSP environment. The results can serve as a source of comparison between our environment and a commercial ADL.

(HDR imaging can capture scenes commensurate with the real-world luminance, which is on the order of 10^8:1. However, the captured luminance values by this technique can be larger than the range perceptible by the human eye and the range supported by some display technology. Rendering HDR images on screens with reduced contrast while maintaining a reasonable perceptual match with the real scene requires a special technique called Tone Mapping (TM) [82]. TM algorithms usually impose high demands on computational resources. Customized processors are in-

teresting options for implementing these algorithms since they can provide a trade-off between the efficiency of a dedicated solution and the flexibility of a programmable solution.

We have chosen a well-known TM algorithm proposed by Reinhard *et al.* [83] as the target application.

In this method, the HDR image is first converted from RGB color space to scene-referred luminance (or world luminance) values as follows:

$$L_w = 0.265R + 0.670G + 0.064B. \tag{3.1}$$

Analogous to photography practices, the scene's key is identified by calculating the log-average luminance value:

$$L_{avg} = exp\left(\frac{1}{N}\sum \log(\delta + L_w)\right), \tag{3.2}$$

where N is the number of pixels in the image and δ is a small value. Next, the initial scaling is computed using (3.3), where the log-average is mapped to a desired value α.

$$L = \frac{\alpha}{L_{avg}} L_w \tag{3.3}$$

Then, the compressed luminance L_d is obtained by applying a linear mapping based on (3.4), as follows:

$$L_d = \frac{L\left(1+\frac{L}{L_{white}^2}\right)}{1+L}, \tag{3.4}$$

where L_{white} is the smallest luminance value to be mapped to pure white. By default, L_{white} is the maximum luminance value after the initial scaling. The final step is to recover the tone-mapped color image.

$$RGB_d = \frac{L_d}{L_w} RGB \tag{3.5}$$

Three CI were selected through analyzing the computational flow graph of Reinhard's algorithm. The first CI, called *LumCI*, realizes Equation 3.1 using three multipliers and two adders. The second CI, *LogCI*, facilitates the calculation of the approximation technique adopted for evaluation

of the logarithm function. The last CI, *MaxCI*, accelerates the compare and store process required for the L_{white} calculation. We have provided detailed descriptions of these three CIs elsewhere [81], where we employed the LISA language for ASIP development. In that work, we used LTRISC As the target processor. LTRISC is a 32-bit RISC-like processor model provided with Synopsys Processor Designer. Its Harvard architecture features 16 general-purpose registers and four pipeline stages.

In the experiments presented here, the MIPS I was used as baseline ISA. Although MIPS I includes more instructions and a larger register-file, compared to LTRISC, the two processors are still similar enough to ensure a fair comparison. To have compatible results, we synthesized the designs for a Xilinx XC5VLX30 FPGA. Since Reinhard's algorithm is based on fixed-point computation, specific fixed-point division and multiplication instructions (obviously by the same implementation) were added to the baseline processors before starting the experiments. These two instructions are necessary for obtaining acceptable accuracy results.

Figure 3.10 compares performance-per-area results (in terms of frame rate per area) of the processors generated by PolyCuSP with those obtained in our previous work [81]. The performance results were measured for 256×192 video frames. To simplify the comparison, only the results achieved by the fastest micro-architectural configuration are illustrated. According to the results, the PolyCuSP processors yield an average of 38% higher performance-per-area compared to the LTRISC ones. These results demonstrate the greater effectiveness of our RTL code generation, compared to existing commercial solutions. Modern FPGAs have dedicated DSP and RAM blocks for high performance realization of the memory elements and major arithmetic functions. By default, synthesis tools map registers and multipliers into these dedicated resources when possible. To have a unique and illustrative metric for area usage measurement, we avoided utilization of these dedicated resources in the experiments by adjusting the related settings in the synthesizer.

Table 3.2 presents an assessment of development complexity in the two environments. In terms of coding length, the LTRISC requires about 307% more programming. The ISA extension process usually starts with finding the well-suited encoding format in the processor architecture based on the input and output data of the intended CI.

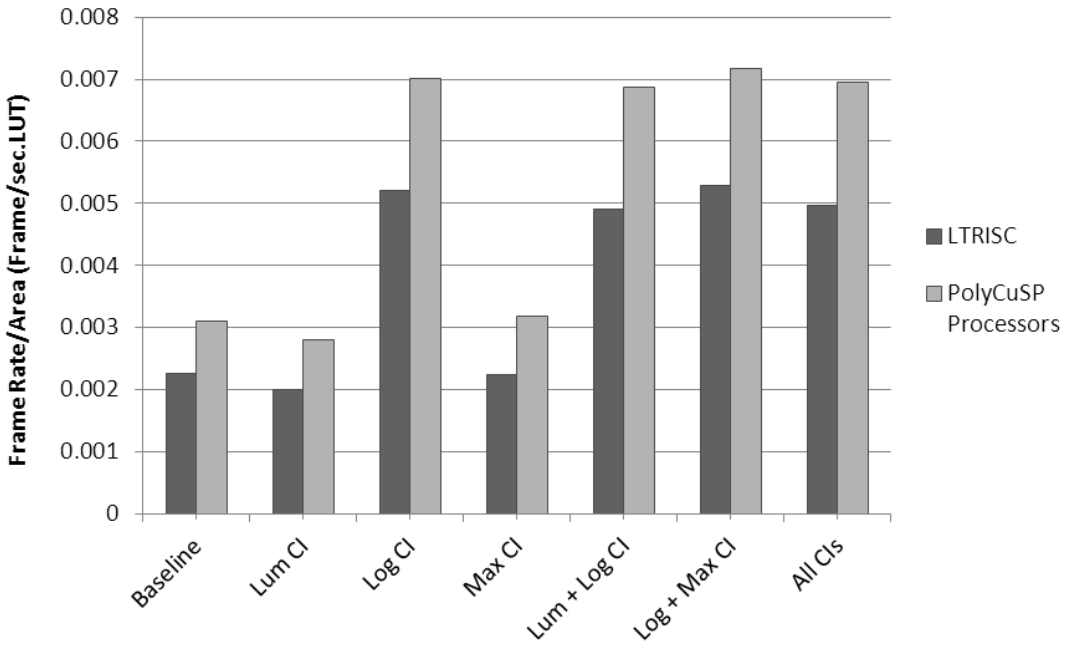


Figure 3.10 Comparing efficiency of PolyCuSP and the LISA-based ASIP design.

Table 3.2 Design complexity of LISA LTRISC and PolyCuSP processor

Factor	Development	LTRISC [81]	PolyCuSP
Number of lines	Baseline	1477	480
	ISA Extension	181	144
	ISA Subsetting	285	135
Development time (hours)	Baseline	-	26
	ISA Extension	~7	~6
	ISA Subsetting	~6	<1

The LISA description is composed of several LISA operations. These operations are organized in a tree-like structure. The functionality, coding and syntax of the instructions are distributed among the operations. To figure out existing encoding formats in a given model, the designer needs to process most of the functions manually. This process is notably easier in the PolyCuSP environment, in which all encoding formats are defined by a few XML elements. Encoding data determination (e.g., opcode value) and functional description form the next major steps of in-

struction-set extension. The complexity of these two steps is comparable in the two description methods. Finally, due to the distributed structure of the instruction description in LISA, removing a specific instruction usually entails manipulation of several operations. Accordingly, ISA subsetting is much more complex in LISA than in the PolyCuSP environment.

3.6 Conclusion

Application-specific customization of programmable processors is widely accepted as an effective approach to improve efficiency of processor centric designs. This chapter introduced a new custom processor development environment, called PolyCuSP, that takes advantage of a novel processor description method. Supporting a significant range of configurable microarchitectural parameters, straightforward instruction-set description format, and automatic generation of RTL along with cycle-accurate simulation models enable PolyCuSP to support rapid design space exploration.

Three image processing applications were used to evaluate the efficiency of the proposed environment. Microarchitectural exploration and instruction-set extension were examined to determine efficient trade-offs. Reported results demonstrate the applicability and efficiency of the proposed design environment.

CHAPTER 4 FIXED-POINT ERROR MODELING METHOD

In this chapter, we present an improved finite-precision error modeling approach that addresses a common hazard of the existing AA-based approaches and offers enhanced accuracy. The error modeling is an essential part of any fixed-point WLO method, which is one of the main subjects of this thesis. The error modeling approach introduced in this chapter is used in Chapter 5 and 6 for WLO in hardware synthesis and microprocessor platforms, respectively.

The contents of this chapter are largely extracted from our paper "Finite-precision error modeling using affine arithmetic," presented in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* in 2013 [84].

4.1 Introduction

Finite-precision error modeling is a key step in accuracy-aware fixed-point design. Error models are essential for word-length optimization methodologies, which have been the subject of a large body of research in the last decade. Accuracy and computational complexity are the main factors for the evaluation of an error modeling approach.

Many simulation-based and analytical techniques have been introduced in the literature for error modeling [51, 61, 62]. IA is a well-known analytical method which was originally invented to find the range of signals in a computational circuit [85]. The main drawback of IA is that it ignores the correlation among signals [50, 51]. AA is a preferable approach that addresses the correlation problem by taking signal interdependency into account. In AA, each signal is represented as a linear combination of certain primitive variables which stand in for sources of uncertainty. Fang *et al.* [51] and Lee *et al.* [38] introduced AA-based error propagation methods for quantization error modeling. Other works have tried to improve the AA-based method, but they typically fail to cover all features of the basic method [59].

This chapter illustrates a common hazard in existing AA-based error modeling approaches and proposes a solution to address it. The chapter also suggests a modification of the propagation process which can effectively improve error model accuracy. Improvements are demonstrated and quantified using a set of widely used case studies.

Section 4.2 briefly reviews necessary affine arithmetic concepts. Section 4.3 describes existing error propagation models and their shortcomings. Section 4.4 presents our proposed solution. Section 4.5 gives experimental results and comparisons, and Section 6 concludes the chapter.

4.2 Affine arithmetic

In affine arithmetic, the estimated value $\hat{x}$ of a signal x is represented by the sum of a constant x_0 and a finite set of n uncertainty terms $x_i\varepsilon_i$, as follows:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n, \quad \varepsilon_i \in [-1, 1]. \tag{4.1}$$

Each ε_i element is an independent uncertainty factor of the total uncertainty of the signal. The estimated value of a signal x with specified range $[x_{min}, x_{max}]$ is represented in affine form by

$$\hat{x} = x_0 + x_1\varepsilon_1, \tag{4.2}$$

where

$$x_0 = \frac{x_{min}+x_{max}}{2}, \quad x_1 = \frac{x_{min}-x_{max}}{2}.$$

The affine form for addition-subtraction is calculated by adding-subtracting the affine expression of the inputs. Multiplication is more complex due to the emergence of non-affine terms in the result expression. The widely used solution is to replace these terms with an affine approximation that introduces a new uncertainty factor [38, 51, 52].

The range of each signal is calculated from its affine representation by finding the minimum and maximum values when the uncertainty factors are replaced by -1 or 1.

In error modeling with AA, the quantization errors must be added to the affine forms. The quantized value x_q of a signal x is represented in affine form as

$$x_q = x + E_{x_q}, \tag{4.3}$$

where

$$E_{x_q} = \begin{cases} 2^{-FWL_{x_q}-1}.\varepsilon_x & Round\ to\ nearest \\ 2^{-FWL_{x_q}}.\varepsilon_x & Truncation \end{cases}, \quad \varepsilon \in [-1, 1].$$

The E_{x_q} and FWL_{x_q}terms correspond to the quantization error and fractional word-length of the quantized signal x_q, respectively. In Equation (4.3), we show two quantization approaches: truncation and round to nearest. With FWL fractional bits, truncation and round to nearest cause a maximum error of $2^{-FWL}$and 2^{-FWL-1}, respectively. To keep consistency with existing works, we use the round to nearest approach in the rest of this chapter. Equation (4.2) is a simplified version of the affine representation that is sufficient for finite-precision error modeling. A more detailed treatment can be found in [51].

Using the affine expression from (4.3), the quantization error from addition-subtraction is obtained as follows:

$$z_q = x_q \pm y_q = x \pm y + E_{x_q} \pm E_{y_q} + \delta$$

$$\Rightarrow E_{z_q} = E_{x_q} \pm E_{y_q} + \delta \tag{4.4}$$

where

$$\delta = \begin{cases} 2^{-FWL_{z_q}-1}\varepsilon_z, & FWL_{z_q} < \max(FWL_{x_q}, FWL_{y_q}) \\ 0, & otherwise \end{cases}.$$

From (4.4), we see that the total error at the output of an addition is equal to sum of errors of its operands added to the quantization error of the output signal. The δ is nonzero only when the fractional part of the output is narrower than at least one operand.

For multiplication, the error is:

$$E_{z_q} = xE_{y_q} + yE_{x_q} + E_{y_q}E_{x_q} + \delta \tag{4.5}$$

where

$$\delta = \begin{cases} 2^{-FWL_{z_q}-1}\varepsilon_z, & FWL_{z_q} < FWL_{x_q} + FWL_{y_q} \\ 0, & otherwize \end{cases}.$$

In a commonly used conservative approximation [38, 51], the *x* and *y* terms in (4.5) are replaced by the maximum absolute values of the *x* and *y* inputs that can be inferred from the range values.

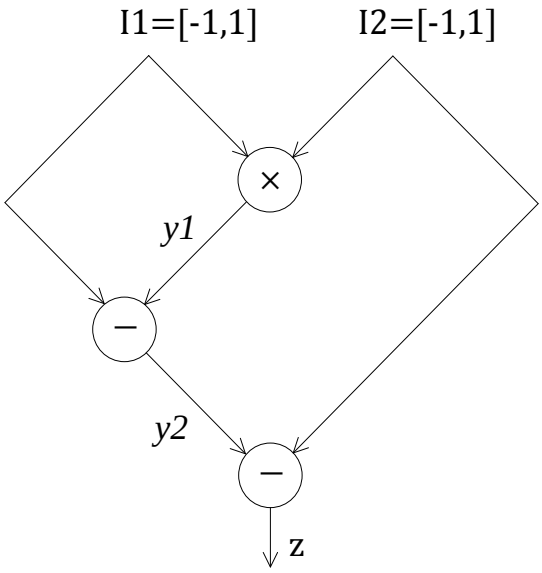


Figure 4.1 Example circuit with input range values shown in brackets.

4.3 Existing error propagation method

To compute the finite-precision error of a computational flow which is composed of consecutive elementary operations, an error propagation procedure is required.

Fang *et al.* [51], Lee *et al.* [38] and Pu *et al.* [86] presented the widely accepted reference methods for AA-based error propagation. As a main contribution, this chapter identifies a common hazard that may arise in all error modeling approaches used in these works. We illustrate this hazard by applying Lee's error modeling approach [38] on the example shown in Figure 4.1. The same problem emerges in Fang's and Pu's methods as well. Lee's method omits the conditional terms of the error equation by assuming there is always a quantization step after each operation. In other words, the number of output fractional bits of the operations is always assumed to be shorter than the maximum number of meaningful fractional bits that is produced by that operation. So, the error model for addition/subtraction becomes

$$E_{z_q} = E_{x_q} \pm E_{y_q} + 2^{-FWL_{z_q}-1}\varepsilon_z. \tag{4.6}$$

For multiplication it is

$$E_{z_q} = xE_{y_q} + yE_{x_q} + E_{y_q}E_{x_q} + 2^{-FWL_{z_q}-1}\varepsilon_z. \tag{4.7}$$

Accordingly, the error models of the signals of Figure 4.1 are obtained as follows:

$$E_{I1} = 2^{-FWL_{I1}-1}\varepsilon_1 \tag{4.8}$$

$$E_{I2} = 2^{-FWL_{I2}-1}\varepsilon_2$$

$$E_{y1} = (I1 \times E_{I2}) + (I2 \times E_{I1}) + E_{I1}E_{I2} + 2^{-FWL_{y1}-1}\varepsilon_3$$

$$E_{y2} = E_{y1} - E_{I1} + 2^{-FWL_{y2}-1}\varepsilon_4$$

$$E_z = E_{y2} - E_{I2} + 2^{-FWL_z-1}\varepsilon_5.$$

The hazard arises in the substitution of the signal terms that emerge in the multiplication expressions. Common approaches, such as Lee's method, substitute these terms with the absolute value of the worst-case bound of the signals, regardless of the rest of the circuit. This has been claimed to be a conservative approximation. So, the $I1$ and $I2$ terms in the expression for E_{y1} above are substituted by the value 1.

Now, starting from the primary inputs, the error terms are substituted by their corresponding expression until the output error expression is obtained as follows:

$$\begin{aligned} E_z &= E_{I1} + E_{I2} - E_{I1} - E_{I2} + E_{I1}E_{I2} + 2^{-FWL_{y1}-1}\varepsilon_3 \\ &\quad +2^{-FWL_{y2}-1}\varepsilon_4 + 2^{-FWL_z-1}\varepsilon_5 \\ &= E_{I1}E_{I2} + 2^{-FWL_{y1}-1}\varepsilon_3 + 2^{-FWL_{y2}-1}\varepsilon_4 + 2^{-FWL_z-1}\varepsilon_6. \end{aligned} \tag{4.9}$$

The $E_{I1}E_{I2}$ term in (4.9) is small and is often disregarded [50, 51]. Since the ε elements are in the range of -1 to 1, the upper bound of the error at the output z is calculated as

$$max(E_z) = 2^{-FWL_{I1}-FWL_{I2}-2} + 2^{-FWL_{y1}-1} + 2^{-FWL_{y2}-1} + 2^{-FWL_z-1}. \tag{4.10}$$

The incorrectness of the error expression in (4.9) and (4.10) can be easily shown by an example: In Equation (4.8), If $I1 = I2 = 0$, the output error expression becomes

$$\acute{E}_z = -E_{I1} - E_{I2} + E_{I1}E_{I2} + 2^{-FWL_{y1}-1}\varepsilon_3 + 2^{-FWL_{y2}-1}\varepsilon_4 \quad (4.11)$$
$$+2^{-FWL_O-1}\varepsilon_5$$

$$\Rightarrow max(\acute{E}_z) = 2^{-FWL_{I1}-1} + 2^{-FWL_{I1}-1} + 2^{-FWL_{I1}-FWL_{I2}-2}$$
$$+2^{-FWL_{y1}-1} + 2^{-FWL_{y2}-1} + 2^{-FWL_z-1},$$

which is larger than the value in (4.10) due to the contribution of the input quantization errors (E_{I1} and E_{I2}). In fact, the early substitution of the signal terms in (4.8) causes incorrect cancellation in later stages.

Cong *et al.* [50] used a more complex propagation method that does not generate the hazard. In this method, the signal terms are substituted with their corresponding affine representation instead of worst-case approximation. In the example of Figure 4.1, the $I1$ and $I2$ terms in E_{y1} expression are substituted with $0 + \varepsilon_6$ and $0 + \varepsilon_7$, respectively.

$$E_{y1} = 2^{-FWL_{I2}-1}\varepsilon_2\varepsilon_6 + 2^{-FWL_{I1}-1}\varepsilon_1\varepsilon_7 + 2^{-FWL_{y1}-1}\varepsilon_3. \quad (4.12)$$

The products $\varepsilon_2\varepsilon_6$ and $\varepsilon_1\varepsilon_7$ are replaced by new uncertainty factors as follows:

$$E_{y1} = 2^{-FWL_{I2}-1}\varepsilon_8 + 2^{-FWL_{I1}-1}\varepsilon_9 + 2^{-FWL_{y1}-1}\varepsilon_3. \quad (4.13)$$

So, the output error expression is obtained as:

$$E_z = 2^{-FWL_{I2}-1}\varepsilon_8 + 2^{-FWL_{I1}-1}\varepsilon_9 + 2^{-FWL_{y1}-1}\varepsilon_3 - 2^{-FWL_{I1}-1}\varepsilon_1 \quad (4.14)$$
$$-2^{-FWL_{I2}-1}\varepsilon_2 + 2^{-FWL_{y2}-1}\varepsilon_4 + 2^{-FWL_O-1}\varepsilon_5$$
$$\Rightarrow \max(E_z) = 2^{-FWL_{I2}} + 2^{-FWL_{I1}} + 2^{-FWL_{y1}-1} + 2^{-FWL_{y2}-1} + 2^{-FWL_z-1}.$$

This is the correct error model for Figure 4.1. The $E_{I1}E_{I2}$ term is neglected in Cong's paper [50].

