# SYMBOLIC ALGORITHMS FOR EMBEDDED SYSTEM DESIGN

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Armita Peymandoust

June 2003

# ABSTRACT

The growing market of multi-media applications requires development of complex embedded systems with significant data-path portions. However, current hardware synthesis and software optimizations tools and methodologies do not support arithmetic-level optimizations necessary for data intensive applications. In particular, most high-level synthesis tools cannot automatically synthesize data paths such that complex arithmetic library blocks are intelligently used. Thus, the data paths of such circuits are often manually designed and mapped to pre-optimized library elements. Similarly, current compilers and software optimization methods are frequently incapable of optimizations required by multi-media software designers. Namely, most high-level arithmetic optimizations and the use of complex instructions and pre-optimized embedded library functions are left to the designers' ingenuity. In this thesis, results from symbolic polynomial manipulation techniques are used to develop algorithms for high-level data-path hardware synthesis, embedded-software optimization, and automated application specific embedded processor design.

Polynomials are chosen to abstract data-intensive software/hardware library elements and high-level specifications. Two new arithmetic-level symbolic polynomial decomposition algorithms are proposed. These algorithms map a specification to an implementation with minimum number of library elements or minimal delay.

The decomposition algorithms are applied to high-level synthesis of data intensive circuits by the tool SymSyn. SymSyn performs arithmetic optimization on dataflow descriptions and automatically maps them into data paths using complex arithmetic library components. SymSyn is capable of finding the minimal component mapping and the minimal critical-path delay mapping of the given dataflow. SymSyn is used in conjunction with a commercial behavioral synthesis tool on a set of dataflow descriptions. The results show impressive improvement in area and delay of the synthesized circuits compared to results from the standalone commercial behavioral synthesis tool.

Since energy optimization is a primary optimization goal in embedded system designs energy profiling is combined with the symbolic decomposition algorithms to optimize power-intensive sections of algorithmic multi-media embedded software. As a result, a tool flow and methodology is proposed that automatically maps critical code sections to complex processor instructions and pre-optimized software library available for a given processor. This optimization methodology is called SymSoft. SymSoft is used to optimize and tune the algorithmic level description of a set of examples including an MPEG Layer III (MP3) audio decoder for the SmartBadgeIV portable embedded system. In addition to improving designers' productivity, SymSoft lowers the number of instructions and memory accesses and thus lowers the system power consumption.

A growing number of embedded systems are using application-specific embedded processors. The design of these processors requires manual specialization of processors based on an application. Moreover, the use of the new complex instructions added to the processor is a manual task. Instruction set selection of application specific instruction set processors is automated by methods that automatically group dataflow operations in the application software as potential new complex instructions. The set of possible instructions is then automatically used for code generation combined with high-level arithmetic optimizations using the symbolic decomposition algorithms. These algorithms and methodology are used to automatically add new instructions to Tensilica processors for a set of examples. Results show improvements in designers productivity and efficient embedded processor specialization for the given applications.

The algorithms and methodologies presented in this thesis cover all aspects of embedded systems design including hardware, software, and processor design. These algorithms also bridge the gap between algorithmic design and the semantics of software and hardware description languages. This task is accomplished by using symbolic computer algebra that adds the knowledge of algebra to design tools.

## DEDICATION

To mom, dad, and Behrooz, with love and gratitude.

# ACKNOWLEDGMENTS

My deepest gratitude goes to my advisor Prof. Giovanni De Micheli for giving me the opportunity to work on this thesis. This work would not have been possible without his keen insight, guidance, and support. I would also like to thank my reading committee members, Prof. David Dill and Prof. Michael Flynn, for their time and effort spent on reading this thesis and serving on my Oral exam committee. I like to thank Prof. Zain Navabi for his encouragements and believing in me since the undergraduate years.

Discussions and suggestions from many members of the CAD group at Stanford have helped with parts of this research. I would like to thank Tajana Simunic for her directions on the software optimization work and her help with the SmartBadgeIV system. I appreciate Prof. Yung-Hsiang Lu's help with the data acquisition device and his feedbacks on my papers. I am grateful for discussions and feedbacks of Luc Semeria, Eui-Young Chung, and Prof. Luca Benini. Also, the presence and patience of all CAD group members during my talks is appreciated. I thank Evelyn Ubhoff and Kathleen DiTomaso for their prompt and caring support.

Life is an amazing journey. These past five years of my personal life were filled with extreme events, both pleasant and sad. It is a blessing to have families and friends to share the joyous moments and lean on when in need: My mother who directed me to where I am today with her love, the memory of my father for his unconditional love and support, my husband Behrooz for the gift of love and humor, Armin for the fun of living on the edge, Jeyran for always being there, and ... I thank you and love you all dearly.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

Today's electronic systems are increasingly more complex as a consequence of the exponentially growing transistor counts enabled by smaller feature sizes and the consumer demand for increased functionality, lower cost, and shorter time-to-market. Design technology tools and methodologies aim to reduce the renowned *productivity gap* and enable engineers to cost-effectively transform ideas into electronic systems. Revolutionary design technology tools have dramatically reduced the total design cost of *systems on a chip* (SOC). Examples of such groundbreaking tools include in-house *placement and route* softwares, *register transfer level* (RTL) synthesizers, and intelligent testbench generators. It is predicted that the next breakthrough design technology will be innovative embedded system level design tools and methodologies [1]. These tools will significantly reduce design cost and improve designer's productivity.

The reason behind this prediction is the ever-increasing demand for embedded systems and their growing complexity. Embedded systems on a chip are integrated circuits dedicated to a specific application or specific domain of applications. Such systems integrate microprocessor cores, memories, and custom hardware blocks on a single chip. Examples of these systems range from cell phones and personal digital assistants (PDAs) to medical instruments and automotive electronics.

There are several differentiating factors between the design of embedded systems and general-purpose electronic systems. The time to market and cost constraints of embedded systems are typically more aggressive. Energy efficient design is most important for portable embedded systems. Most notably, hardware and software design of embedded systems is more tightly coupled. Therefore, co-design of the software and hardware components of the embedded system is necessary for cost effective implementation. Tradeoffs must be made between implementing a function in hardware or software such that given quality and cost constraints are satisfied. Embedded system level design tools and methodologies are needed to automate the time consuming task of exploring the design space and to increase designer productivity.

## 1.1. MOTIVATIONS

In this thesis, the objective is to optimize the design of different components of an embedded system and to shorten the time required to design, optimize, and verify an embedded system by innovative system-level design algorithms, tools, and methodologies. Currently, embedded system designers have noticed that a shift to a high-level design methodology is inevitable in order to stay competitive in the market. Embedded systems include microprocessors, embedded software processes that execute on the microprocessors, custom hardware components, and memory blocks. In order to meet the aggressive time to market constraints, each component of an embedded systems must be designed at a higher level of abstraction.

Algorithmic-level specification of the design allows designers to specify more functionality to be specified more quickly. Synthesizing an algorithmic or architectural-level specification enables wider design space exploration that results in greater improvements of the system metrics. Furthermore, an algorithmic specification can be mapped onto more complex pre-optimized library blocks by high-level synthesis and component mapping tools. Effective reuse of pre-designed complex library elements results in designs that are correct by construction. Therefore, the time required to verify the design is dramatically reduced.

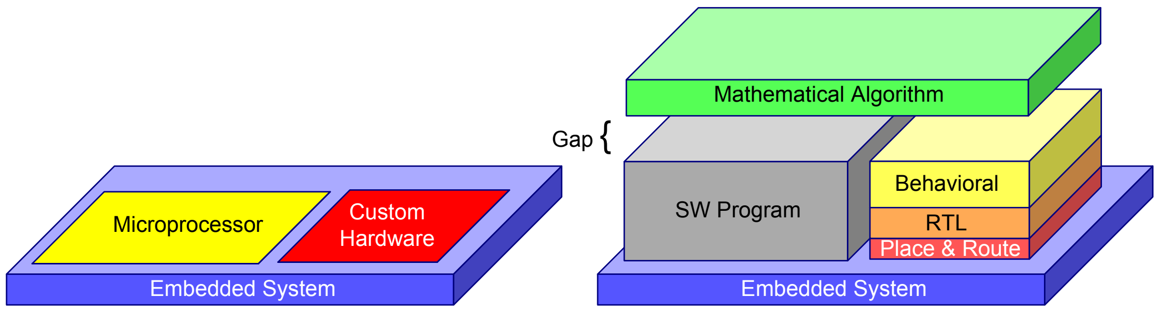


**Figure 1.1. Gap in Multimedia and DSP Embedded System Design**

The design of data-intensive embedded systems for multimedia and digital signal processing (DSP) applications starts with the mathematical description of these algorithms. However, there is a gap between the mathematical domain and the semantics of programming languages and behavioral synthesis as shown in Figure 1.1. An efficient design methodology for data-intensive embedded systems should bridge this semantic gap. In addition to searching the design space, such tool is expected to have the knowledge of algebra in order to automate mathematical optimizations required for algorithmic level design. In order to achieve this goal, the proposed tools and methodologies in this thesis exploit algorithms from symbolic computer algebra. Using symbolic computer algebra in algorithmic synthesis is analogous to the use Boolean algebra in logic synthesis tools that enable efficient synthesis of Boolean logic into control circuitry. Next, we will take a closer look at the design issues of the hardware portion, the embedded software processes, and the application specific processors used in data intensive embedded systems.