Although this method can address the mentioned hazard, we found that it also faces a significant issue. Cong's method can occasionally fail to keep track of the correlation between new uncertainty factors. For example, applying this method on the circuits shown in Figure 4.2 that calculates $Dout = (A \times B \times C) - (A \times B \times C) = 0$. The error expressions are obtained as follows:

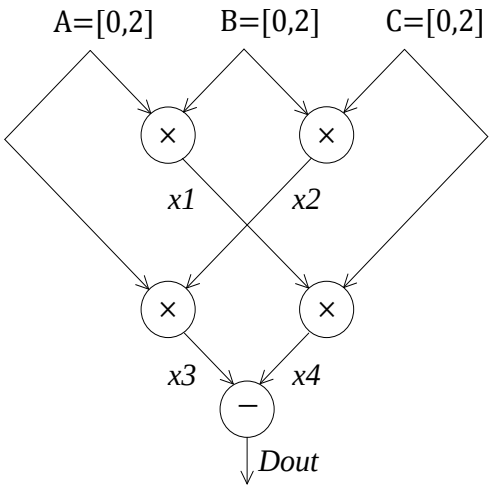


Figure 4.2 Circuit that calculates $Dout = (A \times B \times C) - (A \times B \times C)$.

$$E_A = 2^{-FWL_A-1}\varepsilon_4 \tag{4.15}$$

$$E_B = 2^{-FWL_B-1}\varepsilon_5$$

$$E_C = 2^{-FWL_B-1}\varepsilon_6$$

$$E_{x1} = (1+\varepsilon_1)2^{-FWL_B-1}\varepsilon_5 + (1+\varepsilon_2)2^{-FWL_A-1}\varepsilon_4$$
$$= 2^{-FWL_B-1}\varepsilon_5 + 2^{-FWL_B-1}\varepsilon_7 + 2^{-FWL_A-1}\varepsilon_4 + 2^{-FWL_A-1}\varepsilon_8$$

$$E_{x2} = (1+\varepsilon_2)2^{-FWL_C-1}\varepsilon_6 + (1+\varepsilon_3)2^{-FWL_B-1}\varepsilon_5$$
$$= 2^{-FWL_B-1}\varepsilon_6 + 2^{-FWL_B-1}\varepsilon_9 + 2^{-FWL_C-1}\varepsilon_5 + 2^{-FWL_C-1}\varepsilon_{10}$$

$$E_{x3} = \left((1+\varepsilon_1) \times E_{x2}\right) + \left((2+2\varepsilon_{11}) \times E_A\right)$$
$$= (2^{-FWL_B-1}\varepsilon_6 + 2^{-FWL_B-1}\varepsilon_9 + 2^{-FWL_C-1}\varepsilon_5 + 2^{-FWL_C-1}\varepsilon_{10} + 2^{-FWL_B-1}\varepsilon_{12} +$$
$$2^{-FWL_B-1}\varepsilon_{13} + 2^{-FWL_C-1}\varepsilon_{14} + 2^{-FWL_C-1}\varepsilon_{14}) + (2^{-FWL_A}\varepsilon_4 + 2^{-FWL_A}\varepsilon_{15})$$

$$E_{x4} = \left((2+2\varepsilon_{16}) \times E_B\right) + \left((1+\varepsilon_3) \times E_{x1}\right)$$
$$= (2^{-FWL_B}\varepsilon_5 + 2^{-FWL_B}\varepsilon_{17}) + (2^{-FWL_B-1}\varepsilon_5 + 2^{-FWL_B-1}\varepsilon_7 + 2^{-FWL_A-1}\varepsilon_4 +$$
$$2^{-FWL_A-1}\varepsilon_8 + 2^{-FWL_B-1}\varepsilon_{18} + 2^{-FWL_B-1}\varepsilon_{19} + 2^{-FWL_A-1}\varepsilon_{20} + 2^{-FWL_A-1}\varepsilon_{21})$$

$$E_{Dout} = E_{x3} - E_{x4}.$$

Note that in order to simplify calculations, we do not take the quantization error of intermediate signals ($e.g., x1$- $x4$) into account.

In a correct propagation approach, the quantization errors of the input signals should be cancelled in the subtraction operation before reaching the output $Dout$ error expression. Hence, the correct output error expression is $E_{Dout} = 0$. However, Cong's method gives a different error expression due to the fact that this method fails to keep track of the correlation among new uncertainty factors. As illustrated by the example in Figure 4.2, this issue can cause considerable overestimation of the error values.

Another significant challenge of Cong's method [50] is the large number of uncertainty factors that it introduces in multiplication operations. This leads to high computational complexity of the propagation process.

4.4 Proposed approach

We propose a modified error propagation approach with two improvements. The first improvement addresses the shortcomings of the existing methods. The second improvement involves the propagation of conditional terms which can enhance the accuracy.

4.4.1 Postponed substitution

An effective way to address the problems described in Section 4.3 is to postpone the signal term substitution until the last stage of the propagation process. Applying this modification to the circuit of Figure 4.1, the $I1$ and $I2$ terms in $E_{y1}$are not substituted by a value until the output error E_z is obtained as:

$$E_z = (I1 - 1) \times 2^{-FWL_{I2}-1}\varepsilon_2 + (I2 - 1) \times 2^{-FWl_{I1}-1}\varepsilon_1 \qquad (4.16)$$
$$+2^{-FWL_{y1}-1}\varepsilon_3 + 2^{-FWL_{y2}-1}\varepsilon_4 + 2^{-FWL_z-1}\varepsilon_5.$$

Now, for a conservative evaluation, non-constant coefficients of the error terms are approximated to their maximum absolute values. For example, the $(I1 - 1)$ coefficient is approximated to $max(|I1 - 1|) = 2$ applying $I1 = -1$, while a value of 0 was calculated for this coefficient in

Lee's method due to early substitution. The upper bound error expression of (4.14) is eventually obtained for the circuit of Figure 4.1 using our method. This case study shows that although early substitution of signal terms simplifies the error propagation process, it may lead to an underestimation of error bounds and consequently inaccuracy in precision analysis. Applying this method on the circuit of Figure 4.2 gives the correct result, i.e., $E_{Dout} = 0$. This shows that our method also addresses the overestimation issue of Cong's approach.

Solving the maximization function in the last stage forms the main complexity overhead of our approach. This overhead increases with the number of signal terms that participate in a non-constant coefficient.

4.4.2 Propagation of conditional terms

The second proposed improvement involves the conditional terms in error expressions. Existing methods assume that the quantization error is always introduced at the output of operations to eliminate conditional elements. Although this assumption simplifies the error propagation process, it may introduce unnecessary error elements that eventually lead to overestimation. Our second improvement proposes avoiding this assumption by propagating the conditional terms in the same way as other elements.

This improvement is shown by a real example. Figure 4.3 depicts the RGB-to-YCrCb function. This widely used function calculates

$$\begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix} = MC \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{4.17}$$

where

$$MC = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{bmatrix}.$$

The input signals R, G, and B are assumed to be eight-bit unsigned integers. The existing AA-based error modeling approach gives the following error bound expressions for the RGB-to-YCrCb function.

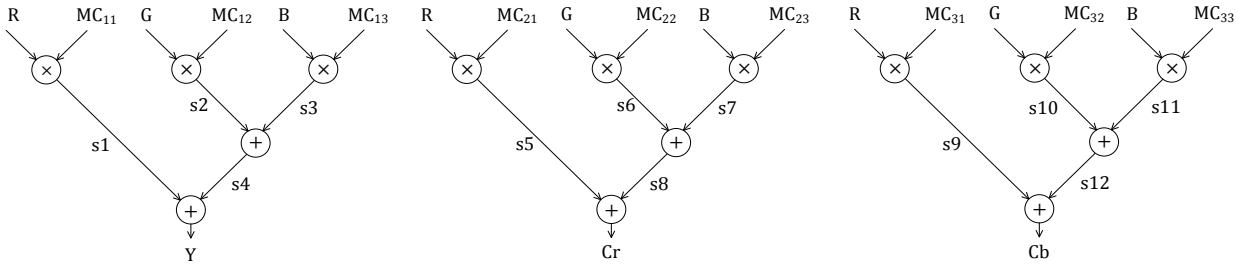


Figure 4.3 RGB-to-YCrCb example.

$$\begin{aligned}\mathrm{Max}(E_Y) &= \mathrm{R}\times 2^{-FWL_{MC_{11}}-1} + \mathrm{G}\times 2^{-FWL_{MC_{12}}-1} + \mathrm{B}\times 2^{-FWL_{MC_{13}}-1} \\ &+2^{-FWL_{s1}-1} + 2^{-FWL_{s2}-1} + 2^{-FWL_{s3}-1} + 2^{-FWL_{s4}-1}\end{aligned} \tag{4.18}$$

$$\begin{aligned}\mathrm{Max}(E_{Cr}) &= \mathrm{R}\times 2^{-FWL_{MC_{21}}-1} + \mathrm{G}\times 2^{-FWL_{MC_{22}}-1} + \mathrm{B}\times 2^{-FWL_{MC_{23}}-1} \\ &+2^{-FWL_{s5}-1} + 2^{-FWL_{s6}-1} + 2^{-FWL_{s7}-1} + 2^{-FWL_{s8}-1}\end{aligned}$$

$$\begin{aligned}\mathrm{Max}(E_{Cb}) &= \mathrm{R}\times 2^{-FWL_{MC_{31}}-1} + \mathrm{G}\times 2^{-FWL_{MC_{32}}-1} + \mathrm{B}\times 2^{-FWL_{MC_{33}}-1} \\ &+2^{-FWL_{s9}-1} + 2^{-FWL_{s10}-1} + 2^{-FWL_{s11}-1} + 2^{-FWL_{s12}-1}\end{aligned}$$

In above expressions, all conditional terms have been replaced by the constant error terms for s1-s12 intermediate signals.

The proposed modification suggests keep of the conditional terms. For instance, applying this modification on the RGB-to-YCrCb circuit gives the following upper bound error expression for the Cr output:

$$\begin{aligned}\mathrm{Max}(E_{Cr}) &= \mathrm{R}\times 2^{-FWL_{MC_{21}}-1} + \mathrm{G}\times 2^{-FWL_{MC_{22}}-1} + \mathrm{B}\times 2^{-FWL_{MC_{23}}-1} \\ &+\delta_{s5} + \delta_{s6} + \delta_{s7} + \delta_{s8}\end{aligned} \tag{4.19}$$

where

$$\delta_{s5} = \begin{cases} 2^{-FWL_{s5}-1}, & FWL_{s5} < FWL_{MC_{21}} \\ 0, & otherwize \end{cases}$$

$$\delta_{s6} = \begin{cases} 2^{-FWL_{s6}-1}, & FWL_{s6} < FWL_{MC_{22}} \\ 0, & otherwize \end{cases}$$

$$\delta_{s7} = \begin{cases} 2^{-FWL_{s7}-1}, & FWL_{s7} < FWL_{MC_{23}} \\ 0, & otherwize \end{cases}$$

$$\delta_{s8} = \begin{cases} 2^{-FWL_{s8}-1}, & FWL_{s8} < max(FWL_{s6}, FWL_{s7}) \\ 0, & otherwize \end{cases}$$

The error expression calculated in (4.18) is less accurate than that in (4.19) since it does not take into account the fact that, in certain circumstances, no quantization error is introduced into the calculations in intermediate signals. In other words, existing approaches simplify the error modeling at the expense of risking overestimation. In FWL allocation, using the error expression of the modified approach (4.19) can potentially lead to more efficient results since the improved approach can gives a more accurate and smaller error estimation for a unique FWL allocation solution. For instance, in the RGB-to-YCrCb example, the modified error model leads to a FWL solution of

$$\{FWL_{MC_{21}}, FWL_{MC_{22}}, FWL_{MC_{23}}, FWL_{s5}, FWL_{s6}, FWL_{s7}, FWL_{s8}\} = \{17, 17, 17, 12, 12, 11, 12\} \quad (4.20)$$

to satisfy 8 bit accuracy at the output *Y* while the error model given by the existing approach rejects it. So, according to the existing approach, the FWL of at least one signal must be increased to meet the accuracy requirements and this may lead to more hardware cost and latency.

Experimental results in Section 4.5 demonstrate the effectiveness of this modification on word length optimization.

4.5 Results and comparison

We developed a word length optimization system to evaluate the efficiency of the conditional term propagation. This system is a reproduction of the method proposed by Osborne *et al.* [52] and is implemented in MATLAB. The error models are given to the word-length optimization process as input. Using a more accurate error model, the optimization process can potentially find shorter signal bit-lengths that still meet the requested output error bound. The shorter signals eventually lead to lower hardware cost during the hardware implementation. This section measures the hardware cost reduction and optimization time overhead obtained by using conditional term propagation in AA-based error modeling. For the experiments described in this sec-

tion, we have employed five well-known case studies that include some commonly used transforms and operators in the signal and image processing domains.

Lee *et al.* [38] described the case studies in detail. For hardware cost measurement, the case studies were modeled in VHDL. The polynomial approximation was implemented in general form, contrary to Lee *et al.* [38] who customized this approximation to a specific function. All experiments were performed on an Intel i7 3-GHz PC with 16 GB RAM. Designs were synthesized with Synplify 9.1 for a Xilinx Virtex 5 XC5VLX110 FPGA.

Table 4.1 illustrates the hardware area cost and optimization time for the five case studies. In these results, conditional term propagation saves hardware area by up to 7.0% at the expense of negligible complexity overhead. Achieved hardware savings are significant regarding the competitive results in previous works in this area.

Figure 4.4 compares the hardware savings obtained in various degrees of polynomial approximation. These results show a slight growth in hardware savings by increasing the application size. The requested output precision was fixed to 8-bits in all reported experiments.

Table 4.1 Impacts of conditional term propagation on efficiency of the results

Case Study	**# of Signals**	**Area (slices)**			**Opt. Time (s)**	
		M1*	**M2****	**Imp (%)**	**M1**	**M2**
Degree-4 Poly	13	1234	1172	5.3	5.1	5.2
B-Spline	15	719	691	4.1	3.5	3.5
RGB to YCrCb	19	558	537	3.9	5.7	5.9
2×2 Matrix Mult.	29	1939	1812	7.0	14.4	14.7
DCT 8×8	55	5178	4902	5.6	127.3	131.1

*M1: Without conditional term propagation **M2: With conditional term propagation

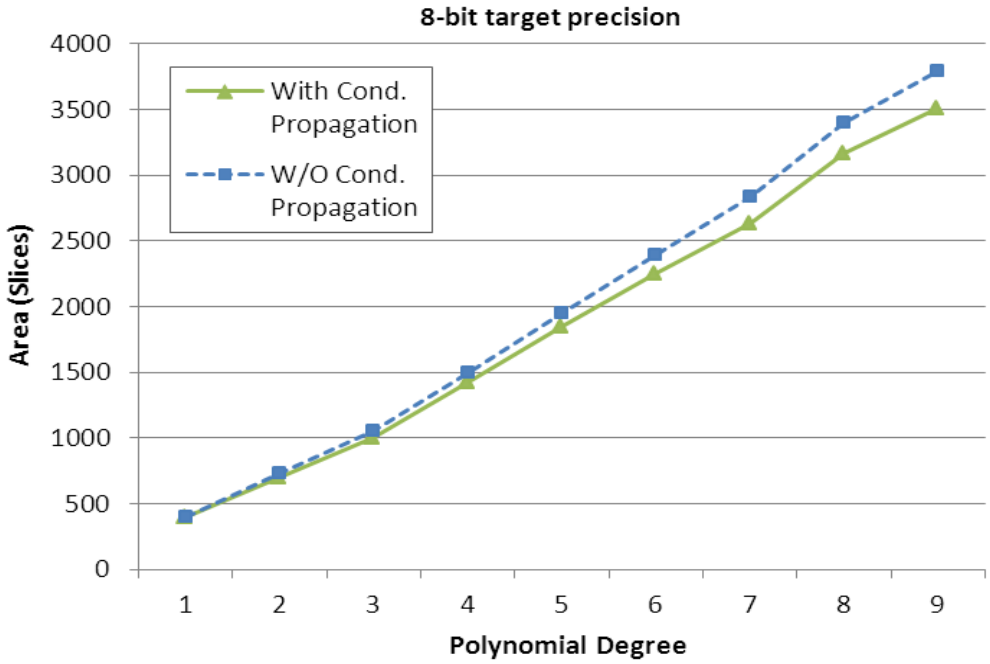


Figure 4.4 Hardware area savings for various polynomial degrees.

4.6 Conclusion

We have demonstrated a common hazard in existing AA-based finite-precision error modeling methods using two counter examples. We have proposed the postponed substitution approach to address this hazard. Furthermore, we have proposed conditional term propagation to enhance error modeling accuracy. The efficiency of our approach was evaluated through a set of case studies. The results show that the approach can yield significant hardware savings with negligible complexity overhead.

CHAPTER 5 WORD-LENGTH ALLOCATION FOR HARDWARE SYNTHESIS

In this chapter, we focus on word-length optimization that targets hardwired circuit design for fixed-point computations. Two new fractional word-length selection algorithms and an acceleration technique are introduced in this chapter.

The contents of this chapter are largely extracted from our paper "Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," published in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* in 2013 [87].

5.1 Introduction

Bit-width allocation is a key step in fixed-point computational circuit design. This process has significant impact on accuracy and hardware efficiency of fixed-point circuits. Design flexibility of reconfigurable hardware devices, such as FPGAs and ASICs allows the designers to customize the bit-width of each signal in a given application, independently. The goal of this customization is to optimize the trade-off between hardware efficiency and computational accuracy of the implementation. Finding the optimal bit-widths is widely known as word length selection or word length determination problem. Word-length allocation is an NP-hard problem [3]. Exhaustive search and manual evaluation methods are not practical for WLO in large designs due to the huge size of the search space.

WLO is composed of two main processes: IWL allocation and FWL allocation. IWL allocation requires range analysis to evaluate the range of values that each signal may take. Knowing the range of values, the number of IBs is simply obtained as the minimum value that ensures overflow avoidance. On the other hand, the goal of the FWL allocation is to minimize the hardware cost while meeting the accuracy requirements.

IWL and FWL allocation methods can be categorized into dynamic (or simulation-based) and static (or analytical) approaches. Dynamic methods are usually far slower due to the large number of input stimuli they need to process. They also cannot guarantee the accuracy of the result because of their dependency to the selected input stimuli. On the other hand, static methods offer

more conservative results. This may occasionally lead to overestimation of the range and/or error bounds and consequently less efficient results.

Finite-precision error modeling is a vital part of most analytical FWL allocation approaches. The error models express the precision of the outputs as a function of quantization error of input and intermediate signals. For a candidate FWL solution, the error model can be used to evaluate the upper-bound error that may appear at the output and verify its compliance to the requested error range.

Many simulation-based and analytical techniques have been introduced for these two problems in literature. Range and error propagation via AA is widely accepted as one of the best analytical approaches for both problems [38, 52].

This chapter introduces novel ideas for WLO for hardwired circuit synthesis using AA. It proposes two new FWL selection algorithms as well as a simplification technique to reduce the complexity of the FWL allocation problem. These contributions build on approaches presented by Lee *et al.* [38] and Osborne *et al.* [52] and offer trade-offs between hardware efficiency and optimization speed. Multi-output circuits are supported in all proposed algorithms and techniques.

An important factor in word length optimization is the error measurement metric. To keep consistency with related works, we have used maximum absolute error in terms of ulp (unit in the last place) in this chapter. We have evaluated the effectiveness of the proposed algorithms and techniques and compared them with previous works.

The rest of the chapter is organized as follows. Section 5.2 presents an overview of the proposed WLO framework that implements all ideas and algorithms introduced throughout the chapter. The proposed techniques and algorithms for FWL allocation are described in Section 5.3. Section 5.4 gives experimental results and comparisons, while Section 5.5 concludes the chapter.

5.2 Implementation framework overview

Figure 5.1 illustrates the WLO framework that implements the proposed algorithms and techniques described in this chapter. In this framework, the source application is first converted from C into the GIMPLE intermediate representation (IR) using the GCC compiler. This conversion facilitates reading and parsing the input code. The IR is then processed in four steps.

Step 1 in the optimization process is the AA-based range analysis. It requires the range of the primary inputs. The FWL allocation is made up of steps 2, 3 and 4. Step 2 consists of the limited-precision error modeling, which requires the range information and the required output precision. Step 3 is a simplification technique described in Section 5.3.2. Step 4 is a new algorithm to select the fractional bit-width of the signals. It takes as input the range information, the error model and the hardware cost estimation model of the given application. The outputs of the framework are the IWL and the FWL.

5.3 FWL allocation

This section describes our proposed FWL allocation approach, which is composed of a preliminary simplification technique (Step 3 in Figure 5.1) and a novel fractional bit-width selection process (Step 4 in Figure 5.1).

5.3.1 Example design

We use an example circuit to illustrate our method in subsequent sections. This example circuit, which is borrowed from Lee *et al.* [2], is given in Figure 5.2.

In this chapter, the enhanced error modeling approach presented in Chapter 4 is used to estimate worst case error bound. Using that approach, the upper bound error expression obtained for Figure 5.2 is:

$$2^{-FWL_{z_q}-1} \geq 2^{-FWL_{a_q}+2} + 2^{-FWL_{b_q}+1} + 2^{-FWL_{c_q}-1} + \delta_d + \delta_e \qquad (5.1)$$

where

$$\delta_d = \begin{cases} 2^{-FWL_{d_q}-1}\varepsilon_d, & FWL_{d_q} < FWL_{a_q} + FWL_{b_q} \\ 0, & otherwise \end{cases},$$

$$\delta_e = \begin{cases} 2^{-FWL_{e_q}-1}\varepsilon_e, & FWL_{e_q} < \max(FWL_{d_q}, FWL_{c_q}) \\ 0, & otherwise \end{cases},$$

In subsequent sections, this inequality will be used as the error bound model.

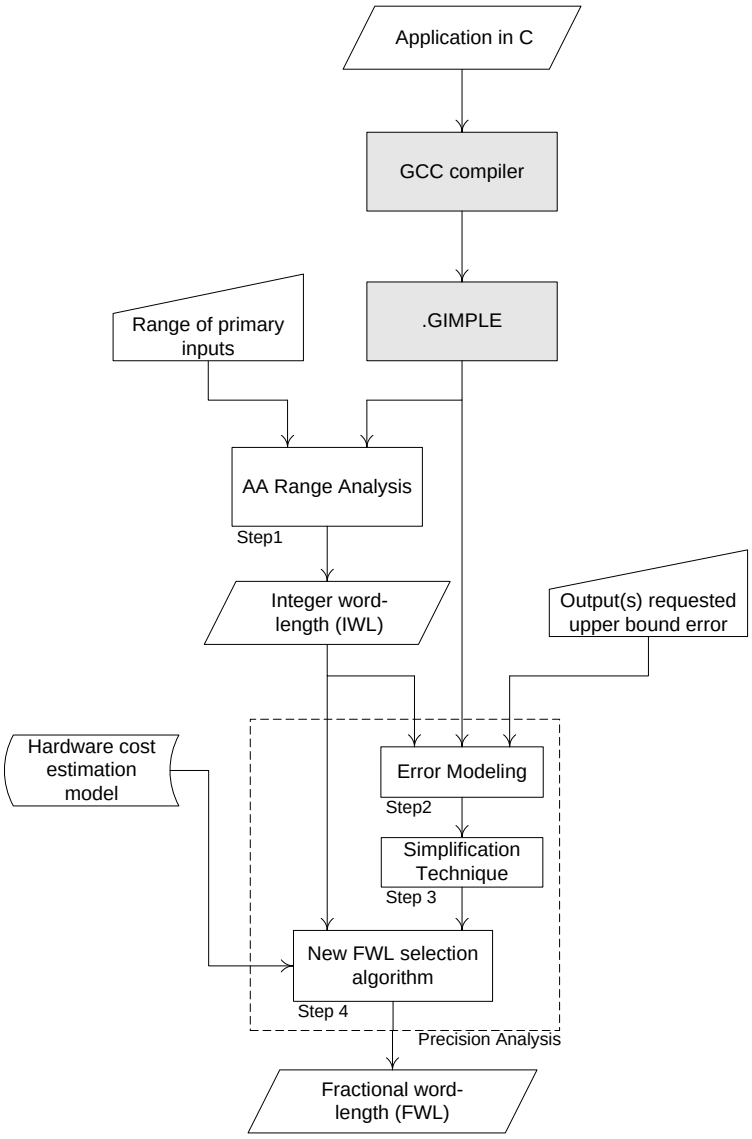


Figure 5.1 Overview of the proposed word length optimization framework.

5.3.2 Preliminary simplification technique

We propose the following Lemma, from which our simplification technique has been derived:

***Lemma*:** Quantization of an intermediate signal between two additions/subtractions does not have any benefit over quantizing the inputs of the preceding one and keeping the fractional length of the intermediate signal equal to the maximum FWL of the input signals.

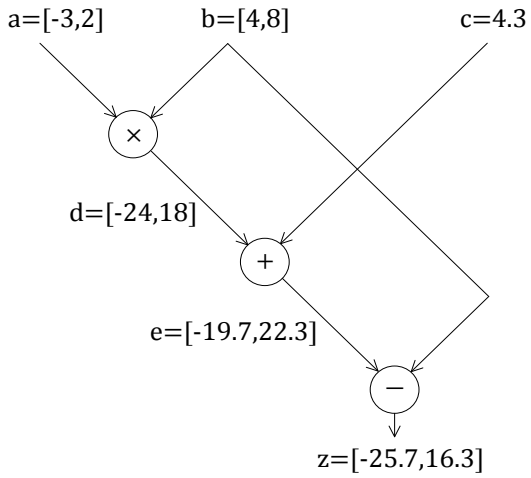


Figure 5.2 Example circuit along with the range information in brackets.

Applying the Lemma to the general subcircuit of Figure 5.3 suggests that to gain a constant accuracy at output *out*, the lowest hardware cost is achieved when

$$FWL_x = max(FWL_{in1}, FWL_{in2}). \tag{5.2}$$

***Proof*:** According to our error modeling approach of Chapter 4, the output error expression of Figure 5.3 circuit is:

$$max(E_{out}) = 2^{-FWL_{in1}-1} + 2^{-FWL_{in2}-1} + 2^{-FWL_{out}-1} + \delta_x \tag{5.3}$$

where

$$\delta_x = \begin{cases} 2^{-FWL_x-1}\varepsilon_x, & FWL_x < max(FWL_{in1}, FWL_{in2}) \\ 0, & otherwize \end{cases}.$$

Comparing the fractional widths of the intermediate signal x and inputs $in1$ and $in2$, the following three possibilities emerge:

1. $FWL_x \geq FWL_{in1}, FWL_x \geq FWL_{in2}$: In this case, the δ_x term is equal to zero. If we reduce the FWL_x to $\overline{FWL_x} = \max(FWL_{in1}, FWL_{in2})$, the error remains the same while the hardware cost can be reduced in proportion to the difference $FWL_x - \overline{FWL_x}$. Hence, reducing the fractional width of signal x, from FWL_x to $\overline{FWL_x}$, improves the efficiency.

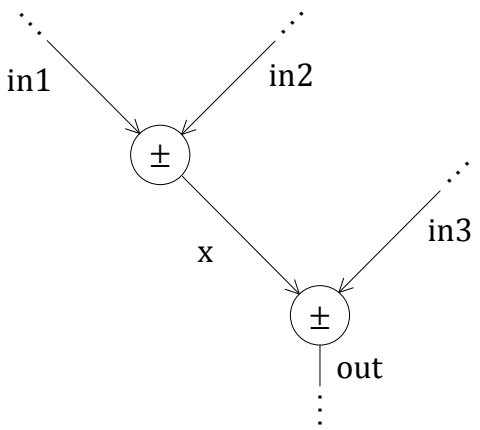


Figure 5.3 A general subcircuit that calculates out=in1±in2±in3.

2. $FWL_x < FWL_{in1}, FWL_x \geq FWL_{in2}$ or $FWL_x \geq FWL_{in1}$, $FWL_x < FWL_{in2}$: Let us assume that $FWL_x < FWL_{in1}$ to discuss this case. The other case can be proved in the same manner. If we reduce $FWL_{in1}$to FWL_x, the error equation becomes:

$$\max(\bar{E}_{out}) = 2^{-FWL_x-1} + 2^{-FWL_{in2}-1} + 2^{-FWL_{out}-1} < \max(E_{out}). \quad (5.4)$$

This reduction always results in smaller error and smaller hardware cost.

3. $FWL_x < FWL_{in1}, FWL_x < FWL_{in2}$: We can rewrite this condition as:

$$FWL_{in1}, FWL_{in2} \geq FWL_x + 1 \quad (5.5)$$

Then, there are two possible states:

The FWL of both inputs ($FWL_{in1}$and FWL_{in2}) are equal to $FWL_x + 1$: in this state, the output error expression is

$$\max(E_{out}) = 2^{-FWL_x} + 2^{-FWL_{out}-1}. \quad (5.6)$$

If the FWL_{in1} and FWL_{in2} are both reduced to FWL_x, the maximum error expression remains the same but the hardware cost of the preceding operator decreases due to one bit shorter operands.

The FWL of at least one of the inputs is longer than $FWL_x + 1$: in this state, if we change the FWL_{in1}, $FWL_{in2}$and FWL_x to $\overline{FWL}_{in1}$, $\overline{FWL}_{in2}$ and $\overline{FWL}_x$ respectively, while $\overline{FWL}_{in1}, \overline{FWL}_{in1}, \overline{FWL}_x = FWL_x + 1$ the upper bound error becomes:

$$max(\bar{E}_{out}) = 2 \times 2^{-FWL_x - 2} + 2^{-FWL_{out} - 1} < max(E_{out}). \quad (5.7)$$

This modification increases the FWL of the intermediate signal x by one bit, while reducing the width of the preceding operator by at least one bit. Hence, considering single fan-out for the intermediate signal, the hardware cost of the modified circuit would always be less than or equal to the primary circuit. Note that we assume uniform hardware cost for each extra bit in addition/subtraction. More precisely, this modification shows that a single fan-out signal between two addition/subtraction operators can always be removed from the word-length selection problem without taking the risk of disregarding more efficient solutions. For this purpose, these unnecessary signals must first be identified and then eliminated from the list of the signals which are to be considered in the FWL selection process.

/* N_{op}: number of existing operations in the circuit */

/* OP_i: i'th operation in the circuit while i $\in\{1,\ldots,N_{op}\}$ */

/* S_i: the output signal of OP_i */

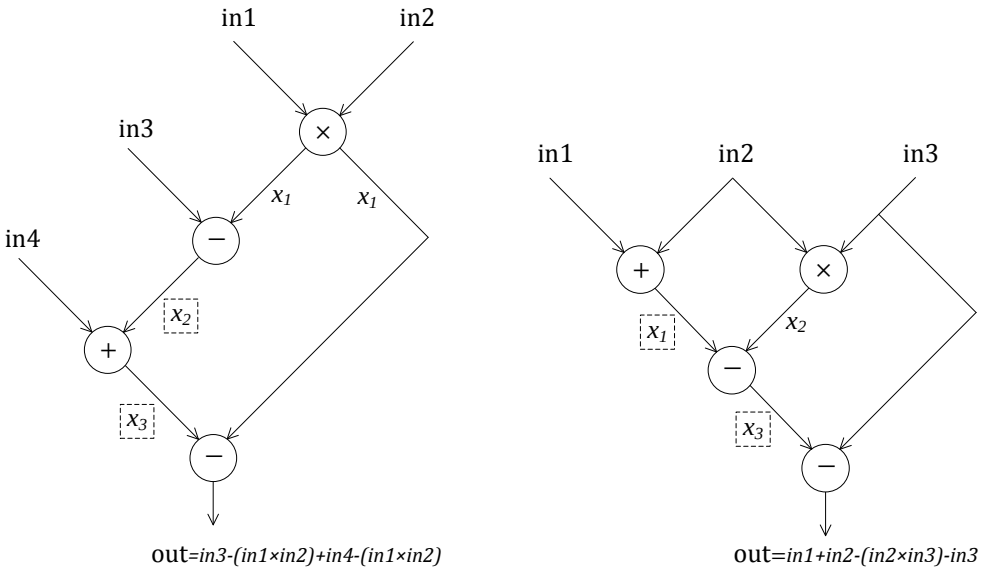
/* NF_i: Fan-out of signal S_i ; number of operations that use S_i as input */

/* FOP_{ij}: j'th operation that use signal S_i as input; j $\in\{1,\ldots,NF_i\}$ */

/* FIS_{ik}: k'th signal that operation OP_i receives as input;
k $\in\{1,2\}$ since we only consider two-input operations here */

for (all intermediate and output signals S_i)
  if (NF_i ==1 and OP_i and FOP_{i1} are addition/subtraction)
    Remove S_i from the list of signals that require word length selection.
    Replace the FWL_{S_i} by $max\left(FWL_{FIS_{i1}}, FWL_{FIS_{i2}}\right)$ in all error equations.
  end if;
end for;

Figure 5.4 The proposed algorithm for preliminary simplification technique.



$$FWL_{x_2} = max(FWL_{in3}, FWL_{x_1}) \qquad FWL_{x_1} = max(FWL_{in1}, FWL_{in2})$$
$$FWL_{x_3} = max(FWL_{in3}, FWL_{x_1}, FWL_{in4}) \qquad FWL_{x_3} = max(FWL_{in1}, FWL_{in2}, FWL_{x_2})$$

Figure 5.5 Applying the preliminary simplification algorithm to two example circuits.

The necessary changes must be made in error models assuming that the FWL of each removed signal is always equal to the maximum FWL of the last preceding operator. Figure 5.4 presents the algorithm we developed to realize this technique. It can significantly simplify the problem by pruning the search space. For example, applying this technique to the circuit of Figure 5.2, the signal e can be exempted from the selection problem assuming that

$$FWL_e = max\left(FWL_{d_q}, FWL_{c_q}\right). \tag{5.8}$$

Accordingly, the δ_e term reduces to zero and the error bound inequality of (5.1) is converted to:

$$2^{-FWL_{z_q}-1} \geq 2^{-FWL_{a_q}+2} + 2^{-FWL_{b_q}+1} + 2^{-FWL_{c_q}-1} + \delta_d \, . \tag{5.9}$$

Figure 5.5 illustrates the result of applying the simplification technique to two other example circuits. The signals marked by the dashed square are removed from the fractional width selection problem.

Note that the presented lemma was only proved in terms of worst case error. This does not guarantee the validity of the lemma when average error is intended.

5.3.3 Hardware cost estimation model

Hardware cost measurement is normally necessary for effective width selection. It is considered as the main efficiency metric by most works in this field [38, 61, 69]. A hardware cost estimation model is necessary for a fast optimization process.

The cost model which is used in this chapter is similar the one used by Lee *et al.* [38]. In this model, the full adder has unit cost. Hence, the cost of the addition/subtraction $x \pm y$ is modeled by $max(IWL_x, IWL_y) + max(FWL_x, FWL_y)$ while the cost of multiplication $x \times y$ is modeled by $(IWL_x + FWL_x)(IWL_y + FWL_y)$. A more precise model for addition and subtraction is to consider the minimum fractional word-length of the operands as the cost of the fractional part instead of the maximum one. It is because the FWLs must be left-aligned prior to addition, the operand with the fewest bits must be right-padded with zeros, and no hardware is necessary to add them. However, in this thesis, we use the Lee's model to keep consistency with previous works.

Based on this model, the hardware cost of the circuit shown in Figure 5.2 is:

$$\begin{aligned} \text{Cost} = {} & \left(\text{IWL}_{a_q} + \text{FWL}_{a_q}\right)\left(\text{IWL}_{b_q} + \text{FWl}_{b_q}\right) \\ & + \max\left(\text{IWL}_{d_q}, \text{IWL}_{c_q}\right) + \max\left(\text{FWL}_{d_q}, \text{FWL}_{c_q}\right) \\ & + \max\left(\text{IWL}_{e_q}, \text{IWL}_{b_q}\right) + \max\left(\text{FWL}_{d_q}, \text{FWL}_{c_q}, \text{FWL}_{b_q}\right) \end{aligned} \tag{5.10}$$

Increasing the fractional width of a signal normally increases the hardware cost while reducing the output error.

5.3.4 Progressive Selection Algorithm (PSA)

The first FWL selection algorithm we introduce, named PSA, is based on a fast progressive approach. The idea we pursue with this algorithm is to partition the allowed amount of error among

the involved signals based on their unit bit hardware cost. The algorithm is composed of three main phases.

1. *Marginal bit cost estimation*. During this phase, the hardware cost of each extra fractional bit is estimated for all signals. Since the hardware cost of a signal may depend on the bit-width of other signals, an initial width assumption is necessary. A simple and effective way is to use Uniform Fractional Bit-width (UFB) as this initial assumption. The UFB can be analytically calculated from the error model by substituting all FWL variables with a single variable in the error bound equation. In the example circuit of Figure 5.2, this calculation gives:

$$\max(E_{z_q}) = 2^{-UFB+1} + 2^{-UFB+2} + 3(2^{-UFB-1}). \tag{5.11}$$

   Assuming that 8-bit accuracy is requested at the output z, then:

$$2^{-9} \geq 2^{-UFB+1} + 2^{-UFB+2} + 2^{-UFB} \Rightarrow \text{UFB} \geq 12. \tag{5.12}$$

   In this case, the minimum value of UFB that meets the required accuracy is 12. Now, we can obtain the initial cost by substituting the IWLs with the range analysis results and substituting the FWLs with the calculated UFB (12 in this example) in the cost equation:

$$cost_0 = (3 + 12)(5 + 12) + 6 + 12 + 6 + 12 = 291 \tag{5.13}$$

   To compute the marginal cost of each signal, the above calculation is repeated by incrementing the FWL of that signal by a single unit. The difference of the result and the initial cost value is considered as the marginal bit cost.

2. Cost-based error allocation. During the second phase, the allowed error bound is divided into small pieces. Each piece corresponds to the maximum error that can be introduced by a signal. The proposed method performs the error division based on the hardware cost of the signals in order to optimize the total hardware cost. To describe the proposed allocation scheme, the error bound equation is first written in a sum-of-products form as follows:

$$max(E_{out}) = \sum_{i=1}^{N} C_{s_i} \times 2^{-FWL_{s_i}} \tag{5.14}$$

where N is the number of signals and s_i stands for the i[th] signal. Assuming non-zero values for conditional statements, the C_{s_i} terms are obtained as constant values in affine expressions. For example, the C_{s_i} values in error bound equation (5.4) are as follows:

$$\{C_a, C_b, C_c, C_d, C_e\} = \{4, 2, {}^1/_2, {}^1/_2, {}^1/_2\}. \tag{5.15}$$

In the proposed approach, the division of each product term in the output error has a direct relationship with the cost of the corresponding signal. The marginal bit cost data, calculated in the first phase, is used as the cost metric in this step. Therefore, the primary fractional width of the signals is calculated as:

$$C_{s_i} \times 2^{-FWL_{s_i}} = \frac{Ucost_{s_i} \times E_{Req}}{\sum_{j=1}^{N} Ucost_{s_j}}$$

$$\Rightarrow FWL_{s_i}^0 = -\left\lfloor \log_2 \frac{Ucost_{s_i} \times E_{Req}}{C_{s_i} \times \sum_{j=1}^{N} Ucost_{s_j}} \right\rfloor, \tag{5.16}$$

where

$S_i: i^{th}\ signal;\ \ N: number\ of\ signals;$

$E_{Req}: Requested\ output\ error$

$Ucost_{s_i}: The\ marginal\ bit\ cost\ of\ signal\ \ s_i$

The primary fractional width of the example circuit signals are computed with (5.16), as illustrated in first row of Table 5.1.

3. *Fractional width refinement:* The rounding down operation causes accumulative imprecision in primary width calculation. Hence, the output error resulting from the primary widths may be considerably smaller than the requested error bound. This error slack allows further refinement of the widths. For example, with the results obtained for the example of Figure 5.2 in the last phase, the output error is computed as follows:

$$\max(E_{z_q}) = 2^{-11} + 2^{-11} + 2^{-15} + 2^{-15} \tag{5.17}$$

$$< Requested\ error = 2^{-9}$$

We developed an iterative error slack allocation algorithm to refine the primary fractional width values. For each iteration, this algorithm selects the best candidate signal as the best signal for fractional width reduction. For efficient selection, this algorithm calculates the *Cost_Error* function for all eligible signals as follows:

$$Cost\_Error_{s_i} = \frac{Bit\_Cost_{s_i}}{Err\_Penalty_{s_i}} \tag{5.18}$$

where the parameters are defined as follows:

s_i: i'th signal

$Err\_Penalty_{s_i}$: amount of error that would be introduced by decreasing the width of s_i by one bit.

$Bit\_Cost_{s_i}$: cost reduction achieved by decreasing the width of s_i by one bit.

The signal that represents the largest *Cost_Error* value is selected and its fractional part is narrowed by one bit. Afterwards, the Err_Penalt, the Bit_Cost, and the *Cost_Error* of all signals and the error slack value are updated, accordingly. The $Bit\_Cost_{s_i}$ term is slightly different from the marginal bit cost, $Ucost_{s_i}$, defined in the first phase. The $Bit\_Cost_{s_i}$ is calculated based on the dynamic word-length values during the refinement process, while the $Ucost_{s_i}$ is constant and calculated based on UFB allocation. The conditional terms in the error model must be particularly considered in these calculations mainly because they can create dependencies between the signals. In other words, in the presence of conditional terms, a change in FWL of a signal can affect the error penalty and bit cost of the others. Only the signals whose $Err\_Penalty$ does not exceed the error slack are eligible for width reduction and are therefore considered in this process. By reducing the error slack in each iteration, the number of eligible signals shrinks.

The algorithm continues until no eligible signal remains. The $Err\_Penalty_{s_i}$ is obtained by measuring the difference between the output error values before and after narrowing the s_i by one bit. In multi-output circuits, the differences are accumulated. Figure 5.6 presents the PSA algorithm with a single output. A multi-output version can be obtained with a few minor modifications.

Table 5.1 Fractional width refinement of the example circuit of Figure 5.2

iter.		a	b	c	d	*Err_Slack*
1	**FWL**	**13**	12	14	14	9.1×10^{-4}
	Cost_Error	**34816**	32768	17408	17408	
2	**FWL**	12	12	**14**	14	4.2×10^{-4}
	Cost_Error	NE*	NE	**17408**	17408	
3	**FWL**	12	12	13	**14**	3.9×10^{-4}
	Cost_Error	NE*	NE	8704	**17408**	
4	**FWL**	12	12	**13**	13	3.6×10^{-4}
	Cost_Error	NE	NE	**8704**	8704	
…	**…**	…	…	…	…	…
10	**FWL**	12	12	11	11	0
	Cost_Error	NE	NE	NE	NE	

*NE: not eligible for fractional width reduction because of error violation

Table 5.1 demonstrates the first four and the last iteration of this algorithm applied to the example circuit. According to these results, the final selected FWL s for the signals are

$$FWLs = \{FWL_a, FWL_b, FWL_c, FWL_d, FWL_e, FWL_z\} = \{12,12,11,11,11,8\} \tag{5.19}$$

5.3.5 Accelerated Tree-Based Search Algorithm (TBSA)

The second FWL selection algorithm we introduce, named TBSA, is based on an accelerated tree-based search. This algorithm also starts with the *marginal bit cost* calculation in the same way as the progressive algorithm. The main process is carried out in a recursive loop. In each iteration, the FWL of one signal is determined in a fractional format based on the cost proportion according to the following equation:

$$\widehat{FWL}_{S_i}^k = \log_2 \frac{Ucost_{s_i} \times Allowed\_Error_i^k}{C_{s_i} \times \sum_{j=i}^{N} Ucost_{s_j}} \tag{5.21}$$

/* $FWL_{s_i}^0$: Primary fractional widths */

/* E_{req}: Requested output error */

Calculate primary output error E_0 using primary fractional widths

$Err\_slack = E_{req} - E_0$; //error slack calculation

for all signals s_i

Calculate the $Err\_Penalty_{s_i}$ //Error penalty of decreasing the //fractional width of s_i by one bit

Calculate the $Bit\_Cost_{s_i}$ //Cost reduction of decreasing the //width of s_i by one bit.

$Cost\_Error_{s_i} = Bit\_Cost_{s_i} / Err\_Penalty_{s_i}$

end for;

$stop = 0$;

while $stop == 0$

$stop = 1$;

$Best\_Cost\_Error = 0$; //keep track of the best $Cost\_Error$

for (all signals s_i)

if ($Err\_Penalty_{s_i} \leq Err\_slack$)

$stop = 0$; //There is at least one eligible signal for //width reduction.

if ($Cost\_Error_{s_i} > Best\_Cost\_Error$)

//better than the best previously obtained signal for width //reduction

$index\_best = i$; //keep track of the index of the best //found signal

$Best\_Cost\_Error = Cost\_Error_{s_i}$;

end if;

end if;

end for;

if (stop==0) // At least one eligible signal has been found

//decreasing the fractional width of the best found signal

$FWL_{s_{index\_best}} = FWL_{s_{index\_best}} - 1;$

$Err\_slack = Err\_slack - Err\_Penalty_{s_{index\_best}};$

for (all signals s_i)

Update $Err\_Penalty_{s_i}$, $Bit\_Cost_{s_i}$ and $Cost\_Error_{s_i}$.

end for;

end if;

end while;

Figure 5.6 The fractional width refinement algorithm PSA.

The S_i, $Ucost_{s_i}$ and C_{s_i}terms are defined in Section 5.3.4. The $Allowed\_Error$ term represents the total amount of error that is allowed to be introduced by the unprocessed signals.

Since the fractional width is an integer, the calculated $\widehat{FWL}$ must be rounded to one of the adjacent integer values. The decision to round up or down has cascading impacts on other signals mainly because they lead to different $Allowed\_Error$ values. So, this decision can make two possible states for each signal. The algorithm considers these two states as two nodes in a binary decision tree.