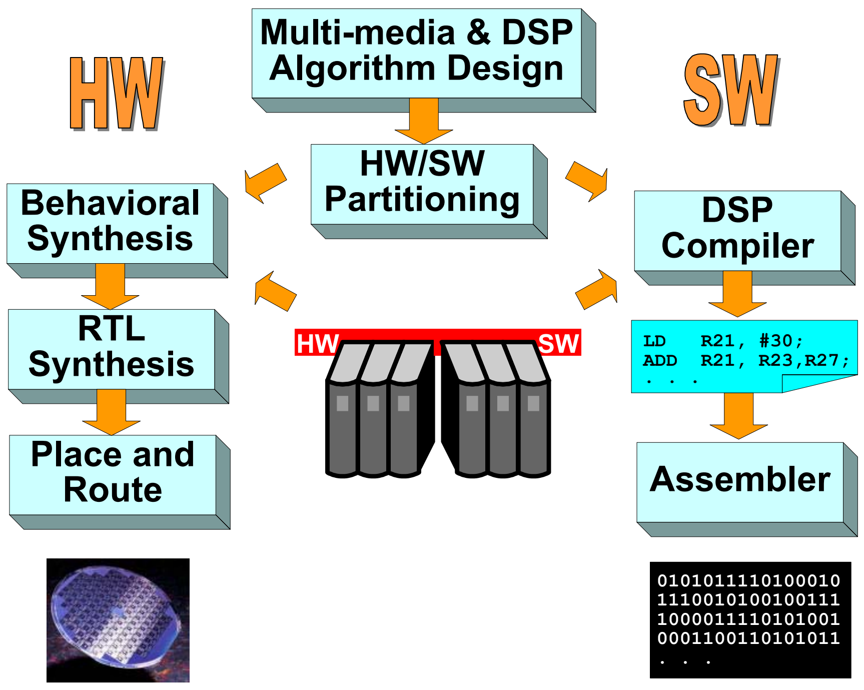


**Figure 1.2. Ideal Embedded System Design Flow**

## 1.2. DESIGN FLOW

As shown in Figure 1.2, the design of embedded systems starts with the algorithmic description of the application in a high-level language such as C or Matlab. In the ideal design flow of embedded systems, a software/hardware-partitioning tool automatically determines which sections of the system specification should be mapped to hardware and which parts should be implemented as software. After this decision is made, the architecture of the system is defined. To implement the custom hardware components of the system, the algorithmic level specification of these components is coded in a *hardware description language* (HDL). Next, a behavioral synthesis tool transforms this algorithmic or behavioral HDL code to its *register transfer level* (RTL) equivalent. The RTL description of the hardware is subsequently synthesized to a net-list of logic gates and memory elements using an RTL synthesis tool. Finally, the layout of the custom hardware is produced by a placement and route tool from the given net-list.

To implement the software portion of the embedded system, a microprocessor should be first chosen for the embedded system. One possibility is to select an off-the-shelf microprocessor suitable for the given application domain. Another possibility is to design an *application specific instruction set processor* (ASIP) for the given embedded system. In the latter case, an ASIP design tool takes the software application code and automatically generates an ASIP architecture and its supporting tools and compiler. In either case, the algorithmic C code of the application software is optimized and translated to assembly code by the compiler of the chosen embedded processor. This assembly code is next translated to machine code for the given microprocessor.

However, the reality of the embedded system design methodology is not as effortless and automatic as described above. Most transformations that start from a high-level algorithmic description require extensive manual intervention by the designer. In addition, with the increasing complexity of the embedded systems designs, automatic software and hardware design reuse is becoming increasingly important. Yet, the tool support for automatic design reuse does not match the real needs of designers.

In reality, most high-level synthesis tools and methods cannot automatically synthesize data paths such that complex arithmetic library blocks are intelligently used. Therefore, the hardware designers change the algorithmic HDL code such that it is suitable for current behavioral synthesis tools and manually map dataflow sections of the design to components available in the library of pre-designed arithmetic hardware blocks. This mapping is generally done by inserting synthesis directives that map the dataflow sections to the desired library components. However, automating this tedious task and the design of data paths from high-level specifications is necessary to meet aggressive time to market requirements. Namely, most arithmetic-level optimizations are not currently supported and they are left to the designers' ingenuity. In this thesis, it is shown that symbolic algebra can be used to construct arithmetic-level optimization and library mapping algorithms.

Moreover, embedded software engineers modify the algorithmic-level C code of the software and manually map the identified critical sections of the code to inline assembly. However, time to market of embedded software has become a crucial bottleneck. As a result, embedded software designers often use libraries that have been pre-optimized for a given processor to achieve higher code quality. Unfortunately, use of complex library elements and complex processor instructions is currently a manual task and depends on the designers' skills. In this thesis, algorithms and methodologies are presented that automate the use of complex processor instructions and pre-optimized software library routines simultaneous with high-level arithmetic optimizations using symbolic algebraic techniques.

Furthermore, there is a growing demand for application-specific embedded processors in system-on-a-chip designs. Current tools and design methodologies often require designers to manually specialize the processor based on an application. Moreover, the new complex instructions added to the processor often should be used manually through intrinsic function calls. In this thesis, a solution is introduced that automatically groups dataflow operations in the application software as potential new complex instructions. The set of possible instructions is then automatically used for code generation combined with high-level arithmetic optimizations using symbolic algebra.

## 1.3. THESIS OBJECTIVES

As seen in the previous section, the growing market of multi-media applications has required the development of complex *application specific integrated circuits* (ASICs) with significant data-path portions that accelerate the execution of the computational intensive kernels of the application. The optimal choice of the arithmetic units implementing complex dataflows strongly affects the cost, performance and power consumption of the silicon implementations. Unfortunately, current commercial tools rely on synthesis directives (*pragmas*) from designers in order to map dataflow into complex arithmetic library elements.

On the other hand, existing high-level synthesis tools are effective in capturing HDL models of the hardware and mapping them into *control/dataflow graphs* (CDFGs), performing scheduling, resource sharing, retiming, and control synthesis [8]. The approach presented in this thesis fits seamlessly into current high-level synthesis flow. The dataflow segments of the CDFG models are analyzed in light of the arithmetic units available as library blocks, and data paths are constructed that best exploit the given library. It is assumed that design is done using libraries that contain, beyond the basic elements such as adders and multipliers, more complex cells such as multiply/accumulate (MAC), sine, cosine, …. An example of such a library is the Synopsys DesignWare® [9] library. The first objective this thesis is to optimize and map dataflow descriptions into data paths that use complex arithmetic components.

In embedded system design environment, the degrees of freedom in software design are often much higher than the freedom available in hardware design. As a result, the primary requirement for embedded system-level design methodology is to effectively facilitate code performance and energy consumption optimization. Automating as many steps in the design of software from algorithmic-level specification is necessary to meet time to market requirements. Unfortunately, current available compilers and software optimization tools cannot meet all designers' needs.

Typically, software engineers start with algorithmic level C code, often developed by standards groups, and manually optimize it to execute on the given hardware platform such that power and performance constraints are satisfied. Needless to say, this conversion is a time-consuming and often error-prone task, which introduces undesired delay in the overall development process. The second objective of this thesis is to develop a software optimization methodology that reduces manual intervention. This methodology, SymSoft, is used to optimize a set of examples for the SmartBadgeIV, explained in Section 4.2, portable embedded system running the Linux embedded operating system [22]. The results of these optimizations show that by using SymSoft the critical basic blocks of the benchmark examples can be mapped to the StrongARM SA-1110™ instruction set much more efficiently than the commercial StrongARM compiler. SymSoft is also used to map critical code sections to commercially available software libraries with complex mathematical elements such as `exp` or the `IDCT` routine. Our measurements on SmartBadgeIV show that even higher performance improvements and energy savings are achieved by using these library elements.

Use of application-specific instruction-set processors (ASIP) in such embedded systems is a natural choice as ASIPs have time-to-market advantage over custom design ASICs and performance and power advantages over traditional fixed instruction set processors. Typically, software engineers start with a high level C code that specifies the application and manually specialize the embedded processor such that performance and cost constraints are satisfied. This process starts with profiling the application software to find the computation intensive segments of the code. Mapping these segments to hardware can greatly reduce the execution time of the application. Most base processors are capable of efficiently handling control segments of the application. Thus, the sections that benefit most from acceleration on hardware are data path or basic block segments. Consequently, the application-specific processor is manually tailored to include new ad-hoc functional units and instructions that calculate the computation critical basic blocks of the code. Nevertheless, specialization and design of ad-hoc functional unit extensions can be very lengthy and burdensome, which in turn introduces undesired delay in the overall development process.

In addition, most C compilers are unable to use the new complex instructions of the ASIP efficiently and automatically. In current design methodology, software designers manually insert intrinsic function calls that correspond to the new complex instructions in the computation intensive sections of the code. Manually inserting function calls is both time consuming and error prone. Moreover, designers often miss the opportunity of reusing the new instructions in other sections of the code to further reduce the execution time of the application. The third objective of this thesis is to provide a novel and effective method for instruction selection that is necessary due to the complexity of the automatically identified instructions. Using this methodology to new instructions are added automatically to Tensilica processors for a set of examples. Results show that designers' productivity is improved and embedded processors are efficiently specialized for the given applications such that the execution time is greatly improved.

## 1.4. THESIS CONTRIBUTIONS

In order to satisfy the objectives presented in the previous section a set of algorithms, tools, and methodologies are presented in this thesis. Their contributions can be summarized as:

1. For algorithmic design of the hardware blocks of the embedded system, two dataflow mapping algorithms are defined. These algorithms automate mapping dataflow sections of a high-level specification of the design to pre-optimized arithmetic library elements. This work introduces optimizations possible by the power of symbolic algebra for the first time in field of hardware thesis. The resulting tool enhances the capabilities of current high-level synthesis tools and designer's productivity.

2. A methodology and tool flow is defined for optimized embedded software programs. This methodology uses energy profiling to select critical section of an embedded software program. Next, algorithms are developed that map the critical section of the software to complex instructions available on the target microprocessor and embedded software library functions. This methodology was used to optimize a set of examples including an MP3 decoder software for a given

embedded system. Measurements on the system show dramatic performance and energy consumption improvements.