The algorithm starts with a single node in the first iteration where two children nodes are generated based on two possible decisions for the first signal. During the second iteration, the second signal is processed in each of the nodes, separately.

The algorithm keeps track of the $Allowed\_Error$ value in all nodes using the following calculations:

$$\begin{cases} Allowed\_Error_{i+1}^{2k-1} = Allowed\_Error_i^k - C_{s_i} \times 2^{\left\lfloor \widehat{FWL}_{s_i}^k \right\rfloor} \\ Allowed\_Error_{i+1}^{2k} = Allowed\_Error_i^k - C_{s_i} \times 2^{\left\lceil \widehat{FWL}_{s_i}^k \right\rceil} \end{cases} \tag{5.22}$$

where

$$Allowed\_Error_1^1 = req\_error.$$

The k term stands for the node number in one level of the tree. So, the $Allowed\_Error_i^k$ represents the maximum allowed error in k^{th} node of the i^{th} level of the tree. We have also employed a technique to accelerate the algorithm by eliminating unpromising paths in each level of the tree. The following pruning rule is used to identify unpromising paths.

Pruning Rule: In each level of the binary FWL decision tree, the node A is unpromising if a node B exists in the same level that offers larger or equal $Allowed\_Error$ with lower cost. The cost value is calculated based on the known FWL of the processed signals and UFB value for unprocessed ones. Figure 5.7 illustrates the pseudo-code of this algorithm with a single output. A multi-output version can be obtained with a few minor modifications. Figure 5.8 shows the first four levels of the decision tree obtained by applying the algorithm to the example circuit of Figure 5.2. The $Node_{12}$ in this figure has been eliminated since its $Allowed\_Error$ has reached zero. This means that the error introduced by the two determined signals has already reached the maximum allowed error and no other error gap remains for the rest of the signals. According to the pruning rule, $Node_{43}$ is unpromising since the $Node_{41}$ outperforms it in both error and cost results.

Finally, this algorithm selects the following FWLs:

$$FWLs = \{FWL_a, FWL_b, FWL_c, FWL_d, FWL_e, FWL_z\} = \{13,11,11,11,11,8\}. \quad (5.23)$$

This solution offers almost the same cost as the one achieved by the PSA. However, for larger applications, we will see in Section 5.4 that the TBSA considerably outperforms the PSA in terms of hardware cost.

5.3.6 Time complexty of the PSA and TBSA algorithms

Time complexity is one of the main criteria to compare WLO techniques. This criterion is particularly important for large designs. This section studies the time complexity of the described algorithms. A brief analysis reveals that the fractional width refinement step takes up most of the time of the PSA algorithm.

The main part of this process is an undetermined-bound loop to select the best candidate signal for width reduction. This loop lasts until the allowed output error bound is violated. The maximum possible number of iterations in this loop can be formalized analytically as follows.

/* $FWL_{s_i}^0$: Primary fractional widths */

/* E_{req}: Requested output error */

Calculate primary output error E_0 using primary fractional widths

$Err\_slack = E_{req} - E_0$; //error slack calculation

/* E_{req}: Requested output error */

/* N: Number of signals */

/* $Ucost_{s_i}$: The marginal bit cost of signal s_i */

/* C_{s_i}: Constant coefficients of $2^{-FWL_{s_i}}$ term in output error equation*/

/* $Cost\_Array(j)$: The cost of the j[th] nod in current level of the tree */

/* Processing one level of the decision tree and generating the next-level nodes. This recursive function is called for the in the main body as *FWL_select* (E_{req}, *UFB_Array, UFB_Cost, 1)*. *UFB_Array* is a one dimensional array with *N* elements and all elements are equal to UFB. The *UFB_Cost* is the cost of the UFB solution */

Function $Best\_FWL = FWL\_select\ (Allowed\_Error, FWL\_Array, Cost\_Array, ind)$

/* ind: the index of the signal that is being processed in this iteration (level of the tree)*/

if ind==N+1 //have all signals been processed ?

return (the array with lowest cost) // by examining the $Cost\_Array$ elements

end if;

Calculating the $C_{s_{index}}$coefficient;

for (i from 1 to $Node\_Num$) //$Node\_Num$: number of nodes in this level of the tree

if (Allowed_Error(i) ≤ 0) // This node has already violated

continue; // the requested error threshold

end if;

Calculating $\widehat{FWL}(i) = \log_2 \frac{Ucost_{s_i}(\mathrm{i}) \times \mathrm{Allowed\_Error(i)}}{C_{s_{\mathrm{ind}}} \times \sum_{\mathrm{j=ind}}^{\mathrm{N}} Ucost_{s_j}(\mathrm{i})}$

// Constructing the nodes of the next level of the decision tree

$FB\_Array\_next(2 \times i - 1) = FWL\_Array(i)$; //Initializing

$FWL\_Array\_next(2 \times i) = FWL\_Array(i)$;

// Updating the FWL of the signal corresponding to this level

$FWL\_Array\_next(2 \times i - 1)(ind) = \lfloor \widehat{FWL}(i) \rfloor$; //round down

$FWL\_Array\_next(2 \times i)(ind) = \lceil \widehat{FWL}(i) \rceil$;

// The $Allowed\_Error$ array for the next iteration

$All\_Error(2 \times i - 1) = Allowed\_Error(i) - C_{s_{ind}} \times 2^{\lfloor \widehat{FWL}_{S_i}^k \rfloor}$

$All\_Error(2 \times i) = Allowed\_Error(i) - C_{s_{ind}} \times 2^{\lfloor \widehat{FWL}_{S_i}^k \rfloor}$

Updating the $Cost\_Array(2 \times i - 1)$ and $Cost\_Array(2 \times i)$ by calculating the cost of $FWL\_Array\_next(2 \times i - 1)$ and $FWL\_Array\_next(2 \times i)$, respectively

end for;

//Acceleration by removing the unpromising nodes

for (i from 1 to $2 \times Node\_Num$)

for (k from 1 to $2 \times Node\_Num$)

if ($All\_Error(k) > All\_Error(i)$)

if ($Cost\_Array(k) \leq Cost\_Array(i)$)

Remove i'th node from FWL_Array_next and

All_Error and $Cost\_Array$

end if;

end if;

end for;

end for;

//Calling the next iteration (to process the next level of the tree)

$ind = ind + 1$;

$Best\_FWl = FWL\_select\ (All\_Error, FWL\_Array\_next, Cost\_Array, ind)$

end function;

Figure 5.7 The accelerated tree-based search algorithm (TBSA).

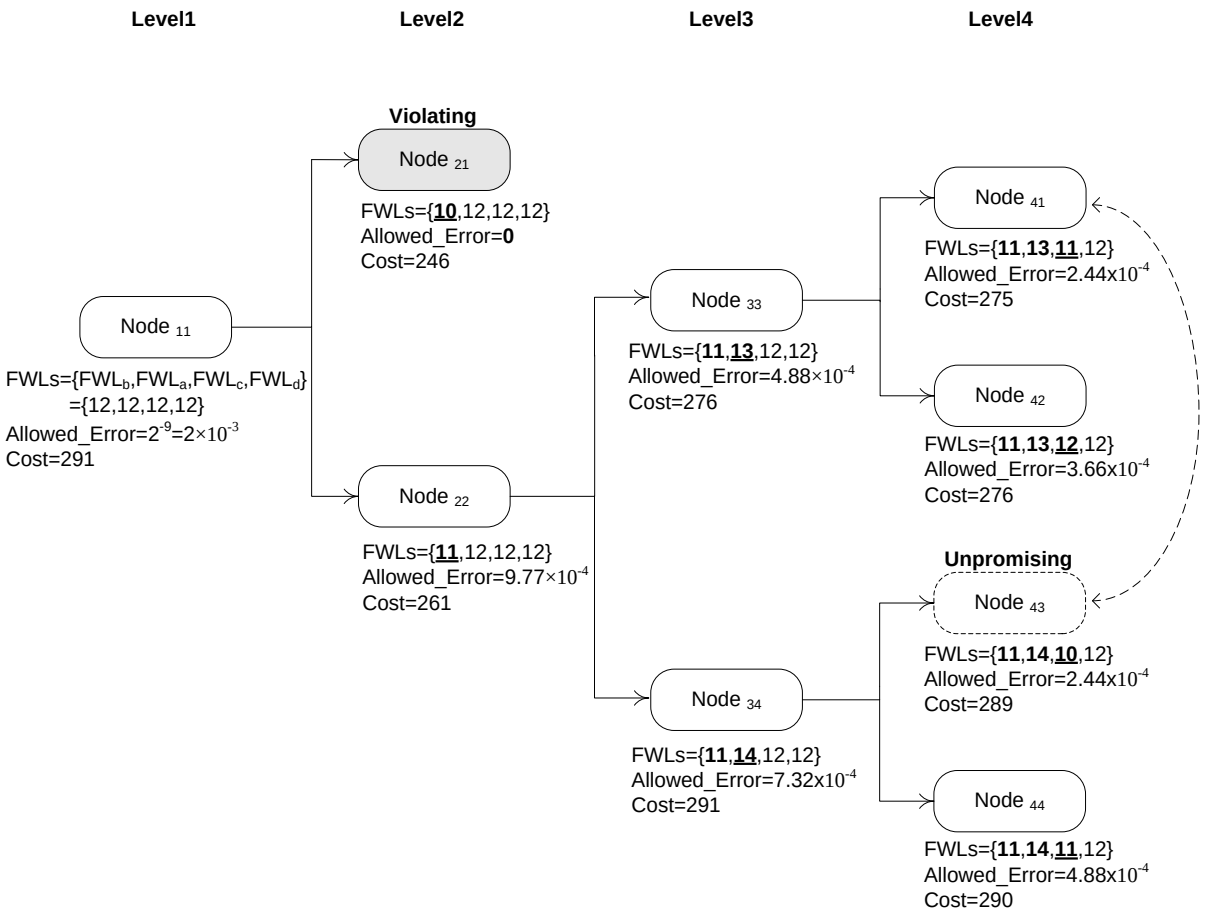


Figure 5.8 First four levels of the decision tree in processing Figure 5.2 circuit with TBSA.

Let us assume that there are N signals in the design and the initial width calculated for each signal according to Equation 5.16 (Section 5.3.4) is represented by FWL_i^0, where $i \in \{1,2,..N\}$. The maximum possible error slack caused by the rounding down operator in initial width calculations is

$$max\_error\_slack < \sum_{i=1}^{N} C_{s_i} \times 2^{-FWL_i^0-1}. \tag{5.24}$$

Decreasing the width of the signal S_i by one bit results in reducing the max_error_slack by $C_{s_i} \times 2^{-FWL_i^0-1}$. Examining different possibilities shows that the largest number of iterations is needed when there is exactly one signal S_k with a large $C_{s_k} \times 2^{-FWL_k^0}$ and $N-1$ signals with equal and small $C_{s_i} \times 2^{-FWl_i^0}$ values, while the latter are selected for width reduction, successive-

ly. In such a case, the total error slack can be dominated by the large value introduced by the S_k initial width calculation, while this large gap is filled by small compensating values at each iteration of the refinement algorithm. We denote the expressions $C_{S_k} \times 2^{-FWL_k^0-1}$ as $Err1$ and $C_{S_i} \times 2^{-FWL_i^0-1}$ of the other $N-1$ signals as $Err2$. The maximum error slack inequality can be rewritten as:

$$max\_error\_slack < Err1 + (N-1) \times Err2 \tag{5.25}$$

In the worst case, during the first $N-1$ iterations of the refinement algorithm, the minimum value of $(N-1)Err2$ is compensated. During the next $N-1$ iterations, the minimum value of $2(N-1)Err2$ is compensated since the FWL of all signals, except the S_k, have been decreased by one bit. So, the termination condition can be formulated as follows:

$$Err1 + (N-1)Err2 < \left(2^{iter/_{N-1}} - 1\right) \times (N-1)Err2 \tag{5.26}$$

$$\Rightarrow \frac{Err1}{(N-1)Err2} + 2 < 2^{iter/_{N-1}}$$

$$\Rightarrow \text{iter} > \log_2\left(\frac{Err1}{(N-1)Err2} + 2\right) \times (N-1)$$

where $iter$ stands for the iteration number.

Taking into account the N-times loop inside each iteration of the algorithm (Figure 5.6) and above calculations, the PSA algorithm can be completed in $O(N^2 \log m - N^2 \log Nl)$ time, where N is the number of signals and m and l are the largest and smallest single-signal quantization errors, respectively.

The basic time complexity of the TBSA algorithm is $O(2^N)$, since all nodes in the *N*-level decision tree must be processed in this algorithm. However, our experiments demonstrate that the employed acceleration technique and error violation condition can significantly reduce the complexity by pruning ineffective nodes. Table 5.2 illustrates the amount of acceleration achieved in the four experiments. For the tested cases, there is no apparent relationship between the design size and time complexity. 8-bit precision is requested for the outputs in these experiments.

Table 5.2 Number of nodes eliminated from the TBSA using the acceleration technique

	No. of eliminated nodes			
	Fig. 5.2	**Degree-4 Polynomial**	**Degree-6 Polynomial**	**Degree-8 Polynomial**
Total nodes	31	16K	1049K	67109K
Accel. Rule	2	4K	397K	21499K
Error Viol.	14	5K	224K	14332K
Total Red.	16 (52%)	9K (50%)	~57%	~55%

5.4 Experimental results and comparisons

This section evaluates the efficiency of the two proposed algorithms (PSA and TBSA) and the simplification technique using eight case studies. The efficiency was measured in terms of time complexity and hardware cost. The presented optimization framework in Section 5.2, containing newly introduced algorithms and techniques, was implemented in MATLAB. The Osborne method [52] and an exhaustive search method were also implemented to enable comparison. All experiments were performed on an Intel i7 3-GHz PC with 16 GB DDR3 RAM. Designs were synthesized with ISE 13.1 for a Xilinx Virtex 6 FPGA.

5.4.1 Case studies

The case studies include some commonly used transforms and operators in the signal and image processing domains. Table 5.3 lists these applications specifying the number of elementary operations and signals in each of them as a measure of the complexity. Four of these applications are the same as those used by Lee *et al.* [38]: RGB-to-YCrCb, B_Spline, 2×2 matrix multiplication, and 8×8 Discrete Cosine Transform (DCT).

For the polynomial approximations, we have implemented the general architecture, contrary to Lee *et al.* [38] who customized it to the $y = log(1 + x)$ function. In other words, we consider the polynomial coefficients as input signals. This provides a more general architecture at the expense of higher hardware complexity. The eighth-order Infinite Impulse Response (IIR) filter is implemented via four second-order cells. A radix-2 decimation in frequency structure is used for

the 64-point Fast Fourier Transform (FFT). All signals are complex in the FFT. The real and imaginary parts are considered as independent variables in our measurements. As a result, the number of operators is very large in this case study. The hardware costs of each case study were measured from VHDL descriptions implemented in Virtex 6 FPGAs. Multiplications and divisions by power of two constants were implemented by shifts. Common intermediate signals were shared in the designs to improve hardware efficiency.

5.4.2 Coding limitations

Like for most of the other analytical methods, the applicability of the proposed techniques and algorithms is limited to feed-forward datapaths. Covering control flow statements is also challenging in analytical WLO. Our approach supports *if* statements by performing the range and precision analysis based on the taken and untaken codes, separately. The results are then merged by selecting the largest achieved IWL and FWL value for each variable.

Loops with static bounds are also supported using an implicit unrolling mechanism. Supporting loops with dynamic bounds is still an open problem [38, 52, 61].

Table 5.3 Complexity of the case studies

Case Study	Add/Sub	Mult.	No. Signals
Degree-4 Poly	4	4	13
Degree-6 Poly	6	6	19
Degree-8 Poly	8	8	25
Degree-10 Poly	10	10	31
RGB-to-YCrCb	6	7	19
B-Spline	7	4	15
2×2 Matrix Mult.	18	7	29
DCT 8×8	32	32	55
FIR32	31	32	94
IIR8	16	20	56
FFT64	1028	520	1548

5.4.3 Preliminary simplification technique

We start the experiments by studying the impact of the preliminary simplification technique, described in Section 5.4.2, on the efficiency. Table 5.4 shows the number of signals that are eliminated from the FWL selection process using the simplification technique. The results reveal that the simplification technique reduces the complexity of the problem by an average of 20.3%. Figure 5.9 and Figure 5.10 compare the optimization run-time and the hardware cost in terms of area obtained for the selected applications before and after the simplification technique. 8-bit precision is requested for the outputs in these experiments. Figure 5.9 illustrates the optimization time ratio of the original applications to the corresponding simplified ones using the preliminary simplification technique. The optimization time improvement of the polynomial approximation has not been shown in Figure 5.9, because no variable can be simplified in this case study. Therefore, the corresponding improvement value is equal to zero. Due to the overlong run-time of the TBSA algorithm in the FIR32 and FFT64 benchmarks, no corresponding results are shown in Figure 5.9.

The results show a significant improvement in optimization run-time and a slight improvement in hardware area when using the simplification technique. The simplification technique has been applied to all benchmarks before being used for evaluation of the optimization algorithms in subsequent sections.

Table 5.4 Complexity reduction using the preliminary simplification technique

Case Study	Removed signals	Reduction [%]
Degree-6 Poly	0	0
RGB-to-YCrCb	6	31.5
B-Spline	3	13.8
2×2 Matrix Mult.	4	20
DCT 8×8	16	29.1
FIR32	30	31.9
IIR8	11	19.6
FFT64	260	16.7

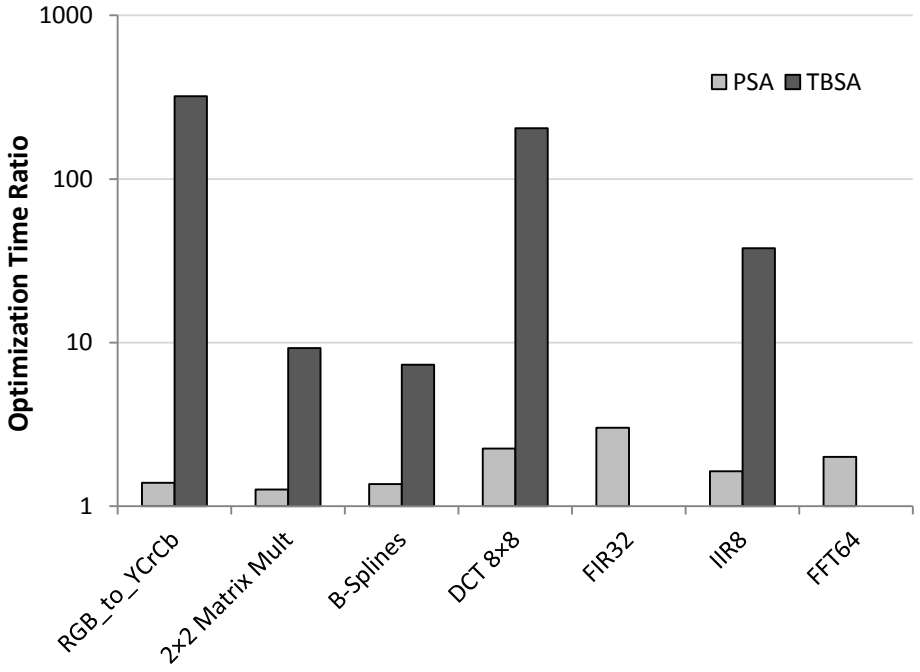


Figure 5.9 Optimization time ratio of the original applications to the corresponding simplified ones.

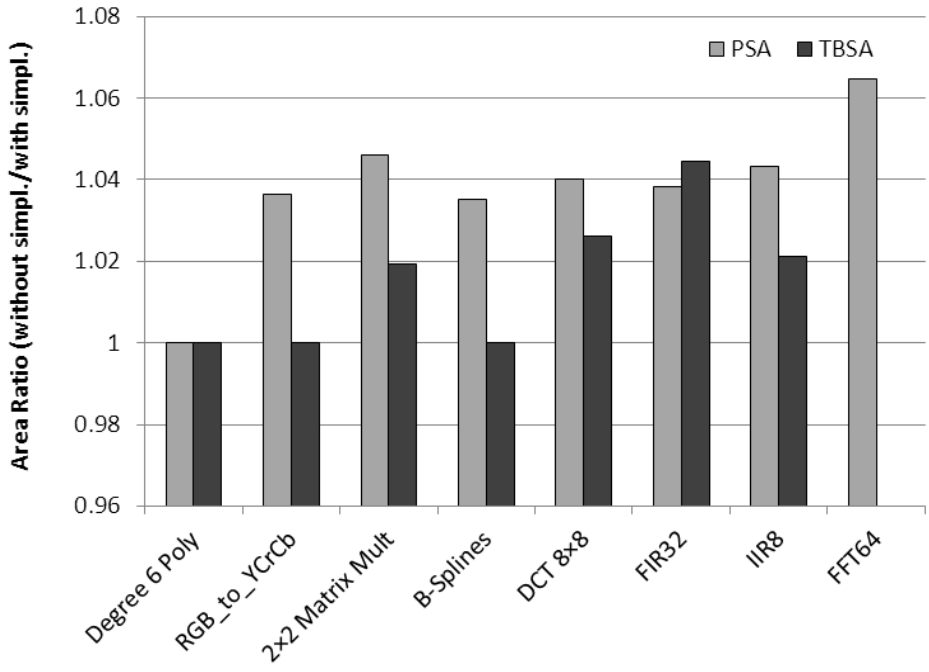


Figure 5.10 Hardware cost ratio without/with simplification technique.

5.4.4 Comparing with UFB and Osborne's method

In this section we compare PSA and TBSA with the naïve UFB allocation and Osborne's method [52], which is based on similar principles as our method.

Osborne's method begins with the calculation of the UFB value. Then, the UFB is added to an initial constant to give the starting value for the second step during which the bit-widths are reduced via an iterative procedure. In each iteration, one signal is selected for FWL reduction. The hardware cost was considered as the main criterion to select a signal for width reduction, such that the reduction that causes the largest decrease in cost will be selected first. If several signals offer the largest cost reduction, the one that increases the error by the smallest amount is selected.