3. Since software and hardware blocks of an embedded system are tightly coupled, an efficient software/hardware co-design methodology is introduced in this thesis. This methodology aims at automating the selection and usage of the instruction set for an application specific processor. First, an algorithm is used to defined a set of promising instructions based on the given software application. Next, a symbolic decomposition algorithm maps the basic blocks of the application to the set of possible instructions. A final set of instructions is selected and used based on performance metrics of the application software. This results in adding hardware to the processor used in the embedded system to accelerate the software application and improve the overall performance of the embedded system.

## 1.5. THESIS OUTLINE

Chapter 2 provides a background on the concepts behind symbolic computer algebra and Buchberger's algorithm to calculate Gröbner basis of an ideal. This algorithm is used for multivariate polynomial elimination. Symbolic multivariate polynomial manipulations and variable elimination are used in the mapping algorithms presented in this thesis. Concepts explained in Chapter 2 are the backbones of this research. Chapter 3 describes how SymSyn uses symbolic algebra and polynomial representations to map dataflow sections of the hardware to a library of complex arithmetic blocks. First, previous work on deriving the canonical polynomial representation of a Boolean function is explained. Next, algorithms are explained that map the polynomial representation of a dataflow to a library represented by a set of polynomials. The mapping algorithms search for the minimal critical path delay implementation or for the implementation that uses the least number of components. Results are presented that show the advantage of component inference by SymSyn compared with a commercial behavioral synthesis tool in terms of area and delay. Chapter 4 describes our embedded software optimization methodology called SymSoft. SymSoft automates use of complex processor instructions and software library routines. First, the critical sections of the code are selected by execution time and energy profiling. These sections are then transformed into their polynomial representations. Symbolic computer algebra is used to map these polynomials to complex instructions available on the given processor and software functions available in the software library. SymSoft is used to optimize a set of application including an MP3 decoder for an embedded system called the SmartBadge. Results show impressive improvements in the performance and energy consumption of these examples. Chapter 5 focuses on the design of application specific instruction set processors. The goal is to take the application software and produce an instruction set and the optimized software based on the chosen instruction set. The dataflow sections of the code are processed to select a set of potential instructions that implement (parts of) the basic blocks. These potential instructions are used by a symbolic mapping algorithm for code generation. Results presented show that our algorithm and methodology can efficiently specialize embedded processors for a set of applications. Finally, Chapter 6 summarizes the contributions of this research and proposes future research directions.

## 1.6. ASSUMPTIONS AND LIMITATIONS

This thesis focuses on the optimization and mapping of dataflow sections of a software program or a hardware description. It is assumed that the control sections of the design are implemented efficiently by state-of-the-art compilers, synthesis tools, and basic embedded processors. The target of this thesis is to optimize and cost effectively design application domains such as multimedia and DSP applications. These applications have significant dataflow sections that perform arithmetic calculations. These dataflow sections are typically optimized manually. The algorithms, tools, and methodologies presented in this thesis complement control optimization capabilities of present compilers and synthesis tools to automated this process.

The mapping algorithms presented in this thesis assume that a polynomial representation is available for the dataflow section to be implemented. This assumption holds in an arithmetic intensive application domain such as the ones targeted in this research. When a dataflow section is calculating a transcendental function, its polynomial representation is obtained by approximation. It should be verified through simulation that the approximation used does not noticeably change the quality of the application output. This approximation and verification process is not the subject of this thesis and is currently a manual task that is to be automated in future work.

# CHAPTER 2
# BACKGROUND

To accelerate design and verification of embedded systems, hardware and software component libraries are available commercially for design reuse proposes. Hardware libraries include a set of pre-optimized complex hardware arithmetic components. An example of such library is the commercial DesignWare® [9] library by Synopsys that includes multiply-and-accumulate (MAC), sine, cosine, etc. A software library is a set of pre-optimized software routines. These library routines can be in-house code reused from previous projects or commercial software libraries available for a given processor. An example of a commercial software library is Intel's integrated performance primitives for the StrongARM SA-1110™ processor with routines such as *finite impulse response* (FIR) filter, *inverse discrete cosine transformation* (IDCT), Hamming decoder, etc.

Proposed algorithms, tools, and methodologies in this thesis, concentrate on arithmetic optimization and library mapping of the dataflow sections of the design. Two factors are key in automating the optimal mapping of dataflow blocks of a design into pre-optimized hardware and software libraries. First, a functionality description formalism for dataflow and library components. Second, methods supporting the decomposition of this formal representation into a set of library elements implementing arithmetic data paths.

The functionality description formalism needs to be compact and canonical. A natural way to represent dataflow sections of a description would be to represent them as polynomials. Polynomial representation has been proven as an effective technique [10][46][47] for representing both high-level specification and bit-level description of an implementation (library component), these methods will be described in Section 3.1. Furthermore, in embedded systems, cost efficiency of computational solutions is extremely important. Since, multi-media applications can tolerate certain output degradation polynomials can also be used for approximation and inexact mapping. The limited accuracy of a polynomial representation is analogous to the limited number of bits to representing floating point numbers in hardware.

Multivariate polynomial can be transformed into different equivalent polynomials and decomposed into other polynomials using a known set of algebraic polynomial decomposition methods and algorithms. These algorithms are implemented in mathematical tools such as Maple and Mathematica and often referred to as symbolic computer algebra. In the following sections, the basic theory behind symbolic multivariate polynomial algorithms is described in more detail.

## 2.1. SYMBOLIC COMPUTER ALGEBRA

Traditional mathematical computation with computers and calculators is based on arithmetic of fixed-length integers and fixed-precision floating-point numbers, otherwise known as numeric computer algebra. In contrast, modern symbolic computation systems support exact rational arithmetic, arbitrary-precision floating-point arithmetic, and algebraic manipulation of expressions containing undetermined values (symbols), such as variable `x` in `(x+1)*(x-1)`. Several commercial symbolic computer algebra systems are available on the market; Maple [2] and Mathematica [3] are most widely used.

The algebraic object to be manipulated symbolically is a multivariate polynomial that represents a (portion of) data path of our design. This polynomial should be decomposed into polynomials representing the building blocks available in the target library. Such decomposition is called *simplification modulo set of polynomials* in symbolic computer algebra. Most interesting symbolic polynomial manipulations for dataflow optimization are based on Gröbner bases [4][5][6][7]. Gröbner bases and Buchberger's algorithm generalize the division and greatest common divisor (GCD) algorithms of univariate polynomials to multivariate polynomials. Therefore, it is the heart of symbolic polynomial factorization.

Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of simplification modulo set of polynomials. In the following section, Gröbner basis and its application to the *simplification* algorithm are reviewed. Commercial symbolic computer programs, such as Maple [2], have a built-in routine that performs *simplification modulo set of polynomials*. In Maple, this method is called *simplify*. Next, the underlying theory of *simplification modulo set of polynomials* is described. The reader solely interested in its applications may proceed to Chapter 3.

## 2.2. BASIC COMMUTATIVE ALGEBRA

**Definition 2.1.** An *Abelian group* is a set G and a binary operation "+" satisfying all the following properties:

i. *Closure*. For every $a, b \in G$; $a + b \in G$.
ii. *Associativity*. For every $a, b, c \in G$; $a+(b+c)=(a+b)+c$.
iii. *Commutativity*. For every $a, b \in G$; $a+b=b+a$.
iv. *Identity*. There is an identity element $0 \in G$ such that for all $a \in G$; $a+0=a$.
v. *Inverse*. If $a \in G$, then there is an element $\bar{a} \in G$ such that $a+\bar{a}=0$.

**Definition 2.2.** A *commutative ring with unity* is a set R and two binary operations "+" and "·", referred to as addition and multiplication, as well as two distinguished elements 0, 1 $\in$ R such that the following axioms hold:

i. R is an Abelian group with respect to addition with additive identity element 0.
ii. *Multiplication closure*. For every a, b $\in$ R; a·b $\in$ R.
iii. *Multiplication associativity*. For every a, b, c $\in$ R; a·(b·c)=(a·b)·c.
iv. *Multiplication commutativity*. For every a, b $\in$ R; a·b=b·a.
v. *Multiplication identity*. There is an identity element 1 $\in$ R such that for all a $\in$ R; a·1=a.
vi. *Distributivity*. For every a, b, c $\in$ R; a·(b+c)=a·b+a·c holds for all a, b, c$\in$ R.

**Definition 2.3.** A *field* K is a commutative ring with unity, where every element in K expect 0 has a multiplicative inverse, i.e, $\forall$a $\in$ K–{0}, $\exists$ â $\in$ K such that a·â=1.

The set of all multivariate polynomials with variables $x_1, x_2, \ldots , x_n$, coefficients from a field K, and the two operations addition and multiplication forms a commutative ring with unity denoted by R [ $x_1, x_2, \ldots , x_n$ ].

**Definition 2.4.** Let R be a commutative ring, a non-empty subset I $\subseteq$ R is an ***ideal*** when [7]:

i. 0 $\in$ I,
ii. p + q $\in$ I for all p, q $\in$ I, and
iii. r · p $\in$ I for all p $\in$ I and r $\in$ R.

**Lemma 2.1.** Let $P = \{ p_1 , p_2 , \ldots , p_k \}$ be a finite subset of the polynomial ring R [$x_1, x_2, \ldots , x_n$] and $< P > = < p_1 , p_2 , \ldots , p_k > = \{ \sum_{i=1}^{k} h_i \cdot p_i \mid h_i \in R [x_1, x_2 , \ldots , x_n] \}$.

Then $< P >$ is an ideal in R [$x_1, x_2, \ldots , x_n$]. $< P >$ is called the ideal generated by $P$ and the set $P$ is called generator or ***basis*** of this ideal. For example, the set of polynomials

$P = \{p_1, \quad p_2, \quad p_3\}$ defined below generates a polynomial ideal over R $[x_1, x_2, x_3]$.