This process continues until the error bound requirement is broken. To be faithful, we have used the original error modeling approach in our implementation of Osborne's method. A significant problem with this method is the lack of a specific way to determine the initial constant. Hence, this value must be selected arbitrarily. A small initial constant may lead to ignoring valuable parts of the search space, while a large one causes unnecessarily long run-time.

Table 5.5 lists the results obtained for the different case studies. Figure 5.11 shows the saved area by the four algorithms with respect to the UFB approach. Figure 5.12 illustrates the relative optimization time for Osborne's and Menard's methods over the PSA.

The results demonstrate that PSA provides superior run-time over the Osborne algorithm by a factor between 5× and 27×, with an average of 16.1×. PSA also provides solutions with a reduction in area by an average of 10.9% and 3.9% compared to the UFB and Osborne methods, respectively.

In hardware area, TBSA outperforms the UFB, Osborne's and progressive search approaches by an average of 13.1%, 6.6% and 2.9%, respectively. There are no results in Table 5.5 for TBSA for the FFT64 case study because of the complexity of that case, with over 1250 variables even after applying the simplification technique. For problems of that size, where the processing is excessive, although for the moment PSA offers a good compromise, an eventual solution with TBSA would be to restrict the variables to those associated with costly components such as multipliers.

Table 5.5 Efficiency of the proposed algorithms and previous works

Case Study	Prec. (bits)	Optimization Time (s)					Area (slices)					
		Osborne [52]	Menard's Greedy [66]	Menard's Greedy +Tabu [66]	This work PSA	This work TBSA	UFB	Osborne [52]	Menard's Greedy [66]	Menard's Greedy +Tabu [66]	This work PSA	This work TBSA
Degree 6 Polynomial	8	5.24	1.3	4.3	0.9	16.7	2008	1841	1831	1806	1819	1773
	16	5.4	1.4	4.6	1.0	16.2	4858	4423	4318	4290	4311	4217
RGB to YCrCb	8	5.5	1.1	3.2	0.7	12	539	511	511	491	494	484
	16	5.6	1.1	3.0	0.7	12.4	912	861	844	823	821	807
2×2 Matrix Multiplication	8	31.9	3.4	37.6	2.7	154.1	1857	1717	1679	1637	1627	1597
	16	31.1	3.6	39.1	2.6	154	3920	3657	3577	3492	3510	3444
B-Splines	8	3.6	0.7	2.9	0.6	4.5	711	643	637	626	626	613
	16	3.6	0.7	2.9	0.5	4.5	1277	1159	1121	1102	1109	1102
DCT 8×8	8	131.1	9.1	187.2	5.9	2375.5	4917	4761	4627	4611	4613	4422
	16	137.6	9.7	191.8	6.1	2262.7	8328	8130	7903	7833	7871	7610
FIR32	8	404.6	31.9	701.8	16.3	7754.1	7912	7217	7092	6898	7027	6505
	16	439.9	33.7	716.0	17.1	7633.6	15390	14141	13551	13224	13411	13002
IIR8	8	216.9	15.5	176.6	9.6	2078.8	5791	5410	5209	5004	5122	4918
	16	197.1	14.2	160.0	9.0	2122.1	10722	9893	9440	9292	9388	9002
FFT64	8	150786.1	19449.1	263677.9	5721.4	-	89234	81225	76922	75005	76177	-
	16	156005.6	18234.4	259071.1	5699.3	-	160511	144961	140545	134933	138109	-

5.4.5 Comparing with Menard et al.

We now compare the proposed algorithms with one of the most recent works presented by Menard *et al.* [66]. Menard's work proposes a combination of HLS and WLO. The target applica-

tion is first given to the HLS process to divide the operations of the same type into a number of groups. The operations that lie within the same group are organized for execution in a single hardware operator. The objective of this process is to share the hardware resources among operations in order to reduce implementation costs.

The output of the grouping process is given to the WLO algorithm to optimize the bit-width assigned to each group and its operator. The grouping function significantly reduces the number of variables which are considered in the WLO process.

Menard *et al.*'s WLO approach is composed of a greedy search followed by a Tabu search algorithm. The greedy search, which is significantly faster, finds an initial solution. Afterwards, the Tabu search refines the solution quality by exploring the neighborhood area of the initial point.

The HLS technique is out of scope of this thesis. To make a fair comparison between our work and Menard *et al.*'s, we have only considered the WLO approach. It was employed in our framework omitting the grouping process. Table 5.5, Figure 5.11 and Figure 5.12 compare the results obtained by Menard's approach with our proposed algorithms. These results demonstrate that Menard's method lies between the PSA and the TBSA in the time-area design space. The area cost results of Menard's method are up to 2.3% better than the PSA method's, while its optimization time is greater than the PSA's by a factor between 4× and 45×. Figure 5.11 and Figure 5.12 represent the average results achieved for 8- and 16-bit output precisions.

The greedy search in Menard's method is similar to our PSA algorithm in some ways. However, the PSA benefits from a more effective selection strategy that enables additional savings before termination of the algorithm. This is achieved by using a list of non-eligible variables that prevents early termination of the bit-width reduction process. Moreover, the PSA takes advantage of a smarter starting point selection scheme, as presented in Section 5.4.4. Having a shorter distance between the starting point and the final solution considerably reduces the execution time. The results in Table 5.5 show that the PSA finds better solutions than Menard's greedy search algorithm in less time. Using the Tabu search, Menard's method compensates for the poor quality of the greedy search solution at the expense of longer execution time.

5.4.6 Comparing with Exhaustive Search

As a last step, the FWL selection algorithms were compared with an exhaustive search method that is expected to generate optimal cost results.

In our exhaustive search algorithm, all FWLs were initially set to their minimum possible value. For the signal s_i, this value is calculated by assuming infinite precision for any other signal that gives following equation:

$$FB_{S_i}^0 = \left\lfloor \log_2 \frac{Req\_error}{C_{s_i}} \right\rfloor \tag{5.27}$$

Any fractional bit-width smaller than $FB_{S_i}^0$ causes a violation of the requested error bound regardless of the precision of the other signals. A maximum bit-width value is also selected to limit the upper bound FWL search process. This value must be large enough to ensure coverage of the valuable solution areas. Then, the widths are incremented iteratively, such that all possible combinations within the obtained bounds are examined. After evaluating all combinations, the one that respects the error requirement with the smallest cost is selected as the final solution.

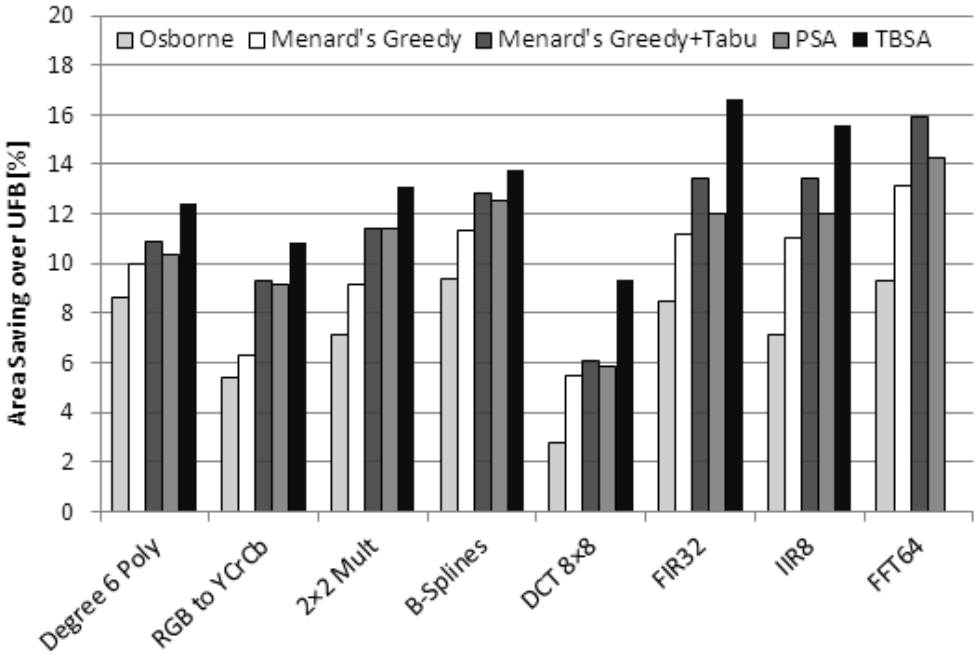


Figure 5.11 Area cost reduction over the UFB approach.

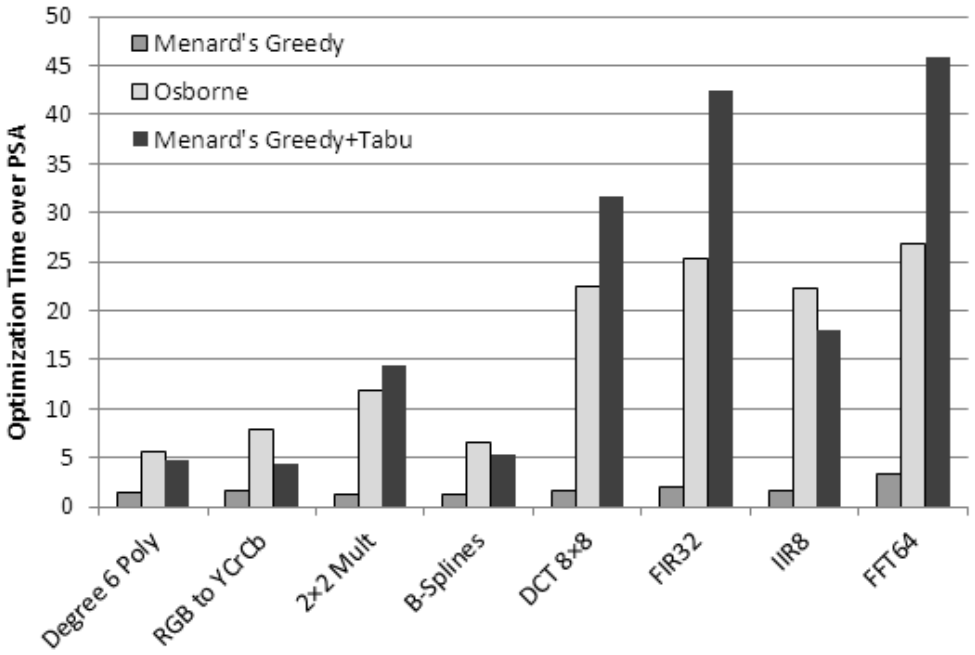


Figure 5.12 Normalized optimization time relative to PSA method.

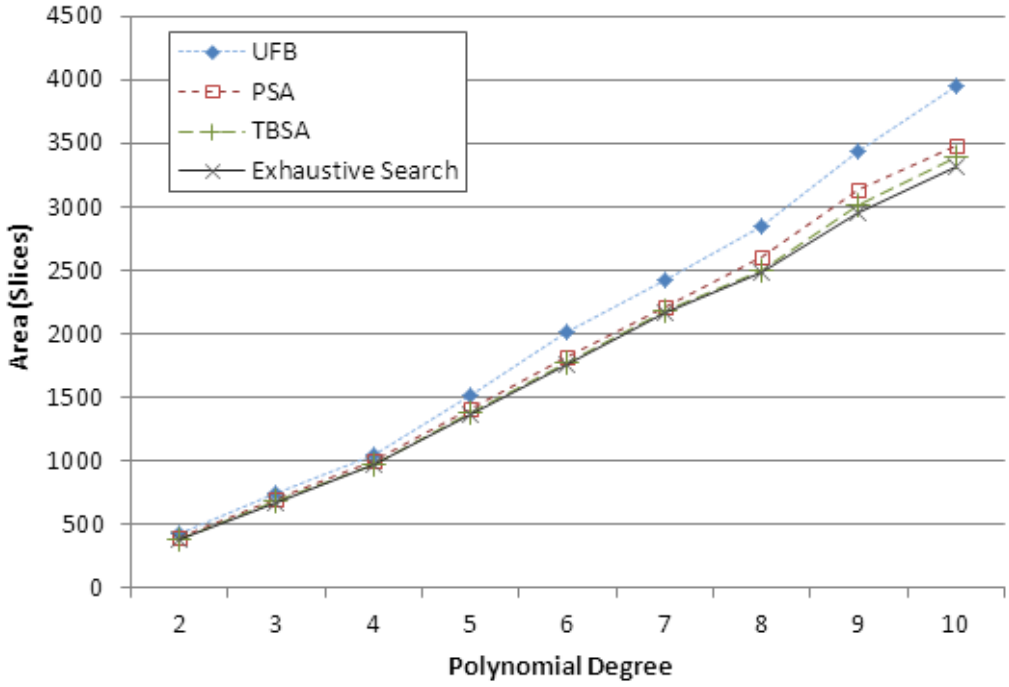


Figure 5.13 Comparing the proposed algorithms with the exhaustive search method.

The experiments show that the exhaustive search run-time takes from a few hours for B-spline to a few days for the DCT case study. Figure 5.13 compares the area obtained by the proposed algorithms, UFB solution, and the described exhaustive search approach for polynomial approximation. 8-bit precision is requested for the outputs in these experiments .The results show that the area efficiency of PSA and TBSA is within 3.8% and 0.9% of the exhaustive search, respectively.

5.5 Conclusion

In this chapter, we introduced a new word length optimization methodology for fixed-point designs. It uses affine arithmetic for both range and precision analyses. Moreover, we have introduced a simplification technique and two new semi-analytical algorithms for FWL allocation. A novel word-length optimization framework was developed that implements our introduced approaches.

A set of eight case studies was used to evaluate the proposed methods. The experimental results show that our simplification technique reduces the complexity of the fractional bit-width selection problem by an average of 20.3%. Moreover, the results demonstrate considerable improvements in optimization run-time and hardware area when using the proposed FWL selections algorithms.

CHAPTER 6 FIXED-POINT PROCESSOR CUSTOMIZATION

In this chapter, we introduce a new processor customization method for fixed-point applications. This method is composed of customization of processor word-length based on a dedicated WLO method and customization of the functional unit architecture regarding the selected word-length configuration.

The contents of this chapter are largely extracted from our paper "Accuracy-aware processor customization for fixed-point applications," submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* in 2014 [88].

6.1 Introduction

Application-specific processor customization is one of the promising trends to promote the efficiency of processor-based designs. This trend includes various state of the art research areas such as instruction-set customization in extensible processors [26], micro-architectural customization in parameterizable processors [9] and application-specific processor design offered in ADLs [30]. In this chapter, we introduce a novel processor customization approach which explores a new dimension in application-specific micro-architectural optimization targeting fixed-point applications. This new dimension is the word-length of the datapath that is normally fixed in microprocessors. In integer computation, the minimum required word-length of the datapath is determined by the maximum range of the data elements in the applications. Customizing the word-length of the processor to this value can potentially improve the efficiency of the processor depending on the application.

In fixed-point computation, the problem of word-length allocation is considerably more complex due to the introduction of new factors. Each fixed-point value is comprised of integer and fractional parts. The IWL of each signal should be long enough to guarantee overflow/underflow avoidance. This lower-bound requirement can be found by range analysis using various existing analytical [38, 39] and simulation-based techniques [89].

Determining the FWL is inherently more complex. The FWL of each signal determines the quantization error which is introduced due to the finite word-length representation of that signal. This quantization error can propagate through the subsequent levels of the circuit and eventually show

up at the outputs as inaccuracy in the computations. Various analytical [38, 51] and simulation-based techniques [58, 59] were introduced in the literature to model the finite-precision error of a circuit based on the word-length allocation of its signals. Reducing the IWL and FWL of signals can significantly improve the efficiency of the implementation. In fixed-point designs, WLO adjusts the IWL and FWL allocated to each signal considering the overflow/underflow hazards and the accuracy requirements. The efficiency can be measured from the hardware area, power consumption, performance, or a combination of them based on the design objectives.

There are basically two word-length allocation approaches. The traditional UWL allocation approach offers a single word-length for all variables. The MWL approach allows different word-lengths for different variables. Figure 6.1 compares a conventional processor with customized processors via UWL and MWL approaches. The important customizable elements in the proposed method include the word-length of the register-files, pipeline buffers and functional units, and the number of words in each register-file.

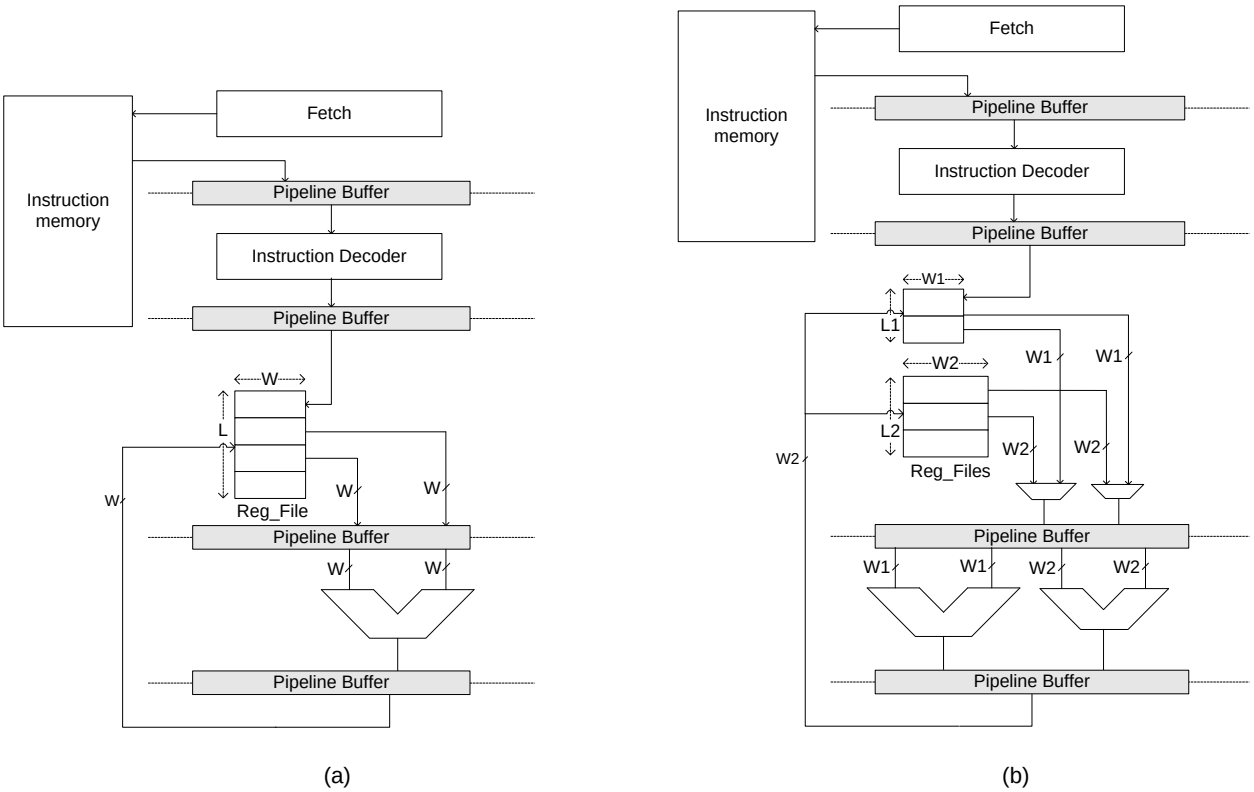


Figure 6.1 Comparing customized processors via UWL and MWL approaches.

WLO has been extensively studied in numerous researches for ASIC designs. However, to the best of our knowledge, no related research work has been considered in custom processor design. The main contribution of this chapter is to present the first work of literature that investigates WLO for application-specific customization of microprocessors by exploring architectural trade-offs. This is illustrated in Figure 6.1a and Figure 6.1b.

More precisely, this work proposes a method for accuracy-guaranteed optimization of the processor word-length for fixed-point applications. This method aims to enhance the efficiency of the processor architecture through application-specific customization, while meeting the precision requirements.

The architecture of the functional unit is the other target that the proposed method aims to optimize in parallel with the WLO. Complex arithmetic functions such as multiplication commonly have a significant impact on area usage and performance of the processors. There are usually various possible architectures to realize an arithmetic functions in hardware. The efficiency of using a specific architecture in a design depends on the application and the word-length allocation. The proposed method customizes the number of hardware operators and architecture of each one regarding the word-length allocation solutions. A dedicated design space exploration algorithm is developed for the architecture selection.

Finite-precision error modeling is an essential part of any WLO method. Such a model formulates finite precision error at the outputs in terms of the FWL of the inputs and the intermediate signals. Given the error model, the uniform word-length can be easily calculated in the UWL approach. However, the MWL optimization is an NP-hard problem that is normally solved by heuristic search algorithms. The proposed method explores both UWL and MWL approaches in its optimization algorithm.

The rest of this chapter is organized as follows. Section 6.2 describes the proposed methodology. The design flow of the proposed method is described in Section 6.3. Section 6.4 presents the optimization algorithm which is used in this chapter. Section 6.5 gives experimental results and comparisons, and Section 6.6 concludes the chapter.

6.2 Proposed methodology

In this section, we present our proposed methodology. This methodology aims to generate application-specific customized processors for fixed-point applications. New hardware elements and components are the target of customization in this work. In the following subsections, we will first present the objectives of the proposed method in processor customization. Then we will illustrate these objectives in detail using an example.

6.2.1 Methodology objectives

Our proposed method is based on the pursuit of three objectives.

The first objective of the proposed method is to improve the efficiency of the processor architecture by customizing the word-length of the data elements and consequently the calculations. The word-length has direct impact on the area cost and speed of various parts of a processor. Any reduction in the word-length of a processor can lead to a significant improvement of overall efficiency. The proposed approach supports the MWL scheme that allows using multiple datatypes with different word-lengths in the customized processor. Although using multiple word-lengths may increase the complexity of the hardware realization and the optimization problem, it also increases the potential of reaching more efficient solutions.

The second objective is to improve the efficiency of the processor architecture by customizing the depths of the register-files. One register-file is dedicated to each selected word-length. The minimum depth required for each register-file depends on the application and the word-length allocation. In addition to the area usage, the depths of the register-files also impact the bitwidth required to index the registers in the instruction and consequently the bitwidth of the instructions and related memory units.