$p_1 = x_1^3\, x_2\, x_3 - x_1\, x_3^2,\; p_2 = x_1\, x_2^2\, x_3 - x_1\, x_2\, x_3,\; p_3 = x_1^2\, x_2^2 - x_3^2$

$< P > = \{a_1 \cdot p_1 + a_2 \cdot p_2 + a_3 \cdot p_3 \;\mid\; a_1, a_2, a_3 \in \mathrm{R}\, [x_1, x_2, x_3]\}.$

Unfortunately, while $P$ generates the infinite set $< P >$, the polynomials $p_i$ in $P$ may not yield much insight into this ideal, since for each ideal in a polynomial ring there are many possible sets of polynomials that generate the ideal. In other words, the ideal basis is not unique. However, Buchberger [4] has shown that an arbitrary ideal basis can be transformed into a basis with special properties, which is called the *Gröbner basis*. A minimal (or reduced) Gröbner basis forms a canonical representation for a multivariate polynomial ideal. A canonical representation for ideals enables us to check whether two ideals are equal. Important applications of Gröbner basis include polynomial decomposition and variable elimination in a set of multivariate polynomials. One may say that Gröbner basis is the cornerstone of polynomial decomposition used in our mapping algorithm. In the next section, a brief description of Buchberger's algorithm is given.

## 2.3. GRÖBNER BASES

Before introducing a formal definition of Gröbner bases, *term ordering* and *reduction* (division) of multivariate polynomials should be defined. A monomial of the form $x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$, where $x_1, x_2, \ldots, x_n$ are the variables of the polynomial and $(i_1, i_2, \ldots, i_n) \in Z^n_{\geq 0}$ are the exponents, is called a ***term***. The set of terms of the polynomial ring R$[x_1, x_2, \ldots, x_n]$ are denoted by $\mathrm{T}_{\mathbf{x}}$, where $\mathbf{N}$ is the set of non-negative integers:

$\mathrm{T}_{\mathbf{x}} = \{x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \;\mid\; i_1, i_2, \ldots, i_n \in \mathbf{N}\}.$

In division of univariate polynomials, R[x], the polynomials are written such that its terms are in decreasing order of the degree of x. To define reduction (division) for multivariate polynomials, an ordering for multivariate term is necessary.

**Definition 2.5.** A *term ordering* on R[$x_1$, $x_2$,… ,$x_n$] is any relation > on $Z^n_{\geq 0}$ satisfying:

i. > is a total (or linear) on $Z^n_{\geq 0}$.

ii. If α, β, and γ $\in Z^n_{\geq 0}$ and α > β, then α + γ > β + γ.

iii. > is well ordered on $Z^n_{\geq 0}$. This means that every nonempty subset of $Z^n_{\geq 0}$ has a smallest element under >.

The *leading monomial* of polynomial $p \in$ R[$x_1, x_2,\ldots, x_n$] with respect to a total ordering of the variables, such as the lexicographical ordering, is the monomial in *p* whose term is the maximal among those in *p*; this monomial is denoted by M( *p*). In addition, hterm( *p*) is defined as the maximal term and the hcoeff( *p*) is defined as the corresponding coefficient, therefore:

M( *p*) = hcoeff( *p*) · hterm( *p*).

**Example 2.1.** Consider $p \in$ R[$x_1$, $x_2$] that is written in lexicographical order: $p = 3x_1^2x_2+5x_1^2+x_2^2$, M( p) = $3x_1^2x_2$, hterm( p) = $x_1^2x_2$, hcoeff( *p*)=3. ■

**Definition 2.6. Reduction**: For nonzero *p*, $q \in$ R[$x_1, x_2,\ldots, x_n$] it is said that *p* reduces modulo *q* if there exists a monomial in *p* which is divisible by hterm(*q*). Let $\alpha \in$ R[$x_1, x_2,\ldots, x_n$]-{0}, i.e. the ring of polynomials after removing the trivial 0 polynomial. If $p = \alpha \cdot t + r$ where $t \in T_{\mathbf{x}}$, $r \in$ R[$x_1, x_2,\ldots, x_n$], and $u = \frac{t}{\text{hterm}(q)}$, $u \in T_{\mathbf{x}}$, then it is written as $p \rightarrow_q p'$ to signify that *p* reduces to *p*' (modulo *q*) and *p*' is equal to:

$$p' = p - \frac{\alpha \cdot t}{\mathrm{M}(q)} \cdot q = p - \frac{\alpha}{\mathrm{hcoeff}(q)} u \cdot q$$

**Example 2.2.** Consider the following two polynomials:
$p = 6x^4+13x^3-6x+1$, $q = 3x^2+5x-1$,
$p \rightarrow_q p'$; $p' = p - 2x^2 \cdot q = 3x^3+2x^2-6x+1$. ■

If $p$ reduces to $p'$ modulo a polynomial in a set of polynomials $Q = \{q_1, q_2, \dots, q_n\}$, it is said that $p$ reduces modulo $Q$ and written as $p \rightarrow_Q p'$ ( $p' = \mathrm{Reduce}(p,Q)$ ); otherwise $p$ is irreducible modulo $Q$. It is denoted that $p \rightarrow^+_Q p'$ if and only if there is a sequence such that:

$$p = p_0 \rightarrow_Q p_1 \rightarrow_Q \dots \rightarrow_Q p_n = p'.$$

If $p \rightarrow^+_Q q$ and $q$ is irreducible, it is written as $p \rightarrow^*_Q q$. It can be shown that for a fixed set $Q$ and a given term ordering, the sequence of reductions is finite [5]. Therefore, Algorithm 2.1 can be constructed which, given a polynomial $p$ and set $Q$, finds a polynomial $q$ such that $p \rightarrow^*_Q q$. In Algorithm 2.1, $\mathrm{R}_{p,Q}$ denotes the set polynomials in $Q$-$\{0\}$ such that hterm($p$) is divisible by hterm($q$). Note that any member of $\mathrm{R}_{p,Q}$ can be chosen in each iteration, but this choice affects the efficiency of the algorithm. For the sake of simplicity, it is assumed that an efficient selection is implemented in *selectpoly*.

As mentioned previously any finite set of polynomials $Q$ generates an ideal $<Q>$ and $Q$ is called the basis of this ideal. If a nonzero polynomial $p$ is reduced to zero modulo $Q$, it is determined that $p$ is a member of the ideal generated by $Q$: $p \rightarrow^*_Q 0 \Rightarrow p \in <Q>$. However, the converse is not true for all basis of $<Q>$.

**Algorithm 2.1.** Full Reduction of *p* Modulo *Q*.

```
procedure Reduce($p$, $Q$)

  # Given a polynomial $p$ and a set of polynomials $Q$
  # from the ring R[$x_1, x_2, \dots, x_{nz}$], find a $q$ such that $p \to^*_Q q$.
  # Start with the whole polynomial.
  $r \leftarrow p$; $q \leftarrow 0$

  # if no reducers exist, strip off the leading
  # monomial; otherwise, continue to reduce.
  while $r \neq 0$ do{
    $R \leftarrow R_{r,Q}$
    while $R \neq \varnothing$ do{
      #select a polynomial $\in$ R
      $f \leftarrow$ selectpoly(R)
      $R \leftarrow R - \{f\}$
      $r \leftarrow r - (M(r)/M(f))\, f$
    }
    $q \leftarrow q + M(r)$; $r \leftarrow r - M(r)$
  }
  return($q$)

end
```

**Definition 2.7.** An ideal basis $G \subset R[x_1, x_2, \dots, x_n]$ is called a *Gröbner basis* (with respect to a fixed term ordering and the implied permutation of variables) when $p \to^*_G 0 \Leftrightarrow p \in \langle G \rangle$.

*S-polynomial* of $p, q \in R[x_1, x_2, \dots, x_n]$, denoted as Spoly($p$, $q$), is defined as:

$$\text{Spoly}(p,q) = \text{LCM}(\text{M}(p), \text{M}(q)) \cdot [\frac{p}{\text{M}(p)} - \frac{q}{\text{M}(q)}].$$

**Example 2.3.** For polynomials $p = 3x^2y - y^3 - 6$ and $q = 6xy^3 + 5x - 1$ with degree ordering:

$\text{LCM}(\text{M}(p), \text{M}(q)) = \text{LCM}(3x^2y, 6xy^3) = 6x^2y^3$,

$$\text{Spoly}(p,q) = 6x^2y^3 \cdot [\frac{3x^2y - y^3 - 6}{3x^2y} - \frac{6xy^3 + 5x - 1}{6xy^3}] = -2y^5 - 12y^2 - 5x^2 + x \qquad ■$$

**Algorithm 2.2.** Buchberger's Algorithm for Gröbner Bases.

```
procedure Gbasis($Q$)

  # Given a set of polynomials $Q$, compute $G$ such that $<G> = <Q>$ and $G$ is a Gröbner basis.
  $G \leftarrow Q$; $k \leftarrow$ length($G$)

  # Initialize B to all possible pairs
  $B \leftarrow \{[i, j] : 1 \leq i < j \leq k\}$

  while $B \neq \emptyset$ do {
     $[i, j] \leftarrow$ select a pair from $B$
    # mark that pair as selected
    $B \leftarrow B - \{[i, j]\}$
    # $G_i$ denotes the i-th element of the ordered set $G$
    $h \leftarrow$ Reduce(Spoly($G_i$, $G_j$), $G$)
    if $h \neq 0$ then {
      $G \leftarrow G \cup \{h\}$; $k \leftarrow k + 1$
      $B \leftarrow B \cup \{ (i, k) : 1 \leq i < k\}$ }}
    return ($G$)

end
```

In can be shown that [5][6], $G$ is a Gröbner basis when:

1. the only irreducible polynomial in $<G>$ is $p = 0$;
2. Spoly(p, q) $\rightarrow^{+}_{G}$ 0 for all $p, q \in$ G;
3. if $p \rightarrow^{*}_{G} q$ and $p \rightarrow^{*}_{G} r$, then q = r.