The third objective is to improve the efficiency of the design by customizing the architecture of the functional units. There are usually many possible architectures to realize a unique operator. The latency, area cost and throughput of the operators may vary for the selected architecture. In this work, one of the issues considered in the optimization algorithm is to select the best architecture to implement the hardware operator. The number of hardware operators needed to be implemented in the execution stage depends on the number of data types and the operators which are

required for each data type. Both of these factors are determined through the design space exploration in the proposed optimization algorithm. The architecture of each operator can also be optimized based on application requirements and the word-length allocation.

6.2.2 Illustration of the objectives

In this subsection, we illustrate how the three objectives are met by our optimization method using a design example. Figure 6.2 illustrates an example circuit that performs the following calculations:

$$z1 = b \times a^5,$$
$$z2 = c \times d + c \times b,$$
$$z3 = d \times e + d \times c \tag{6.1}$$

The input values are assumed to be in the range of [0,128). Using the approach described in Chapter 4, the error model of the example circuit is calculated as follows:

$$2^{-FWL_{z1_q}-1} \geq 5 \times 2^{-FWL_{a_q}+34} + 2^{-FWL_{b_q}+34} + 2^{28} \times \delta_{s1} + 2^{21} \times \delta_{s2} + 2^{14} \times \delta_{s3} + 2^{7} \times \delta_{s4}$$

$$2^{-FWL_{z2_q}-1} \geq 2^{-FWL_{c_q}+7} + 2^{-FWL_{b_q}+6} + 2^{-FWL_{d_q}+6} + \delta_{s5} + \delta_{s6}$$

$$2^{-FWL_{z3_q}-1} \geq 2^{-FWL_{d_q}+8} + 2^{-FWL_{c_q}+7} + 2^{-FWL_{e_q}+7} + \delta_{s6} + \delta_{s7} \tag{6.2}$$

where $\delta_{s1} = \begin{cases} 2^{-FWL_{s1_q}-1}\varepsilon_{s1}, & FWL_{s1_q} < FWL_{a_q} + FWL_{b_q}, \\ 0, & otherwise \end{cases}$

$\delta_{s2} = \begin{cases} 2^{-FWL_{s2_q}-1}\varepsilon_{s2}, & FWL_{s2_q} < FWL_{a_q} + FWL_{s1_q}, \\ 0, & otherwise \end{cases}$

$\delta_{s3} = \begin{cases} 2^{-FWL_{s3_q}-1}\varepsilon_{s3}, & FWL_{s3_q} < FWL_{a_q} + FWL_{s2_q}, \\ 0, & otherwise \end{cases}$

$\delta_{s4} = \begin{cases} 2^{-FWL_{s4_q}-1}\varepsilon_{s4}, & FWL_{s4_q} < FWL_{a_q} + FWL_{s3_q}, \\ 0, & otherwise \end{cases}$

$\delta_{s5} = \begin{cases} 2^{-FWL_{s5_q}-1}\varepsilon_{s5}, & FWL_{s5_q} < FWL_{b_q} + FWL_{c_q}, \\ 0, & otherwise \end{cases}$

$$\delta_{s6} = \begin{cases} 2^{-FWL_{s6_q}-1}\varepsilon_{s6}, & FWL_{s6_q} < FWL_{c_q} + FWL_{d_q}, \\ 0, & otherwise \end{cases}$$

$$\delta_{s7} = \begin{cases} 2^{-FWL_{s7_q}-1}\varepsilon_{ds7}, & FWL_{s7_q} < FWL_{d_q} + FWL_{e_q} \\ 0, & otherwise \end{cases}$$

The value FWL_{s_q} represents the fractional word-length allocated to the signal *s*.

We start with the word-length allocation, which is the first objective of the proposed method. Analyzing the error model reveals that the five multiplications that lie within the path of calculating *z1* must be much wider than the other operations in the application when the same accuracy is requested for all outputs. For instance, assume that 8-bit fractional accuracy is requested for all outputs. Solving the error inequalities in (6.2) shows that at least 54 bits are required for *a*, *b*, *s1*, *s2*, *s3*, and *s4* to meet the requested accuracy at output *z1* and to provide the necessary integer word-length for the signals. This value is obtained by assuming a uniform word length for all signals to simplify the calculations and by solving the first inequality of (6.2). However, 26 fractional bits are enough for the *b*, *c*, *d*, *e*, *s5*, *s6* and *s7* signals to guarantee 8-bit accuracy at output *z2* and *z3*. The integer word-length of the signals is taken into account in these calculations. Fourteen bits are required for the integer part of signals *s1*, *s5*, *s6*, and *s7* while the integer part of signals *s2*, *s3*, and *s4* should be at least 21, 28 and 35 bits, respectively.

A single datatype processor that can calculate the above example must have a datapath that is at least 54 bits wide. This is, in fact, the realization of the UWL approach in the processor domain. The register-file, inter-stage signal routes and the hardware operators in the functional unit, including the multiplier, must also be able to handle this bitwidth. However, we know that 26-bits are enough for three of the multiplications and 7 signals that only participate in the calculation of the *z1* and *z2* outputs. Converting the processor to a double-word-length architecture with 26- and 54- bit datatypes can reduce the hardware area of the register-files. Moreover, adding a separate 26-bit multiplier to the processor may enable the design to use more diverse types of multiplier architectures. All these decisions need an effective exploration of a large search space. The mentioned word-length allocation possibilities show the importance of the first objective.

The word-length allocation also has a direct impact on the minimum necessary depth of the register-files as the second objective. In microprocessors, more than one variable can be mapped to the

same physical register if there is no time overlap between their living times in the application code. Register allocation algorithms are normally used in compilers to map the registers to the application variables. The register allocation result determines the minimum required depth for the register-files. In the proposed method, variables are divided into different word-lengths in MWL word-length allocation solutions. A change in word-length allocation can change effective factors in the register allocation and can eventually change the minimum required depths of the register-file. For instance, if a 54-bit word-length is selected for variable *a* and a 26-bit length is selected for variable *e*, then variable *a* cannot be mapped to the 26-bit register file. These constraints are considered in the modified register allocation algorithm of the proposed optimization algorithm described in Section 6.4. Hence, the register allocation and the required depth of the register-files depend on the word-length allocation. The register-file depth must be evaluated and considered during the word-length selection. In the proposed algorithm, the word-length allocation is performed through design space exploration. The fitness of each candidate solution is evaluated by measuring the effective factors on the efficiency including the depth of the register-files.

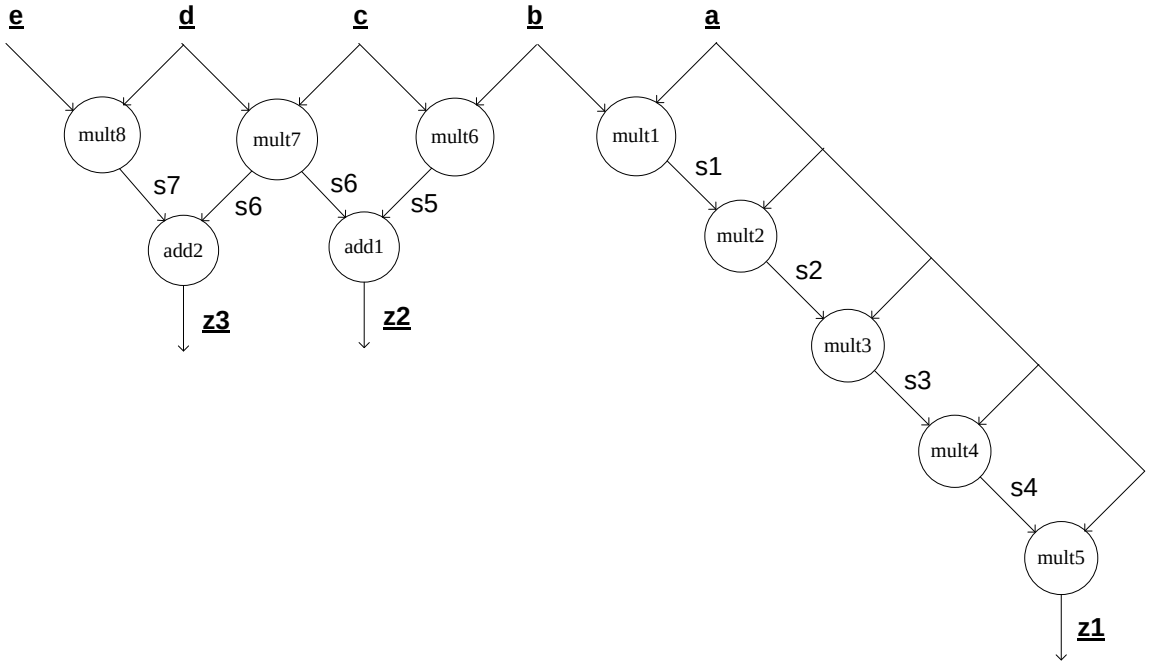


Figure 6.2 Example circuit

The third objective focuses on the architecture of complex functions. In this work, we limit our explorations to the multipliers as the most widely used complex function in order to simplify the presentation. Hardware multipliers are used in most modern embedded processors since multiplication is a basic operation in most DSP and image processing applications. So, the proposed method searches for the most efficient multiplier architectures to integrate into the customized processor. However, all proposed procedures can be easily extended to cover other complex functions such as division and logarithm. A multiplier can be implemented by various architectures with different efficiency characteristics. In this work, we consider three well known architectures: the basic combinational multiplier, the multi-cycle shift and add multiplier and the pipelined shift and add multiplier.

The multi-cycle and the pipeline architectures divide the multiplier datapath into *n* fragments, where *n* is less than or equal to the bitwidth of the operands. When *n* is equal to the width of the operands, the multi-cycle architecture becomes a fully-serial shift and add multiplier. The case *n=1* corresponds to the combinational architecture. For multi-cycle multipliers, a larger *n* means smaller hardware area and latency due to less calculation in each clock at the expense of more clock cycles to complete a calculation. The three mentioned architectures are examined for any requested multiplier unit. Let *n* be the number of stages in multi-cycle and pipeline architectures, then *n* is the other configurable parameter that is explored by the optimization algorithm. The hardware cost of each candidate architecture is measured separately. The results are given to the optimization algorithm as tables to facilitate the overall cost estimation.

The operations of shorter bitwidth can always be calculated in wider hardware operators. For example, an 8-bit addition can be calculated using a 10-bit adder. Adders/subtractors and logical operators are small and fast enough to be always implemented as combinational circuits. Therefore, a single, wide hardware operator is sufficient for operations on both narrow and wide data types. However, for complex operators such as multipliers, a wider bitwidth implies a considerably larger hardware area and/or longer latency. Hence, having multiple hardware operators of different sizes for the complex functions may improve the overall efficiency. First, this allows faster and more efficient processing for the shorter data types. Second, the wider operators are freed from processing short data types and can thus be implemented more efficiently as multi-cycle operations.

The design space of the multiplier architecture depends on the word-length allocation. Therefore, exploration for the best multiplier architecture should be carried out based on a known word-length allocation. To show how different function architectures may be used in a customized processor, we give two possible solutions for the multiplier of the Figure 6.2 example. A 2-datatype configuration is selected for bitwidth configuration, with signals *a*, *b*, *s1*, *s2*, *s3*, and *s4* assigned to the 54-bit type and the rest to the 26-bit type.

Figure 6.3 illustrates these two solutions. The first solution uses a single hardware multiplier unit that is 54 bits wide. Figure 6.3a represents the time scheduling of the instructions using this solution. In all experiments, the throughput of the execution is supposed to have the highest priority.

Therefore, the search space of the function architectures is limited to those that do not require extra clock cycles. Different multiplications must be calculated in successive clock cycles as shown in Figure 6.3a. If a single multiplier is used in the processor, the selected architecture must support a throughput of one multiplication per clock cycle to avoid pipeline stalls.

Either combinational or pipelined multiplier architectures can support this throughput. The time scheduling of Figure 6.3a is based on using a combinational architecture for the processor multiplier. A 54-bit combinational multiplier is large and slow enough to become the critical path in most embedded processors.

The second solution uses a 54-bit multiplier for *mult1* to *mult5* and a 26-bit multiplier for *mult6* to *mult8*. The time schedule of this solution is presented in Figure 6.3b. Using separate multipliers combined with appropriate ordering of the instructions allows 2-cycle gaps for completion of the calculation in each multiplier. This extra cycle flexibility enables the use of a 2-cycle architecture for the multipliers. Moving from combinational to 2-cycle architecture causes significant reduction of hardware resources and latency. The latter leads to an increase of the clock frequency when the multiplier is in the critical path.

This example demonstrates how multiple hardware units for a complex operator can improve the efficiency of the design. The optimization algorithm should explore the possible architectures to find the optimal solution that achieves the highest efficiency.

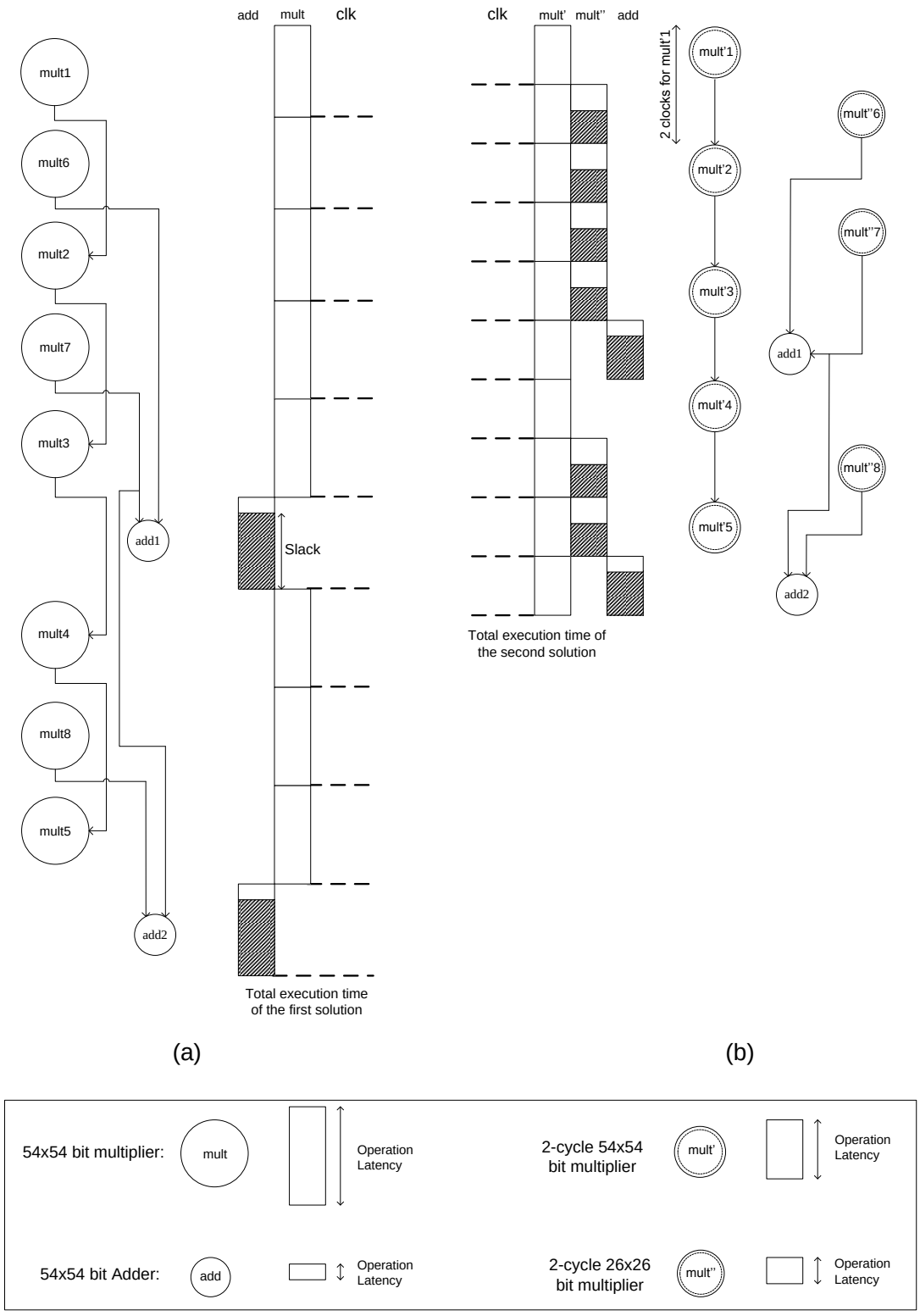


Figure 6.3 Time scheduling of two possible solutions with (a) single multiplier (b) double multipliers.

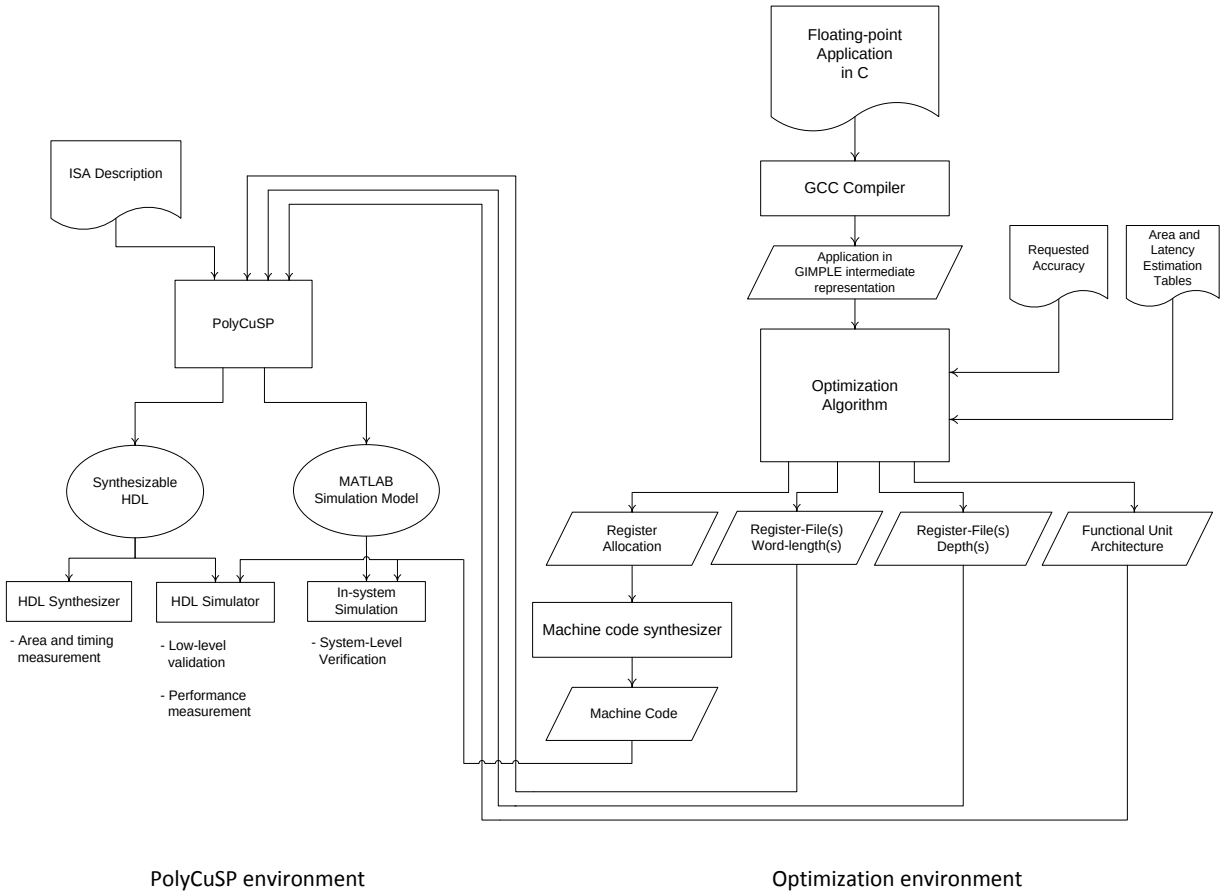


Figure 6.4 The design flow in the proposed method.

6.3 Design flow integration

We propose a complete design flow that incorporates the optimization and the custom processor generation environments. The optimization environment is mostly developed in MATLAB. This environment consists of the optimization algorithm, which will be described in Section 6.4, and some peripheral units such as cost estimation tables. The processor generation environment creates the customized architecture based on the selected solution in the optimization algorithm. This section discusses the design flow in more detail.

6.3.1 Overview

The design flow is illustrated in Figure 6.4. The optimization process starts by providing a C application by the designer. All fractional variables are represented in floating-point in this input

code. In the first step, the input C code is converted into Gimple, which is an intermediate representation of the GCC compiler, similar to a simplified C code. This conversion significantly facilitates code interpretation. The Gimple code is then given to the optimization algorithm to find a solution to address the customization objectives described in Section 6.2. The selected solution is finally given to PolyCuSP to generate a corresponding architecture in RT-level VHDL.

The area and latency estimation tables are given to the optimization algorithm to be used for the fitness evaluation of the candidate solutions. These tables contain the cost estimations of different possible values of the customizable elements. For example, one important table gives the estimated area and maximum achievable frequency for different word-lengths of the datapath regardless of the architecture of the other parts of functional unit. The datapath word-length is equal to the word-length of the longest datapath. The machine code synthesizer is like a compiler backend that converts the intermediate representation of the application into the machine code. The register allocation solution is generated by the optimization algorithm and given to this unit. The generated machine code is sent to PolyCuSP to store it in the instruction memory of the output processor.

6.3.2 Custom processor design environment

The proposed method is eventually evaluated by implementing the selected solutions on a real processor. In this chapter, we use PolyCuSP to generate the processor architecture based on the selected customization solution. As described in Chapter 3, PolyCuSP provides a fast custom processor design method that largely outperforms the traditional method of manual processor design from scratch. In addition, PolyCuSP supports broad flexibility in microarchitectural and instruction-set customizations [76].

The proposed customizations are realized through specific parameters in the processor description. PolyCuSP support the flexibility in configurable parameters that is required for the proposed customizations. Other parts of the architecture are fixed in our experiments. For example, the MIPS II ISA is used as the fixed instruction-set that is supported by the output processors and the machine code synthesizer also works based on this instruction set. The PolyCuSP environment offers automatic generation of synthesizable RT-level VHDL code, assembler, and MATLAB simulation model from the given processor description.