Buchberger's algorithm (Algorithm 2.2) uses the properties above to convert a finite set $Q \subset R[x_1, x_2, \ldots, x_n]$ into a Gröbner basis [4].

In order to check whether a polynomial $p$ is a member of the ideal $<Q>$, first Algorithm 2.2 is used to form $G$ a Gröbner basis for $<Q>$. Procedure Reduce($p$, $G$) (Algorithm 2.1) must then return zero.

## 2.4. SUMMARY

The subset of symbolic computer algebra that performs multivariate polynomial manipulations was described in this chapter. These algorithms are mostly based on Gröbner basis. A minimal (or reduced) Gröbner basis is a canonical representation for a multivariate polynomial ideal that enables equality check of two ideals. Gröbner basis also facilitates ideal membership evaluation and multivariate variable elimination in a set of polynomials. Decomposing a dataflow polynomial into elements of a library represented by a set of polynomials, requires a sequence of reductions on the dataflow polynomial modulo library polynomials. Reduction, the basic step in polynomial division, was explained in this chapter. In the following chapters, it is shown how Gröbner basis and reduction of multivariate polynomials are used in automatic data-flow mapping and embedded system design.

# CHAPTER 3
# HIGH-LEVEL DATA-PATH SYNTHESIS

In this chapter, a tool called SymSyn is presented that leverages results from Gröbner basis [4][5][6][7] applications and symbolic polynomial manipulation techniques to automate mapping of (a portion of) dataflow into complex arithmetic library blocks. SymSyn framework contains two decomposition algorithms that assume the dataflow and library elements are represented as polynomials. The first algorithm finds a minimal-component decomposition of a polynomial representing a (portion of) dataflow. The decomposition is done in terms of arithmetic library elements, also represented as polynomials. Due to the importance of high performance design, a second algorithm in the SymSyn framework is developed to automatically map the dataflow to arithmetic library elements such that the dataflow has minimal critical path delay. The timing-driven decomposing algorithm uses various polynomial manipulation techniques as guidelines to achieve optimal component mapping and resource sharing for minimal delay.

**Example 3.1.** As a motivating example, consider the anti-alias function of a MP3 decoder that calculates the following equation in one of its basic block:

$z = \dfrac{1}{2\sqrt{x^2 + y^2}}$; under the assumption that $x^2 + y^2 \geq \varepsilon > 0$ .

A straightforward realization of this equation would use a divider and a square root operator, which are large and slow components and may not be available in the component library. For the sake of the example, assume there are no square root and division operators available in the library. Alternatively, assume the existence of adder, multiplier, and multiplier-accumulator (MAC) in the given library. Thus, $c=x^2+y^2$ can be easily computed. Next, using symbolic manipulations $x^2+y^2$ is substituted by $c$.:

$$z = \frac{1}{2\sqrt{c}} .$$

The given equation can be approximated to a polynomial representation using Taylor series expansion for a range of $c$ based on the given application:

$$z \cong \frac{1}{64}c^6 - \frac{9}{32}c^5 + \frac{115}{64}c^4 - \frac{75}{16}c^3 + \frac{279}{64}c^2 - \frac{81}{32}c + \frac{85}{64}$$

The explanation is valid for a given range of $c$ and the error can be computed using standard approximation methods [11]. If Horner based transform is performed on the polynomial approximation of $z$, we obtain:

$$z \cong \frac{85}{64} + \left( -\frac{81}{32} + \left( \frac{279}{64} + \left( -\frac{75}{16} + \left( \frac{115}{64} + \left( -\frac{9}{32} + \frac{1}{64}c \right)c \right)c \right)c \right)c \right)c$$

This formula can be implemented using a chain of 6 MACs, or one MAC in 6 cycles. Figure 3.1 demonstrates one possible implementation. ■

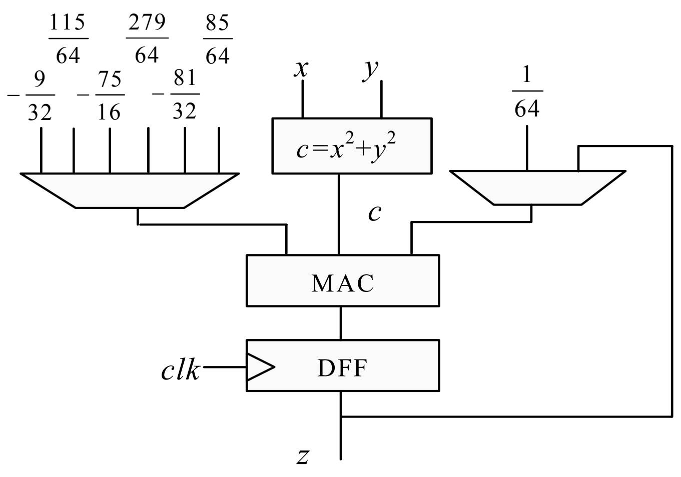


**Figure 3.1. An Implementation for $\frac{1}{\sqrt{x^2+y^2}}$**

The synthesis tool described in this chapter, SymSyn, automates the algebraic manipulations shown in this example. SymSyn converts the basic blocks of a behavioral description, representing dataflow portions of the design, to their polynomial representations and uses numerical methods for exact and inexact matching with library elements. If a match is not found, the dataflow is decomposed into the library elements using symbolic computer algebra.

This chapter is organized as follows: Section 3.1 gives an overview on related work in this area. Section 3.2 explains how symbolic algebra and Gröbner basis are used in polynomial decomposition algorithms. In Section 3.3, it is shown how results from symbolic algebra can be leveraged to decompose a polynomial representing a (portion of) dataflow. In Section 3.3, the dataflow synthesis tool, SymSyn, is explained with an example. Sections 3.4 and 3.5 describe the two new algorithms developed for automatic decomposition of dataflow into complex arithmetic library components. Section 3.6 shows a set of library independent symbolic transformations that are used to accelerate the proposed algorithms. Finally, Section 3.7 explains the implementation of SymSyn and shows a set of experimental results.

## 3.1. RELATED WORK

High-level synthesis and design reuse are essential for system on chip designs. They can shorten the time required to specify and design a complex system. Since high-level synthesis takes specifications at a level of abstraction greater than RTL, wider design space exploration becomes possible [8]. Current high level synthesis tools are capable of optimizations such as scheduling and resource sharing. Moreover, these tools synthesize control sections of the design efficiently. However, dataflow and arithmetic optimizations of the design are generally left to the designer. Mapping the dataflow sections to pre-designed components is also a manual task. This is presently possible by synthesis directives manually inserted by the designer.

Most classical work on data-path synthesis focus on allocation of hardware resources based the availability and scheduling constraints. The MAHA system used critical path determination to perform hardware allocation [59]. The expert system approach was taken in the DAA system that develops a rule based data memory controller [60]. More recently, carry save representation was used for module selection simultaneous with retiming [61]. In this work, carry-save transformations are preformed across register boundaries to optimize a synchronous circuit.

In other work [10][46][47], algorithms were developed that enhance high-level synthesis tools with the capability of mapping high-level specifications onto existing components. Word-level polynomial representations were introduced as a mechanism for canonically and compactly representing the functionality of complex components. These polynomials provide the basis for efficiently comparing the functionality of a circuit specification and a complex component. Polynomial methods allow a specification to be compared against potential implementations by computing the numerical distance between the two. This not only enables fast allocation of exact implementations, but also allows for detection of approximate and partial implementations.

Polynomial representation of Boolean functions is performed by determining the order of the minimum polynomial that can represent the given function. This figure is then used to extract the appropriate number of coordinates from a component to compute polynomial coefficients. Polynomial representation has been used in matching dataflow clusters of the design to library cells in the tool POLYSYS [10][46][47]. However, POLYSYS is limited to test for a match in the library of existing components. In case a match did not exist, there was no automated way to search for possible interconnections of library blocks matching the dataflow cluster. In this chapter, symbolic computer algebra is used to map a polynomial representation of the extracted dataflow section of our design to a set of polynomial representations of our library elements. This mapping is performed simultaneously with high-level arithmetic optimizations.

## 3.2. GRÖBNER BASES AND DATA-PATH SYNTHESIS

The application of the theory described in Chapter 2 is presented in this section. Let $L$ be the set of polynomial representations of the library elements. In order to synthesize a data path for a polynomial representation $S$ using library $L$, $S$ should be a member of $<L>$. In order to examine membership in $<L>$, first $G$ the Gröbner basis of $<L>$ is calculated and next Reduce($S$, $G$) is used. If $S$ is reduced to zero then $S \in <L>$. If $S$ is reduced to zero only using polynomials in $G$ that are also in $L$, then $S$ can be built from the given library elements.

**Example 3.2.** As an example, consider:
$S = x + x^2 + x^3 + y + xy + x^2y$;
$L = \{1+x+x^2, x+y\}$;
$G =$ Gbasis($L$) $= \{x+y, y^2-y+1\}$;
Reduce($S$, $G$) returns zero, therefore $S \in <L>$.
While performing Reduce($S$, $G$), we determine that:
$S = (x+y)(1+x+x^2)$; therefore $S$ can be decomposed into elements of $<L>$. ■

## 3.3. SYMBOLIC ALGEBRA AND LIBRARY MATCHING

After extracting the CDFG of an algorithmic level DSP model, the polynomial representations of its basic blocks are calculated. The polynomial representation of a basic block can be directly extracted from algorithmic-level code if the basic block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [46] can be used to formulate the equivalent polynomial representation. When the basic block implements a transcendental function, an approximation such as the Taylor or Chebyshev series expansion is used as its polynomial. The chosen polynomial approximation has to be verified manually by simulation to ensure that constraints, such as accuracy, are satisfied.

Symbolic computer algebra is subsequently used to intelligently decompose dataflow to library components and automatically synthesize the data path. The symbolic algebra routine used in this algorithm is *simplification modulo set of polynomials* that has been described in Chapter 2. Assume a basic block (or part of it) is represented by polynomial $p$ and the library components available are represented by a set of polynomials $L$. As a reminder, to simplify a polynomial $p$ modulo the side relation set $L$, a Gröbner basis is derived from $L$, $G \leftarrow \text{Gbasis}(L)$, and Reduce($p$, $G$) is used to obtain the simplified answer. The built-in function that implements *simplification modulo set of polynomials* in Maple is called *simplify* [2]. In order to comply with Maple terminology, we call the set of polynomials the *side relations*.

Note that any polynomial representation can be implemented using only adders and multipliers. Therefore, any polynomial representation of a basic block is guaranteed an implementation if the library includes adder and multiplier. Our goal is to find non-trivial solutions that are minimal in terms of component count or critical path delay.

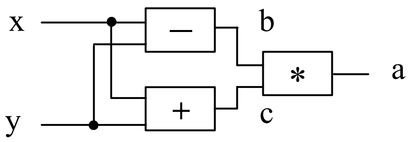


**Figure 3.2. An Implementation of $x^2$-$y^2$**

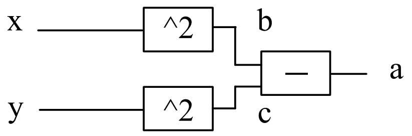


**Figure 3.3. An Alternative Implementation of $x^2$-$y^2$**

**Example 3.3.** As an example, consider a dataflow implementing `x^2-y^2` and a library that includes add, multiply subtract and square functions. Using Maple syntax, we have:

```
>  a:=x^2-y^2: siderels:={b=x-y, c=x+y}
>  simplify(a, siderels,[x,y,b,c]);
   b*c
```

This is equivalent to the implementation shown in Figure 3.2. Note that `siderels` is a subset of our library. Maple computes the Gröbner basis *G* of `siderels` and prints out the result of Reduce(`a, siderels`). The result indicates that:

```
a:=x^2-y^2:=b*c:=(x-y)*(x+y)
```

If the side relation set is changed, other possible solutions for the specification might be found, for example:

```
>  a:=x^2-y^2: siderels:={b=x^2, c=y^2}
>  simplify(a, siderels,[x,y,b,c]);
   b-c
```

results in the implementation shown in Figure 3.3. ■

As shown in the previous example, different side relation sets can result in different implementation of the specification. Therefore, to find the best possible implementation, the side relation set should be set equal to all subsets of the library with all possible permutations of the input variables. Since this is exponentially expensive, a guided architectural exploration is necessary. In the next two sections, two algorithms are introduced that reduce the complexity of this search with two different final objectives. The first algorithm finds the minimal component decomposition for the given dataflow. The second algorithm finds the minimal critical path delay implementation of the dataflow.

## 3.4. MINIMAL COMPONENT DECOMPOSITION ALGORITHM

In this section, one of the algorithms implemented in our tool SymSyn is introduced. This algorithm automatically maps a polynomial representation of a (portion of) dataflow to a set of complex arithmetic library components while using the least number of library components. This algorithm in conjunction with classical high-level synthesis algorithms can be used for efficient high-level DSP synthesis. The minimal component decomposition algorithm described is empowered by Gröbner basis fundamentals described in Chapter 2. The inputs to this algorithm are polynomial representation of the dataflow basic block to be synthesized and a set of polynomials that represent the set of complex arithmetic library components available to the designer. As mentioned in the previous section, different side relation sets result in different implementations of the dataflow. Therefore, the described algorithm aims at intelligent side relation set selection to accelerate the decomposition process for a given criteria. The high-level view of the selection criteria for minimal number of components is illustrated in Algorithm 3.4.

```
Algorithm 3.4. Decompose S into elements of library L

procedure Decompose(S, L)

  # Given polynomial representation of the spec S and a set of polynomials L as library,
  # decompose S into elements of library L.

  # initialize tree
  treeroot(S);
  depth ← 0
  bound ← -1

  while depth ≠ bound do {
     bound ← Explore(S, L, depth)      # Explore is defined below
     depth ← depth +1
  }

  report best solution in tree

end

# used in Decompose procedure
int function Explore(S, L, d)

  bound ← -1
  for all n ∈ in tree with depth d do{
    for all sr ∈ L do{
      result = simplify(n, sr);

      # make result a child of node n
      addchild(n, result);

      if result ∈ L
         # solution is found
         bound = treedepth(result);  }}

  # returns –1 if no solution is found yet.
  return(bound)

end
```

Let $S$ be the polynomial representation of the basic block to be decomposed into complex library elements. The algorithm starts by simplifying $S$ modulo each library element as the side relation. The simplification results are stored in a tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the simplification result stored in a tree node does not correspond to a library element, the same steps are recursively applied to the new tree node.

To further reduce the search space a bounding function is used. The bounding function is the number of library components used to build the specification. In other words, if a solution is found with two library components the solutions requiring more than two components will not be explored. Nevertheless, all two-component solutions will be uncovered and the one with optimal cost (area or delay) will be chosen. The number of components used is the same as the depth of the simplification tree. Therefore, the tree is bounded by the depth of the first solution found.

Such bounding function is chosen assuming that if a component is custom designed to perform a combination of arithmetic operations, it is more cost effective than connecting a series of components that perform the same arithmetic operations. Clearly, the merit of the result is strongly dependent on the available library.

## 3.4.1. Minimal Component Example

To clarify the algorithm described above, the library is chosen as a subset of the Synopsys DesignWare® library consisting of six combinational elements; multiplier, adder, subtracter, multiplier-accumulator, sine, and cosine. As an example, consider synthesizing a phase shift keying (PSK) modulator used in digital communication. A dataflow basic block of PSK has the following polynomial representation:

```
>  S:= 1-.5*x0^2-x0*x1-.5*x1^2+.041667*x0^4+.166668*x0^3*x1+
       .250002*x0^2*x1^2+.166668*x0*x1^3+.041667*x1^4;
```

As the first step, SymSyn initializes a tree data structure and stores polynomial *S* in the root of the tree. For all library elements, SymSyn makes a call to Maple and requests simplify with side relation set equal to the library element. The results reported by Maple are kept as new children of the *S* tree node.

In the first iteration of our example, the side relation is set to the first element in the library, the multiplier. Shown below are the Maple commands. The first two lines are the requests sent by SymSyn and the third line is the simplification result reported by Maple to SymSyn. SymSyn searches for a component in the library that implements the result, but it is not successful to find one for this instance.

```
  >  siderel := {y=x0*x1};
  >  simplify(S, siderel, [x0,x1,y]);
     .041667*x0^4+.166668*x0^2*y-
  .5*x0^2+.041667*x1^4+.166668*x1^2*y-.5*x1^2+.250002*y^2-1.*y+1
```

In the second iteration, the same steps are performed with the adder as the side relation. The simplification result now matches an approximation to the cosine function. Therefore, SymSyn marks this node as one possible solution. The following Maple commands show the result of this iteration. Note that the result is a Taylor series approximation of cosine. Since cosine is one of the library elements, one possible solution is found as shown in Figure 3.4.

```
  >  siderel := {y=x0+x1};
  >  simplify(S, siderel, [x0,x1,y]);
     1.+.041667*y^4-.5*y^2
```

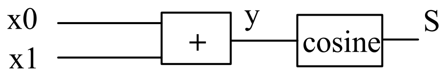


**Figure 3.4. Mapping the S dataflow to Two Components**

Since there is a solution with depth equal to *one* in the tree, a bound of *one* is set on the tree growth. Note that the root is denoted with depth equal to *zero*. Therefore, a solution at depth *one* consists of two components. SymSyn performs the steps described above for the rest of library elements and keeps the results in root offsprings. After going through all library elements, SymSyn finds only one solution using two components. The solution is demonstrated in Figure 3.4. SymSyn will stop decomposing the leaf nodes, since continuation would result in a search for solutions with three or more components while the objective is to find a solution using minimal number of components.

## 3.5. TIMING DRIVEN DECOMPOSITION ALGORITHM

In this section, the second algorithm implemented in SymSyn is introduced. In contrast to the algorithm described in Section 3.4, here the focus is on minimizing the critical path delay of the dataflow implementation. Previously, minimizing the number of components used to implement the dataflow was the primary objective. Similar to Algorithm 3.4, this algorithm selects side relation sets intelligently to accelerate the decomposition process, since selecting different side relation sets result in different implementations of the dataflow.