6.4 Optimization algorithm

This section presents a detailed description of the proposed optimization algorithm. The optimization algorithm contains two nested genetic algorithm (GA) procedures for word-length allocation (Objective 1) and a dedicated search algorithm to find the best architecture for the functional units (Objective 3). Furthermore, the algorithm has a fitness evaluation routine that determines the fitness of the complete candidate solution using cost and performance estimation. Register allocation is a necessary part of the fitness evaluation that also addresses the second objective of the proposed method. The flow chart of this algorithm is illustrated in Figure 6.5. The following definitions facilitate elaboration of each part of the algorithm:

N: $number\ of\ wordlengths\ or\ number\ of\ datatypes\ in\ the\ processor$.

W_i: $the\ wordlength\ of\ the\ i'th\ datatype\ \ 0 < i < N$.

S: $number\ of\ variables$

V_j: $the\ datatype\ used\ for\ j'th\ variable\ \ 0 < j < S, \qquad V_j \epsilon \{1..N\}$

The different parts of the proposed algorithms are described in the following subsections.

6.4.1 The first- and second-level genetic algorithms

The algorithm starts by selecting *N* in the main loop that contains all other parts of the optimization algorithm. Then, the algorithm performs *N* iterations (one for each possible number of wordlengths). In our experiments, we selected $N = 4$ as the upper-bound, in order to provide a wide range of flexibility to the optimization algorithm. In each iteration, the design space is explored for *N* datatypes. After completion of the fourth round, the best solution is sent to PolyCuSP to generate the corresponding processor architecture. The solution also includes the register allocation information that is used to generate the machine code of the application from the given GIMPLE code.

The two nested GAs are named GA1 and GA2. On the first level, GA1 searches for the best datatype assignment scheme (V_j). Each chromosome is a string composed of *S* elements, while each element determines the candidate datatype for the corresponding variable in that chromosome. The first generation is generated randomly. After generating a population, the second GA procedure starts.

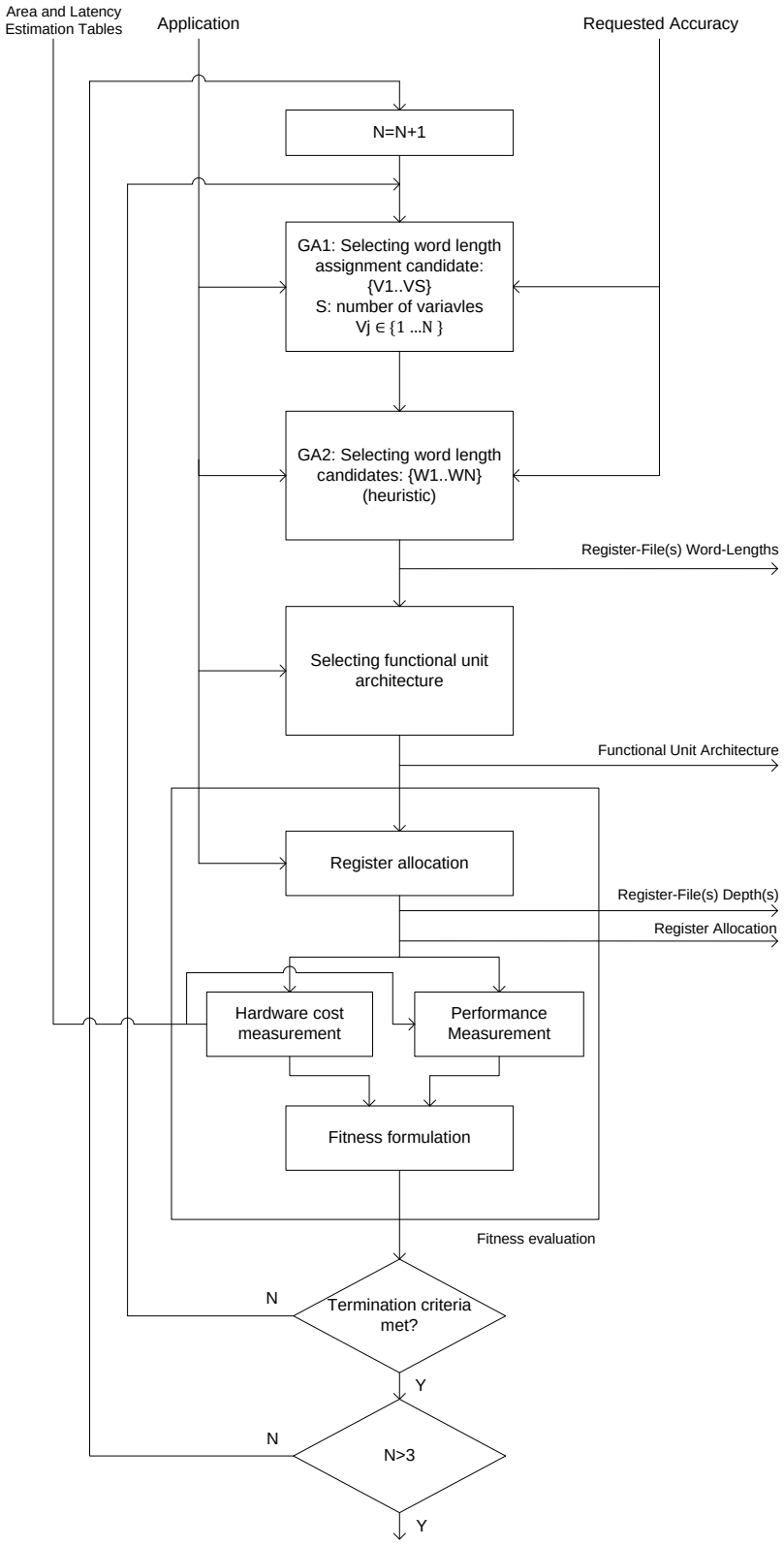


Figure 6.5 The flow chart of the optimization algorithm

On the second-level, GA2 finds the optimal word-length(s) for the datatypes (W_i) for each chromosome of the given population determined by the first GA. In this phase, each chromosome is composed of $N' \leq N$ word-length values for the N datatypes. N' is less than N when two or more word-lengths in a chromosome have the same values. To limit the search space, the lower bound and upper bound values of each datatype are determined analytically before the start of the GA2.

Only the values between the lower-bound and the upper-bound are searched in the GA2. The lower-bound value of each datatype is calculated by assuming infinite word-length for all other data types and solving the output error model with this assumption. The lower-bound values of all datatypes are calculated first. Then the upper-bound word-length of each datatype is calculated by assuming the lower bound word-length for all other datatypes and the output error model is solved based on this assumption. The upper- and lower-bound word-lengths are highly dependent on the datatype assignment scheme. This means that these boundary values may change for each chromosome of the given population determined by GA1. Hence, for each chromosome, the boundary values must be calculated first.

Then, GA2 explores the range between the boundary values of all datatypes to find the best word-length combination for all datatypes. Each chromosome in GA2 represents a candidate solution for the word-length of the datatypes, i.e., a candidate W vector. With W and V vectors as the candidates of the first- and second-level chromosomes, the word-length of each variable in the application can be identified as

$$WL_j = W_{V_j}, \tag{6.3}$$

where WL_j is the word-length assigned to the j^{th} variable. So, the combination of one chromosome of GA1 and one chromosome of GA2 gives a complete candidate solution for word-length (or datatype) allocation.

6.4.2 Architecture selection

The next step consists of finding the best architecture for each candidate word-length allocation solution. This algorithm can be extended to any complex function. In this work we concentrate on exploring the best architectures for the multipliers.

The developed algorithm, which is presented in Figure 6.6, starts by listing the word-lengths of all required multiplications in the given application. For example, in a 2-word-lengths solution with *w1* and *w2*, at most three multipliers with *w1*×*w1*, *w1*×*w2* and *w2*×*w2* widths may be required. This architecture selection algorithm is executed for each candidate of GA2. The aforementioned list of required multiplication widths depends on the word-length allocation and the application. The narrower multiply operations can always be calculated on a wider multiplier. Therefore, this list may differ from the hardware multiplier units that are in the processor.

In the next step, the widest required multiplication operation is identified using the application and the word-length allocation information given by the GAs. There should be a multiplier unit that can handle this widest multiplication. So, the width of the first multiply unit is known. This multiplier is enough to handle all multiplications in the application but it does not necessarily represent an optimal solution. Adding a shorter multiplier unit may open new areas in the design space of the multipliers. This design space expansion can enable the use of more efficient architectures just like the example shown in Figure 6.3. Therefore, the algorithm evaluates addition of shorter multiplications based on the list of required multiplication widths in the application using an exhaustive search.

For each candidate solution, each multiplication operation is assigned to the shortest multiplier that exists in that solution. For example, assume that there are two datatypes with *w1* and *w2* widths while *w1*>*w2*. If multiplications with all three possible widths exist in the application, then a multiplier with *w1*×*w1* width must exist in the processor. If a solution suggests adding a *w1*×*w2* multiplier, then *w2*×*w2* multiplications can be assigned to the *w1*×*w2* multiplier.

Different possible architectures are examined for each multiplier of a candidate solution. We only consider combinational, pipeline and multi-cycle multipliers, in this work. The architectures that do not impose further stall cycle in the execution time are evaluated. The total area cost and maximum latency of each solution is estimated. Comparing these estimations, the search algorithm selects the best solution. The designer-defined priorities of area cost and performance are considered in the selection criteria.

Store all multiplication word-lengths present in the application into vector *M_width.*

Find the widest multiplication word-length and add it to the list of candidate word-length for the hardware multipliers in vector *HM_width*

for all possible combinations of the elements in the *M_width* (excluding the widest one found in Step2)

{

Add the selected combination of the bitwidths to the *HM_width*

Find the best architectures for the multipliers with word-lengths that exist in *HM_width* using exhaustive search (each element in the *HM_width* means a hardware multiplier with the same word-length).

Delete all element of the *HM_width* except the widest one (clear the vector for the next candidate)

}

Figure 6.6 The multiplier selection algorithm

6.4.3 Fitness function

Now, the algorithm needs to measure the fitness of this candidate solution. The fitness of a solution is the efficiency that it achieves when implemented. The GA requires a single fitness value for each chromosome in order to identify more promising areas in the search space. Therefore, the efficiency must be evaluated as a single value that represents a combination of all important efficiency metrics. The weight of different efficiency metrics in the fitness calculation can be adjusted by the designers based on their priorities.

In this work, a formula is developed to calculate a single value to represent the fitness. This formula takes into account four metrics:

1. L: Word-length of the largest datatype. This parameter defines the bitwidth of the major parts of the datapath.
2. M_{reg}: Amount of on-chip memory resources used for the register-files.
3. A_{mult}: Estimation of the hardware area used for the multipliers.
4. $Freq$: Estimation of the maximum frequency that can be used by the processor.

The fitness formula is as follows:

$$fitness = \frac{C_F \times Freq}{C_L \times L + C_M \times M_{reg} + C_A \times A_{mult}} \tag{6.4}$$

C_F, C_L, C_M, and C_A are weighting factors. The fitness value is returned to GA2, where it is stored for processing. When the fitness evaluation of all chromosomes of one population is completed, the fitness values are compared to identify the best found candidate solutions for GA2, i.e., the best *W* vector. These best found chromosomes are used to generate a fraction of the chromosomes in the next generation. Crossover and mutation techniques, which are well-known methods in GA, are used to generate new chromosomes. This process continues until the termination criterion of GA2 is met. In this work, the termination criterion for both GAs is when no additional improvement is achieved between two consecutive iterations. The current GA2 searched for the best *W* vector for one chromosome given by the first-level. After termination of this GA2, the best fitness value returns to the first-level. This value is considered as the fitness value of the corresponding chromosome in the GA2. For each chromosome in one population of GA1, GA2 is executed from beginning to the end. When the fitness of all chromosomes in one generation is evaluated, the next generation is generated using the best found candidates.

This process continues until the termination criterion of GA2 is met. Then, the optimization algorithm terminates and the combination of the best found solutions in the two GA levels (i.e., the best found *V* vector and the best *W* vector found for it) is returned as the final solution found by the optimization algorithm.

The depth of the register-files is one of the effective factors on the efficiency of the processor architecture. As mentioned in Section 6.2.1, the second objective of the proposed method is to improve the efficiency of the processor architecture by customizing this factor. The depth of the

register files is determined based on the word-length allocation and the requirements of the application. The minimum required depth for the register-files can be found via register allocation of the target application.

A modified graph coloring algorithm is developed for register allocation. This algorithm assigns a physical register to each variable of the application and also determines the minimum required depth of each register file. This algorithm considers the word-length assigned to the variables in the register assignment process. Hence, the register allocation and the depths of the register-files depend on the candidate word-length allocation solution given by GA1 and GA2. The modified graph coloring algorithm avoids allocation of a register R_i to the variable W_j when the bitwidth of R_i is narrower than W_j. For example, assume that a GA2 candidate suggests two register files with bitwidths of w_1 and w_2 while $w_1 > w_2$. If w_1-bit datatype is selected for this variable by GA1, then the register allocation algorithm cannot assign a w_2-bit register to this variable. Yet, the algorithm allows allocation of a longer register to a shorter variable. In this case, the simplest approach to address the register allocation is to use an independent graph coloring algorithm for each datatype. However, the proposed approach is smarter and yields more efficient results by allowing the use of a register resource for a shorter variable. The register allocation algorithm is a part of the fitness evaluation that determines the area cost of the register-files for a given word-length allocation candidate.

6.5 Experimental results

In addition to the Figure 6.2 example, three well-known benchmark applications were used to evaluate the proposed customization method. The first application is a 126-tap linear-phase low-pass FIR filter with direct form II transposed structure. The second application is an RGB-to-YCrCb converter which is implemented based on the form suggested by the ITU [90]. The third application is an IIR filter of fourth-order [65]. The Virtex IV FPGA family was selected as the target platform for implementations and evaluations. ISE 13.2 was used for synthesis and measurements. We gave higher priority to the area usage in the fitness formulation in our experiments. Regarding Equation (6.4), this means that C_F was set to a much smaller value compared to C_L, C_M, and C_A. The reason is to facilitate demonstration of the effectiveness of the proposed method by focusing on area minimization as the main optimization goal.

Table 6.1 Hardware cost and performance results of the benchmark applications

Application	Type of multipliers	Word-lengths [bits]	# of multipliers	Delay of the multipliers [clock cycles]	Area		Freq. [MHz] (Imp)
					LUTs (Imp*)	FFs (Imp)	
GPP32**	Single_WL	32	1	1	2143	1868	58
FIR	Single_WL	16	1	1	1419 (0%)	630 (0%)	94 (0%)
	Combinational	13,18	1	1	1288 (10%)	585 (7.7)	73 (-22.4%)
	Pipeline	13,18	1	5	1302 (9.0%)	645 (-2.3%)	141 (50.0%)
	Multi-cycle	13,18	2	3,1	1213 (17.0%)	619 (1.8%)	122 (29.8%)
RGB-YCrCb	Single_WL	18	1	1	1577 (0%)	346 (0%)	76 (0%)
	Combinational	14,18	1	1	1413 (11.6%)	307 (12.7%)	74 (2.6%)
	Pipeline	14,18	2	5,4	1508 (4.6%)	388 (-11.1%)	133 (75%)
	Multi-cycle	14,18	2	2,2	1373 (14.9%)	339 (2.0%)	116 (52.6%)
IIR	Single_WL	12	1	1	1123 (0%)	377 (0%)	142 (0%)
	Combinational	9,12	1	1	1034 (8.6%)	331 (13.9%)	136 (-4.2%)
	Pipeline	9,12	2	3,2	1090 (3.0%)	364 (3.6%)	191 (34.5%)
	Multi-cycle	9,12	2	2,2	977 (14.9%)	352 (7.1%)	177 (24.6%)
Fig. 6.2 example	Single_WL	54	1	1	2933 (0%)	672 (0%)	16 (0%)
	Combinational	54,26	1	1	2755 (6.5%)	531 (26.5%)	16 (0%)
	Pipeline	54,26	2	2,1	2771 (5.8%)	588 (14.3%)	28 (75%)
	Multi-cycle	54,26	2	2,2	2609 (12.4%)	612 (9.8%)	24 (62.5%)

* Imp: % improvement

** GPP32: A conventional 32-bit non-customized processor with 32-word register-file for comparison

Table 6.1 illustrates the hardware cost and the performance results of single-datatype and double-datatype solutions for the four applications. It also compares the result of employing different multiplier architectures in the MWL approach.

The results demonstrate that moving from a single-datatype to a 2-datatype architecture can significantly improve the area usage and performance of the processor. In these experiments, 8-bit accuracy was requested for the outputs. The results demonstrate how the architecture of complex functions can affect the overall efficiency of the design. We illustrate the best found result by using three different multiplier architectures, separately. New instructions can be fed into the pipeline multiplier in each clock cycle. However, a multi-cycle multiplier does not accept any new input until it completes the last calculation. So, the number of stages in pipeline multipliers can be more than the multi-cycle ones. The pipeline multipliers gave better latency results in the experiments. Note that customized single-datatype solutions are used as the reference to measure the improvements. The word-lengths of the processor in these reference designs are adjusted to the value achieved by UWL word-length optimization. Comparing with the existing processors with fixed power of two word-lengths can obviously achieve significant improvements.

Figure 6.7 shows the run-time progress of the optimization algorithm for the three mentioned applications. This figure illustrates how the genetic algorithms converge toward better solutions. The area and the maximum frequency metrics are measured by synthesizing the best found candidate solution of each generation with the ISE tool. The results show that the 2-datatype solutions achieve the best results for the evaluated benchmarks, while the 3- and 4-datatype solutions can reach very close results in most cases. The main reason is that the benchmark applications are not large enough to be able to take advantage of more than two datatypes.

Increasing the number of datatypes in a processor can lead to a reduction of the memory usage in the register-files and some other related area resources. It can also increase the area usage in some cases such as the multiplexers that select among the register-files in operand read stage. Hence, there is a trade-off between the savings that more datatypes can achieve and the overheads that they imply. Figure 6.7 also shows that increasing the number of datatypes results in a longer search time for convergence of the GAs. This is due to the fact that adding a new datatype to the processor significantly expands the search space of the optimization algorithm.

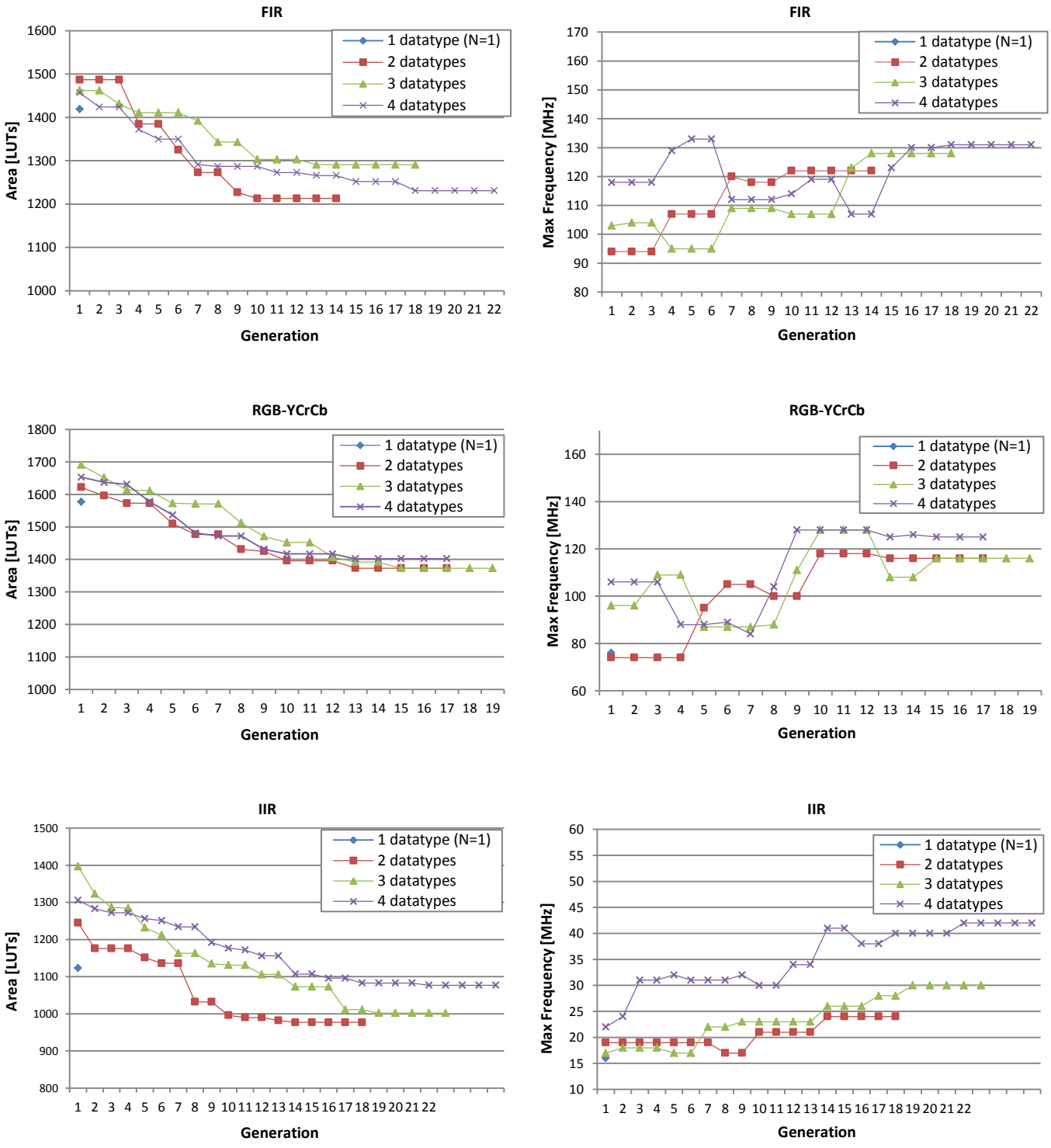


Figure 6.7 Run-time progress of the optimization algorithm in each generation of the GA1.

6.6 Conclusion

We proposed a new processor customization method for fixed-point computations. This method combines the word length optimization with application-specific processor customization. The word-lengths of the supported datatypes, depth of the register-files, and the architecture of the functional unit form the customization targets. A multi-level genetic algorithm and a dedicated fitness evaluation method were developed for the optimization algorithm.

Four benchmark applications were used to evaluate the proposed method. The experimental results show that, in the selected FPGA platform, moving from a single word-length processor to a customized double-word-length processor can reduce the area consumption in terms of the number of LUTs and flip-flops by an average of 14.8% and 5.2%, respectively. The results also show an average of 42.4% improvement in the speed of the processor by this customization. These results demonstrate the effectiveness of the proposed customization method.

CHAPTER 7 CONCLUSION AND FUTURE WORK

7.1 Summary of the work

In this thesis, we introduced new methodologies, techniques and algorithms to improve the hardware realization of fixed-point computations in hardwired circuits and customizable processors. We proposed new methods to improve the efficiency of different analyses required for automatic hardware realization of fixed-point computation.

In the first step, we presented PolyCuSP which is a new processor design environment that is used for fast and easy custom processor generation. PolyCuSP bridges the gap between architecture description languages (ADLs) and extensible soft processors. The basic goal of designing PolyCuSP was to have an environment that supports the required flexibility to realize the new customizations proposed in this thesis and to facilitate design space exploration in a large design area. PolyCuSP offers full flexibility in instruction-set description, while limiting the datapath customization to a predefined set of tunable microarchitectural parameters.

We evaluated and compared the design and customization complexities offered by PolyCuSP with competitive approaches by some experiments. The results demonstrated the efficiency of applying customization techniques in the proposed environment.

In the second step, we introduced an enhanced finite-precision error modeling approach based on affine arithmetic that addresses some shortcomings of existing methods and improves their accuracy. We demonstrated that there is a common hazard in existing affine arithmetic-based error modeling approaches. The hazard is linked to early substitution of the signal terms that emerge in operations such as multiplication and division. We proposed postponed substitution combined with function maximization to address this problem. We also proposed a modification in the error propagation process to enhance the error modeling accuracy. An existing word length optimization method was reproduced to evaluate the efficiency of this modification. The results demonstrated that the proposed modification can significantly improve the accuracy of the error estimation at the expense of a negligible complexity overhead.

In the third step, we presented new WLO solutions for the hardware synthesis of fixed-point computational circuits. These include two fractional word-length selection algorithms and an

acceleration technique. While the first FWL selection algorithm follows a progressive search strategy, the second one uses a tree-shaped search method for fractional width optimization. The algorithms offer two different time-complexity/cost efficiency trade-off areas. The first algorithm has polynomial complexity and achieves comparable results with existing heuristic approaches. The second algorithm has exponential complexity but it achieves near-optimal results compared to the exhaustive search method.

A set of eight case studies was used to evaluate the proposed methods. The experimental results show that our simplification technique significantly reduces the complexity of the fractional bit-width selection problem. Moreover, the results demonstrated considerable improvements in optimization run-time and hardware area by using the proposed FWL selections algorithms.

In the last step, we introduced a new processor customization method based on fixed-point word-length optimization. We proposed a method that for the first time combines word-length optimization with processor customization. The supported datatype word-lengths, the size of register-files and the architecture of the functional units are the main target objectives to be optimized by this method. Accuracy requirements, defined as the worst-case error bound, is the key consideration that must be met by any solution. PolyCuSP was used to realize the processor architecture based on the solution found in the proposed optimization algorithm.

Four benchmark applications were used to evaluate the efficiency of the proposed method. The results show that for a specific application, the proposed method can improve the efficiency of the processor architecture via customizing the word-length, functional unit architecture and register-file size.

7.2 Summary of the contributions

This section reviews the contributions of the different parts of the thesis. The main contributions for the PolyCuSP environment, which was introduced in Chapter 3, are:

1. Proposition of a processor design approach that bridges the gap between ADLs and parameterizable processors by combining the former’s instruction-set customization and the latter’s microarchitectural tuning.

2. A design environment called PolyCuSP that realizes the proposed idea. PolyCuSP offers a simplified design and customization process that automates generation of major datapath elements and interconnection signals from the instruction-set description.
3. An original processor description approach that supports definition of multiple register-files, multi-port register-files, and multi-cycle functional units, etc.

These contributions were published in our paper entitled “Customised soft processor design: a compromise between architecture description languages and parameterisable processors," published in *IET Computers & Digital Techniques*, journal, May 2013, [76].

The main contributions in finite-precision error modeling, which were presented in Chapter 4, are:

1. Illustration of a common hazard in existing AA-based error modeling approaches and presentation of a solution to address it.
2. A modified error propagation method which can effectively improve error model accuracy.

These contributions were published in the paper entitled "Finite-precision error modeling using affine arithmetic," presented in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2013 [84].

The main contributions in word-length optimization for hardwired circuit design, which were presented in Chapter 5, are:

1. An analytical method to simplify the precision analysis process.
2. A new polynomial-time semi-analytical algorithm for fractional word-length selection.
3. A new exponential-time semi-analytical algorithm for fractional word-length selection that can reach near-optimal results.

These contributions were published in the paper entitled “Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," published in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, journal, Dec. 2013 [87].

The main contributions in WLO-based processor customization, which were presented in Chapter 6, are:

1. A novel method to utilize WLO for application-specific customization of microprocessors by exploring architectural trade-offs.
2. A multi-objective genetic algorithm (GA) to optimize the datapath word-length allocation.
3. Optimization of functional unit architecture based on the word length information and the application requirements to prove the design efficiency.

These contributions have been submitted to the journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* in a paper entitled "Accuracy-aware processor customization for fixed-point applications," Jan. 2014 [88].

7.3 Future works

Even though the work described in this thesis has presented multiple contributions in the field of hardware design for fixed-point computation. Several improvements could be proposed to the solutions presented and, moreover, many extensions could be provided.

The novel idea of generating customized processors for fixed-point applications introduces a fertile ground for research. This idea, which was mainly presented in Chapter 5, offers a combination of fixed-point optimization and processor customization. We demonstrated how customization of specific elements in processor architecture can improve the overall efficiency. However, the customizable elements are limited in our experiments. Therefore, the range of customizations can be extended to other units.

For instance, we introduced a dedicated search-based routine to find the best architecture for functional units in an optimization algorithm, presented in Chapter 5. We demonstrated how combining this routine with WLO processes can enhance the achievable design efficiency. However, we limited the functional units to multipliers for which we considered three specific architecture options. One possible future work is to extend the list of the functional units whose architectures can be customized in the proposed method. Other complex functions, such as division, exponential and logarithm, are required in many applications. These functions can also be added

to the functional unit selection algorithm. For this purpose, the most potentially efficient hardware architectures for each function must first be developed. The hardware efficiency and accuracy of each candidate architecture should be measured for different word-lengths. This information should be given to the optimization algorithm to be used for the efficiency measurements.

The proposed customizations were applied on a simple pipeline processor architecture. An interesting future work would be to use the same idea for more advanced architectures such as VLIWs. The complexity of the problem may change in the new architectures. For example, using multiple parallel pipelines in VLIW architectures introduces new parameters to the optimization algorithm that enlarges the search space.

The idea of using fixed-point optimization for processor customization was evaluated for the first time in this thesis. For this first work, we used a genetic algorithm, which is a well-known generic heuristic method, as the optimization algorithm. GA is an appropriate choice for the early developments to evaluate the potentials of the idea. Although the capabilities of the GA to find the optimal solution are widely proven, it does not have the best search speed. This makes the optimization process very slow for large designs. A highly useful future work would be to develop a dedicated heuristic algorithm for this problem to reduce the execution time of the optimization algorithm.

We have only considered the area usage and latency as the effective efficiency factors in our development and experiments. In other words, we limited the optimization goal to minimization of the hardware area and latency of the customized processor. However, the power consumption is the other important efficiency factor that can be considered in the optimization. For this purpose, a new procedure could be added to the optimization algorithm to estimate the power consumption of a candidate customization solution. Moreover, the fitness function should be modified so that the estimated power consumption can properly contribute in the fitness calculation.

REFERENCES

[1] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: a fixed-point design and simulation environment," in *Proceedings of Design, Automation and Test in Europe*, 1998, pp. 429-435.

[2] L. Zhang, Y. Zhang, and W. Zhou, "Tradeoff between approximation accuracy and complexity for range analysis using affine arithmetic," *Journal of Signal Processing Systems,* vol. 61, pp. 279-291, 2010.

[3] G. A. Constantinides and G. J. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters,* vol. 15, pp. 137-140, 2002.

[4] K. Karuri, R. Leupers, G. Ascheid, and H. Meyr, "A generic design flow for application specific processor customization through instruction-set extensions (ISEs)," in *International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2009, pp. 204-214.

[5] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *Springer Journal of Reconfigurable Computing: Architectures, Tools and Applications,* pp. 209-220, 2008.

[6] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *International Conference on VLSI Design*, Jan. 2001, pp. 76-81.

[7] J. Turley, "Embedded systems survey: who uses custom chips," *Embedded Systems Programming,* vol. 18, pp. 39-42, 2005.

[8] J. Henkel, "Closing the SoC design gap," *IEEE Journal of Computers,* vol. 36, pp. 119-121, 2003.

[9] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and customization of FPGA-based soft processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, pp. 266-277, 2007.

[10] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *International Symposium on Field Programmable Gate Arrays*, Monterey, California, USA, 2006, pp. 201-210.

[11] R. Dimond, O. Mencer, and W. Luk, "CUSTARD - a customisable threaded FPGA soft processor and tools," in *International Confference on Field Programmable Logic and Applications*, Aug. 2005, pp. 1-6.

[12] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, "Application-specific customization of parameterized FPGA soft-core processors," in *IEEE/ACM International Conference on Computer-Aided Design*, 2006, pp. 261-268.

[13] S. Padmanabhan, R. K. Cytron, R. D. Chamberlain, and J. W. Lockwood, "Automatic application-specific microarchitecture reconfiguration," in *20th International Parallel and Distributed Processing Symposium*, 2006, pp. 1-8.

[14] M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Datapath and ISA customization for soft VLIW processors," in *Proc. Int. Conf. Reconfigurable Computing and FPGAs*, 2006, pp. 1-10.

[15] O. Hebert, I. C. Kraljic, and Y. Savaria, "A method to derive application-specific embedded processing cores," in *Workshop on Hardware/Software Codesign*, 2000, pp. 88-92.

[16] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen, "A flexible DSP core for embedded systems," *IEEE Journal of Design & Test of Computers*, vol. 14, pp. 60-68, 1997.

[17] D. Fischer, J. Teich, M. Thies, and R. Weper, "Efficient architecture/compiler co-exploration for ASIPs," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES'02* 2002, pp. 27-34.

[18] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: an ASIP design environment," in *International Conference on Computer Design*, Sept. 2000, pp. 430-436.

[19] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: generation of high-quality instruction set extensions by iterative improvement," in *Proceedings of Design, Automation and Test in Europe* 2005, pp. 1246-1251

[20] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," in *Conference of Hardware/Software Codesign and System Synthesis*, 2006, pp. 160-165.

[21] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of Design Automation Conference.* , 2003, pp. 256-261.

[22] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Transaction on Design Automation of Electronic Systems*, vol. 7, pp. 605-627, 2002.

[23] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, pp. 1258-1270, 2005.

[24] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 216-228, 2004.

[25] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, pp. 1209-1229, 2006.

[26] R. E. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro,* vol. 20, pp. 60-70, 2000.

[27] J.-H. Yang, *et al.*, "MetaCore: an application specific DSP development system," in *Design Automation Conference*, Apr. 2000, pp. 800-803.

[28] I. D. L. Anderson and M. A. S. Khalid, "SC Build: a computer-aided design tool for design space exploration of embedded central processing unit cores for field-programmable gate arrays," *IET Computers & Digital Techniques,* vol. 3, pp. 24-32, 2009.

[29] A. Chattopadhyay, H. Meyr, and R. Leupers, "LISA: A uniform ADL for embedded processor modelling, implementation and software toolsuite generation," in *Processor Description Languages*, P. Mishra and N. Dutt, Eds., ed: Morgan Kaufmann, 2008, pp. 95-130.

[30] P. Mishra and N. Dutt, "Architecture description languages for programmable embedded systems," *IEE Proceedings of Computers and Digital Techniques,* vol. 152, pp. 285-297, 2005.

[31] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *European Design and Test Conference*, 1995, pp. 503-507.

[32] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *Design, Automation and Test in Europe Conference and Exhibition*, 1999, pp. 485-490.

[33] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven exploration of pipelined embedded processors," in *International Conference on VLSI Design*, 2004, pp. 921-926.

[34] V. Zivojnovic, S. Pees, and H. Meyr, "LISA-machine description language and generic machine model for HW/SW co-design," in *VLSI Signal Processing Workshop*, 1996, pp. 127-136.

[35] J. Kairus, J. Forsten, M. Tommiska, and J. Skytta, "Bridging the gap between future software and hardware engineers: a case study using the Nios softcore processor," in *IEEE Frontiers in Education*, Nov. 2003, pp. 1-5.

[36] S. Changchun and R. W. Brodersen, "Automated fixed-point data-type optimization tool for signal processing and communication systems," in *41st Design Automation Conference* 2004, pp. 478-483.

[37] A. A. Gaffar, O. Mencer, and W. Luk, "Unifying bit-width optimisation for fixed-point and floating-point designs," in *12th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 79-88.

[38] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design Integration Circuits and Systems*, vol. 25, pp. 1990-2000, 2006.

[39] P. Yu, K. Radecka, and Z. Zilic, "An efficient method to perform range analysis for DSP circuits," in *International Conference on Electronics, Circuits, and Systems (ICECS)*, 2010, pp. 855-858.

[40] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, pp. 886-896, 2005.

[41] A. Benedetti and P. Perona, "Bit-width optimization for configurable DSP's by multi-interval analysis," in *Proceedings of Signals, Systems and Computers Conference*, 2000, pp. 355-359

[42] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 1432-1442, 2003.

[43] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," *EURASIP Journal on Applied Signal Processing*, vol. 2006, 2006.

[44] K. Seehyun and S. Wonyong, "A floating-point to fixed-point assembly program translator for the TMS 320C25," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 41, pp. 730-739, 1994.

[45] K. Ki-Il, K. Jiyang, and S. Wonyong, "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, pp. 840-848, 2000.

[46] K. Ki-Il, K. Jiyang, and S. Wonyong, "A floating-point to integer C converter with shift reduction for fixed-point digital signal processors," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1999, pp. 2163-2166 vol.4.

[47] G. Caffarena, G. A. Constantinides, P. Y. K. Cheung, C. Carreras, and O. Nieto-Taladriz, "Optimal combined word-length allocation and architectural synthesis of digital signal processing circuits," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, pp. 339-343, 2006.

[48] J. A. Clarke, G. A. Constantinides, and P. Y. K. Cheung, "Word-length selection for power minimization via nonlinear optimization," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, p. 39, 2009.

[49] R. Moore, *Interval Analysis.* Englewood Cliffs, NJ: Prentice-Hall, 1966.

[50] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou, and Y. Zou, "Evaluation of static analysis techniques for fixed-point precision optimization," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 231-234.

[51] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," in *IEEE/ACM conf. Comput.-aided design (ICCAD'03)*, 2003, pp. 275-282.

[52] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems," in *Conference on Field Programmable Logic and Applications*, 2007, pp. 617-620.

[53] S. Kim, K. I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 45, pp. 1455-1464, 1998.

[54] G. A. Constantinides, "Word-length optimization for differentiable nonlinear systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, pp. 26-43, 2006.

[55] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A methodology and design environment for DSP ASIC fixed point refinement," in *Design, Automation and Test in Europe Conference and Exhibition*, , 1999, pp. 271-276.

[56] G. Caffarena, "Combined Word-Length Allocation and High-Level Synthesis of Digital Signal Processing Circuits," Ph.D Dissertation, E.T.S.I. Telecommunication (UPM), 2008.

[57] J. A. Lopez, G. Caffarena, C. Carreras, and O. Nieto-Taladriz, "Fast and accurate computation of the roundoff noise of linear time-invariant systems," *IET Circuits, Devices & Systems*, vol. 2, pp. 393-408, 2008.

[58] G. Caffarena, C. Carreras, J. A. Lopez, and A. Fernandez, "SQNR estimation of fixed-point DSP algorithms," *EURASIP Journal of Advanced Signal Process*, vol. 2010, pp. 1-12, 2010.

[59] J. A. Lopez, C. Carreras, and O. Nieto-Taladriz, "Improved interval-based characterization of fixed-point LTI systems with feedback loops," *IEEE Transactions on Computer-Aided Design Integration Circuits and Systems*, vol. 26, pp. 1923-1933, 2007.

[60] A. B. Kinsman and N. Nicolici, "Finite precision bit-width allocation using SAT-modulo theory," in *Design, Automation and Test in Europe (DATE '09)*, 2009, pp. 1106-1111.

[61] L. Zhang, Y. Zhang, and W. Zhou, "Floating-point to fixed-point transformation using extreme value theory," in *IEEE/ACIS International Conference on Computer and Information Science*, 2009, pp. 271-276.

[62] D. Boland and G. A. Constantinides, "Bounding variable values and round-off effects using Handelman representations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 1691-1704, 2011.

[63] M. A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," in *IEEE International Symposium on Circuits and Systems, ISCAS'02*, 2002, pp. 612-615

[64] B. Le Gal and E. Casseau, "Word-length aware DSP hardware design flow based on high-level synthesis," *Journal of Signal Processing Systems*, vol. 62, pp. 341-357, 2011.

[65] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum and heuristic synthesis of multiple word-length architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, pp. 39-57, 2005.

[66] D. Menard, N. Herve, O. Sentieys, and H.-N. Nguyen, "High-Level synthesis under fixed-point accuracy constraint," *Journal of Electrical and Computer Engineering*, vol. 2012, p. 14, 2012.

[67] H.-N. Nguyen, D. Ménard, and O. Sentieys, "Novel algorithms for word-length optimization," in *European Signal Processing Conference*, 2011, pp. 1944-1948.

[68] P. D. Fiore and L. Li, "Closed-form and real-time wordlength adaptation," in *IEEE Conference Acoustics, Speech, and Signal Processing*, 1999, vol.4, pp. 1897-1900

[69] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum wordlength allocation," in *IEEE Symp. Field-Programmable Custom Computing Machines*, 2002, pp. 219-228.

[70] P. Yu, K. Radecka, and Z. Zilic, "Optimization of imprecise circuits represented by Taylor series and real-valued polynomials," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1177-1190, 2010.

[71] D. U. Lee and J. D. Villasenor, "A bit-width optimization methodology for polynomial-based function evaluation," *IEEE Transactions on Computers*, vol. 56, pp. 567-571, 2007.

[72] H.-N. Nguyen, D. Menard, and O. Sentieys, "Design of optimized fixed-point WCDMA receiver," in *European Signal Processing Conference (EUSIPCO)*, Aug. 2009, pp. 993-997.

[73] S. A. Wadekar and A. C. Parker, "Accuracy sensitive word-length selection for algorithm optimization," in *International Conference on Computer Design: VLSI in Computers and Processors* 1998, pp. 54-61.

[74] N. Herve, D. Menard, and O. Sentieys, "Data wordlength optimization for FPGA synthesis," in *IEEE Workshop on Signal Processing Systems Design and Implementation*, 2005, pp. 623-628.

[75] G. Caffarena, J. A. Lopez, C. Carreras, and O. Nieto-Taladriz, "High-level synthesis of multiple word-length DSP algorithms using heterogeneous-resource FPGAs," in *Proceedings of FPL'06*, 2006, pp. 1-4.

[76] S. Vakili, J. M. P. Langlois, and G. Bois, "Customised soft processor design: a compromise between architecture description languages and parameterisable processors," *IET Computers & Digital Techniques*, vol. 7, pp. 122-131, 2013.

[77] A. Solomatnikov, A. Firoozshahian, O. Shacham, Z. Asgar, M. Wachs, W. Qadeer, S. Richardson, and M. Horowitz, "Using a configurable processor generator for computer architecture prototyping," in *IEEE/ACM International Symposium on Microarchitecture*, 2009,, pp. 358-369.

[78] P. R. Panda and N. D. Dutt, "Behavioral array mapping into multiport memories targeting low power," in *International Conference on VLSI Design*, Jan. 1997, pp. 268-272.

[79] C. Price, "Mips IV instruction set," *MIPS Tech. Inc*, 1995.

[80] P. Yiannacouras. (Apr., 2013). *SPREE:Soft Processor Rapid Exploration Environment*. Available: http://www.eecg.toronto.edu/~yiannac/SPREE/

[81] S. Vakili, D. C. Gil, J. M. P. Langlois, Y. Savaria, and G. Bois, "Customized embedded processor design for global photographic tone mapping," in *International Conference on Electronics, Circuits and Systems (ICECS)*, 2011, pp. 382-385.

[82] A. El-Mahdy and H. El-Shishiny, "High-quality HDR rendering technologies for emerging applications," *IBM Journal of Research and Development*, vol. 54, pp. 8:1-8:15, 2010.

[83] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda, "Photographic tone reproduction for digital images," *ACM Transactions on Graphics*, vol. 21, pp. 267-276, 2002.

[84] S. Vakili, J. M. P. Langlois, and G. Bois, "Finite-precision error modeling using affine arithmetic," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 2591-2595.

[85] R. E. Moore and F. Bierbaum, *Methods and applications of interval analysis*: SIAM, Philadelphia, 1979.

[86] Y. Pu and Y. Ha, "An automated, efficient and static bit-width optimization methodology towards maximum bit-width-to-error tradeoff with affine arithmetic model," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2006, pp. 886-891.

[87] S. Vakili, J. M. P. Langlois, and G. Bois, "Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 1853-1865, 2013.

[88] S. Vakili, J. M. P. Langlois, and G. Bois, "Accuracy-aware processor customization for fixed-point applications," *submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2014.

[89] S. Wonyong and K. Ki-Il, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Transaction on Signal Processing*, vol. 43, pp. 3087-3090, 1995.

[90] B. L. Evans. Raster Image Processing on the TMS320C7X VLIW DSP. Available: http://users.ece.utexas.edu/~bevans/hp-dsp-seminar/07_C6xImage2.pdf