After extracting the CDFG of an algorithmic-level DSP model, the polynomial representations of its dataflow basic blocks are passed as inputs to the timing-driven decomposition algorithm. Algorithm 3.5 shows the pseudo-code of the timing-driven decomposition algorithm. This algorithm takes the same inputs as Algorithm 3.4; the polynomial representation of the basic block to be implemented and the polynomial representations of the complex library elements. Algorithm 3.5, uses the branch-and-bound method to reduce the side-relation-set selection space while searching for the implementation with least critical path delay. We define the bounding function as the best critical path delay of implementations seen so far. The lower bound computed at each decision branch is the critical path delay of components in the side relation set in view of data dependencies. If this lower bound is greater than the best critical path delay of implementations seen so far, the corresponding decision branch is pruned.

```
Algorithm 3.5. Decompose S into elements of library L
function GuidedDecomposition(exp_tree, max_CPD, L){
  # initialize a solution tree
  solution_tree ← tree(exp_tree);
  depth ← 0
  bound ← max_CPD
  for all n ∈ in solution_tree with depth == depth do{
    if depth == 0 then
       choose all sr ∈ L that preserve the exp_tree structure
    else for all sr ∈ L do{
      if cost of sr + cost of node n < bound then {
        result = simplify(n, sr);
        # make result a child of node n
        addchild(n, result);
        add cost of sr to cost of result;
        if result ∈ L then {
          # solution is found
          bound = cost of node result;  }
              if no more n ∈ in solution_tree with depth == depth
          depth ← depth +1
    }}
    return the best solution in solution_tree
end
int function CalcMaxCPD(expression_tree){
    CPD = the critical path delay of expression_tree assuming
            the expression is mapped to adders and multipliers only.
    return(CPD)
end
procedure main(S, L)
# Given polynomial representation of the spec S  and a set of polynomials L as library,
# decompose S into elements of library L such that the CPD of S is minimized.
# perform expression manipulation techniques
  exp_tree [1..NumberOfManipulations] = AllManipulations (S);
  for i= 1 to NumberOfManipulations do{
    maxCPD[i]=CalcMaxCPD(exp_tree[i]);
    solution[i]=GuidedDecomposition(exp_tree[i], maxCPD[i]);
  }
  report the best solution in solutions[i]
end
```

Let *S* be the polynomial representation of the dataflow. Our goal is to decompose *S* into the elements of the library *L* such that the critical path delay of *S* is minimized. Decomposing *S* is synonym to simplifying *S* modulo elements of the library *L* as side relations. In order to decide which library elements should be used as the side relations, a decision tree (*solution_tree*) is used to implement the branch-and-bound algorithm. The bounding variable is initialized to the critical path delay of mapping the polynomial solely to adders and multipliers, a.k.a. the lexicographical mapping.

The *simplify* results are also saved in the tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the critical path delay of the solution is smaller than previously encountered solutions, the bounding variable is set to the current delay. In case the simplification result stored in a tree node does not correspond to any library elements, the same steps are recursively applied to the new tree node.

In general, the branch-and-bound algorithm is practically applicable to most problems. However, introducing heuristics that lead quickly to promising solutions can improve the execution time without hampering the quality of the solution. As for all branch-and-bound algorithms, the worst-case complexity remains exponential.

The expression manipulation techniques presented subsequently in Section 3.6 are used as heuristic guidelines for choosing the side relation set. Initially, tree-height reduction, factorization, expansion, and the Horner-based transform are applied on *S*. As a result, there are several polynomials (*exp_tree*) representing the same dataflow. Each of these representations can result in the desirable implementation based on the available library elements. Starting with the primary inputs, the expression tree is covered with the library elements. All library elements that cover the primary inputs and a portion of the expression tree are chosen as elements of side relation sets. If the result of simplify modulo side relation is not a library element, the result is decomposed without further guidance from the expression tree. Algorithm 3.5 in conjunction with substitution and tree-height reduction can be generalized to several polynomials in a basic block or across basic blocks.

**Example 3.4.** As an example, consider a dataflow segment of a Gabor filter with the following polynomial representation:

$$D = 1 - a^2 - b^2 + a^2b^2 + \frac{1}{2}a^4 + \frac{1}{2}b^4 - \frac{1}{6}a^6 - \frac{1}{2}a^4b^2 - \frac{1}{2}a^2b^4 - \frac{1}{6}b^6 + \frac{1}{24}a^8 + \frac{1}{6}a^6b^2 + \frac{1}{4}a^4b^4 + \frac{1}{6}a^2b^6 + \frac{1}{24}b^8$$

Assume that *D* is to be mapped to a library consisting of functions implementing add, multiply, MAC, square, exp. After factorization, *D* will be converted to:

$$D = \frac{1}{24}(a^2 + b^2) \cdot (a^6 - 4a^4 + 3a^4b^2 + 12a^2 - 8a^2b^2 + 3a^2b^4 + 12b^2 + 24 - 4b^4 + b^6) + 1$$

The factored form of *D* guides us to use c=a^2+b^2 as an initial side relation and sets an initial bound by mapping the factored form lexicographically to adders and multiplier. SymSyn makes a call to Maple and requests result of the following simplify operation.

```
> siderel := {c=a^2+b^2};
> result:=simplify(D, siderel, [a,b,c]);
  result=1-c+1/2*c^2-1/6*c^3+1/24*c^4
```

The last line is the result reported to SymSyn by Maple. As it can be seen, the result is a Taylor series expansion of `exp(c)`. Therefore, the dataflow can be implemented using two square components, an adder, and one `exp` component, as shown in Figure 3.5. The bounding function is now changed to the critical path delay of the potential implementation. By exploring the other branches of the decision tree (*solution_tree*), we realize that all other branches are pruned by the new bound. Therefore Figure 3.5 is implementation with the least critical path delay.

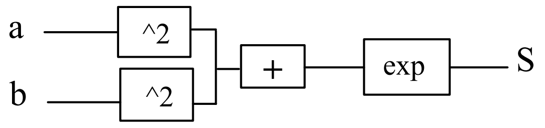


**Figure 3.5. Mapping the D dataflow to Four Components**

Now, assume that there is no `exp` block in the library. In order to show the power of other polynomial transformations, the Horner transform (see Section 3.6.3) is performed on the polynomial *result*:

$$result = 1 + (-1 + (\frac{1}{2} + (-\frac{1}{6} + \frac{1}{24} \cdot c) \cdot c) \cdot c) \cdot c$$

The formula given above can be implemented using a chain of 4 MACs, or one MAC in 4 cycles. Figure 3.6 demonstrates one possible implementation. ■

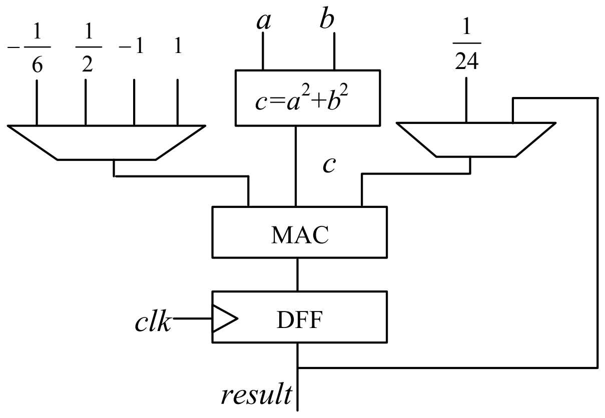


**Figure 3.6. A Possible Implementation for** $e^c$

## 3.6. EXPRESSION MANIPULATION TECHNIQUES

In Section 3.5, an algorithm was introduced that maps a polynomial representation of a (portion of) dataflow to complex arithmetic library elements such that the critical path delay is minimized. This algorithm was implemented in the Symbolic Synthesis tool, *SymSyn*. To accelerate the speed of minimal critical path delay decomposition in *SymSyn*, a guideline is necessary for side-relation selection. Such guideline should facilitate mapping for maximum parallelism. Different symbolic polynomial manipulation techniques are chosen as such guidelines. These transformations are the counterparts of the library independent transformations used in logic synthesis [8]. These heuristics can also be used as an enhancement to the minimal component decomposition algorithm. The intent of this section is to describe the manipulation techniques through simple examples.

### 3.6.1. Tree-height Reduction

*Tree-height reduction* (THR) was introduced long ago [12][13] as an optimization method for parallel software compilers. It is a technique to reduce the height of an arithmetic expression tree, where the height of the tree is the number of steps required to compute the expression. In the best case, it achieves the tree height of $O(\log n)$ for an expression with $n$ operations. Tree-height reduction uses commutativity, associativity, and distributivity properties of addition, subtraction, and multiplication. In the classical case, tree-height reduction is achieved at the expense of adding more resources to obtain maximum parallelism in the expression. In previous work for hardware synthesis, THR has been proven useful in high-level synthesis of data-intensive circuits such as DSP and multimedia applications [14][15][16].

In our work, THR is used as an expression tree manipulation technique. THR will achieve the best execution time when using unlimited number of two input adders, subtracters and multipliers. Since the focus in this thesis is on libraries with more complex blocks, THR may or may not result in the optimal execution time. The result is dependent on the library components available.

**Example 3.5.** Figure 3.7 shows an example of how THR can reduce the critical path delay. Figure 3.7b is obtained after applying THR on Figure 3.7a. ■

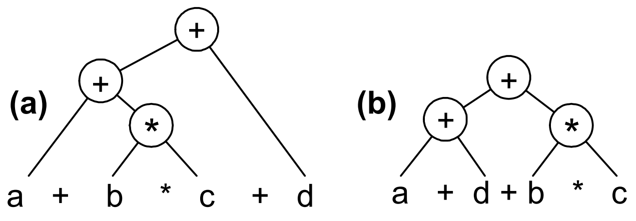


**Figure 3.7. Performing THR on (a) Produces (b)**

3.6.2. FACTOR AND EXPAND

As mentioned previously, traditional tree-height reduction [12][13] only uses associativity, commutativity, and distributivity to transform expressions. Since we have access to a symbolic manipulation tool in *SymSyn*, we can benefit from other transformations as well. One such transformation is common sub-expression factorization. Factorization can reduce the number of components used as well as the tree height of a given expression.

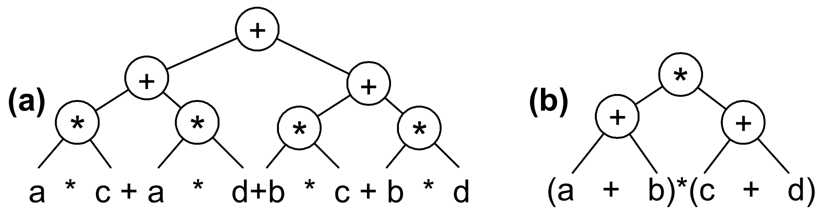


**Figure 3.8. Factor May Reduce Number of Components and CPD**

**Example 3.6.** An example is shown in Figure 3.8. Factorization transforms the expression shown in Figure 3.8a to the expression show in Figure 3.8b. Figure 3.8b has three less multiplications, one less addition, and shorter tree height compared to Figure 3.8a. . ■

Another useful symbolic manipulation technique is expansion. This manipulation technique changes the polynomial into its sum of products format. Meanwhile, it is capable of straightforward simplification techniques that can save both delay and area.

**Example 3.7.** A small example of expansion transforming `a+a+a` to `3*a` which is more simplified. ■

### 3.6.3. Horner Form

Horner form of a polynomial is a nested normal form with minimal number of multiplications and additions. Any polynomial can be rewritten in Horner, or nested, form. The general univariate case is defined as follows [3]:

$$\begin{aligned} p(x) &= a_0 \cdot x^n + \cdots + a_{n-1} \cdot x + a_n \\ &= (\cdots((a_0 \cdot x + a_1) \cdot x + a_2) \cdot x + \cdots a_{n-1}) \cdot x + a_n \end{aligned}$$

Assume that $x^n$ can be calculated using only $\log_2(n)$ multiplications for integer $n$. For a polynomial of degree $n$, the Horner form requires $n$ multiplications and $n$ additions. The expanded form, however, requires:

$$\sum_{i=1}^{n} \log_2(i) = \log_2(n!)$$

multiplications, which is more than twice as expensive for a polynomial of degree 10. Thus, one advantage of Horner form is that the work involved in exponentiation is distributed across addition and multiplication, resulting in savings of some basic arithmetic operations. Another advantage is that Horner form is more stable to evaluate numerically when compared with the expanded form. This is because each sum or product involves quantities which vary on a more evenly distributed scale [3]. For hardware implementation, Horner form has a distinct advantage. It effectively maps a univariate polynomial to cost effective multiplier-accumulators (MAC). Horner form is generalized for multivariate polynomials by specifying an ordered list of variables.

**Example 3.8.** As a simple example consider the following polynomial in which the number of multiplications is reduced from 32 to 13:

```
>  S:=x^3+3*x^2*y+x^2+3*x*y^2+2*x*y+2*x^2*z+
   y^3+y^2+2*y^2*z+2*y*z+z^2*x+z^2*y+z^2;
>  convert(S, 'horner', [x,y,z]);
   z^2+((2+z)*z+(2*z+1+y)*y)*y+((2+z)*z+(2+4*z+3*y)*y+
  (2*z+1+3*y+x)*x)*x
```
■

### 3.6.4. Substitution and Elimination

Substitution is defined as replacing a subexpression by a previously computed variable [8]. It reduces complexity of a function by using an additional variable that was not previously in its support set. This transformation creates a new dependency between expressions, but may also eliminate previous dependencies. Substitution has been previously used in multi-level combinational logic optimization [17][18][19]. Elimination theory [7] based on the Gröbner basis formalizes substitution and variable elimination for multivariate polynomials. We refer the interested reader to the reference [7] for the detailed mathematical proof. Note that for arithmetic polynomials, use of a more general decomposition model is necessary as compared to the algebraic division modeled in combinational logic synthesis. This is due to the fact the Boolean idempotence property does not hold in arithmetic polynomials and arithmetic polynomials can have exponents. Therefore, there is no restriction on the support set of the divisor and quotient of an expression. For example $\frac{x^2-y^2}{x-y}=x+y$ is a legitimate division.

Substitution can be combined with THR in order to select a subexpression that maximizes parallelism.

**Example 3.9.** As a simple example let us consider a basic block which consists of two arithmetic expressions:

```
X:= a*b*c+d;

Y:= X+e*f;
```

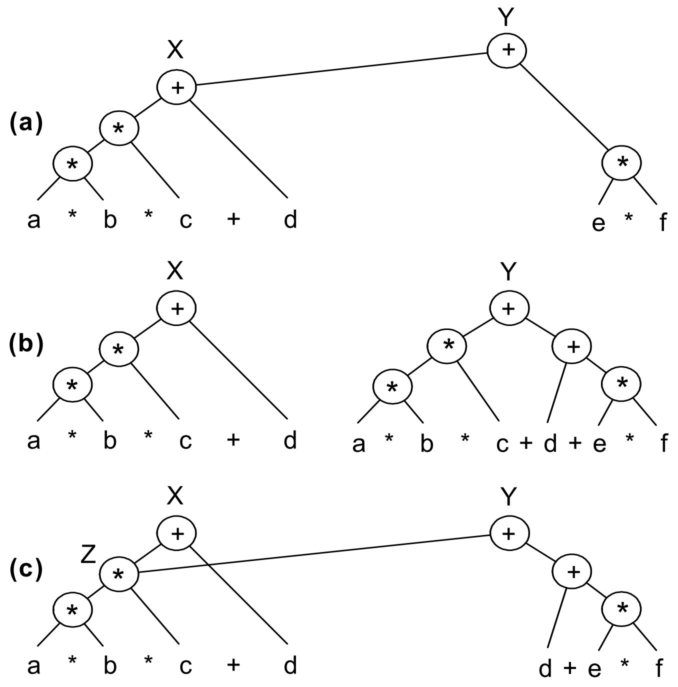


**Figure 3.9. Substitution with THR can Maximize Parallelism**

It can be seen that `Y` is dependent on `X`, therefore `Y` is calculated after the value of `X` is known as shown in Figure 3.9a. However, if we eliminate `X` in `Y`, `Y:=a*b*c+d+e*f`, `Y` can be evaluated in parallel with `X`. Figure 3.9b shows the results of tree-height reduction on both `X` and `Y` expressions. In order to achieve maximum parallelism between `X` and `Y`, we now substitute only subexpression `a*b*c` in `Y` with a new variable `z:=a*b*c`. The result is shown in Figure 3.9c. ■

## 3.7. IMPLEMENTATION AND EXPERIMENTAL RESULTS

SymSyn is an environment that used in conjunction with classical high-level synthesis algorithms can automate efficient synthesis of dataflow intensive circuits. It takes as input a data path of the circuit under design and automatically maps that data path to complex library elements, without need of any directives from the designer. The program inputs are polynomial representations of dataflow and a set of polynomials representing the library elements. Output is a report of components used to implement the dataflow and the way they are connected, such that the critical path delay or number of components used is minimized. SymSyn contains implementations of the algorithms described in Sections 3.4 and 3.5 and the heuristics described in Section 3.6 as accelerators. The implementation is mainly in C programming language with calls to Maple V [2] for the symbolic manipulations.

**Table 3.1. Normalized Delay and Area of Library Elements**

| **Library Element** | **Delay** | **Normalized Delay** | **Area** | **Normalized Area** |
|---|---|---|---|---|
| Add | 7.54 | 1 | 15090 | 1 |
| Square | 7.89 | 1.05 | 89814 | 5.95 |
| Mult | 10.17 | 1.35 | 133401 | 8.84 |
| MAC | 17.28 | 2.29 | 142554 | 9.45 |
| Sine | 45.21 | 6.00 | 625218 | 41.43 |
| Cosine | 45.37 | 6.02 | 622849 | 41.28 |
| SQRT | 21.42 | 2.84 | 36031 | 2.39 |

The efficiency of SymSyn is tested on a number of data-path examples. In these tests, the area and critical path delay reported are normalized by the area and critical path delay of a full adder. For example, the critical path delay of an adder is 1 and critical path delay of a multiplier is 1.35. This number is calculated from the critical path delay reported by Synopsys Design Compiler™ (DC) for a 16-bit multiplier divided by the critical path delay reported by Synopsys DC for a 16-bit adder. The normalized critical path delay calculation is done for all library components available in the Synopsys

DesignWare® arithmetic component library [9]. Normalized area and critical path delay of several library elements are shown in Table 3.1.

Experimental results are shown in Table 3.2. The first four dataflows in Table 3.2 are simple benchmark polynomials. The fifth dataflow polynomial is a basic block of a one-dimensional inverse discrete cosine transform (IDCT). The next dataflow example is the anti-alias block described in the introduction. IDCT and anti-alias are widely used in audio and video compression standards such as JPEG, MPEG, and MP3. The geometric transformation is used in graphics for image rotation. The next three examples come from the field of digital communication. One is a band pass filter in frequency domain. The other performs phase shift keying (PSK) modulation and the third one performs turbo decoding. The last example is a dataflow segment of the Gabor transform used in neural systems.

**Table 3.2. SymSyn Results for Some Examples**

| | **Lexicographical Mapping** | | | **Minimal Component Mapping** | | | **Minimal CPD Mapping** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Dataflow Examples** | **# of comps** | **Area** | **CPD** | **# of comps** | **Area** | **CPD** | **# of comps** | **Area** | **CPD** |
| $a^2$-$b^2$ | 3 | 18.68 | 2.35 | 3 | 10.84 | 2.35 | 3 | 12.90 | 2.05 |
| $b^3$+$ba^2c$ | 6 | 45.20 | 3.70 | 4 | 30.19 | 4.69 | 6 | 39.42 | 3.70 |
| 1-$x0^2$/2+$x0^4$/24+x0+x1x2 | 11 | 65.88 | 5.70 | 3 | 51.72 | 7.02 | 6 | 41.24 | 5.58 |
| Cos(sin(x0)) | 24 | 180.81 | 7.40 | 2 | 82.71 | 12.01 | 9 | 64.88 | 6.43 |
| IDCT | 9 | 63.88 | 4.70 | 2 | 15.40 | 3.34 | 2 | 15.40 | 3.34 |
| anti-alias | 27 | 191.65 | 9.09 | 8 | 60.14 | 14.55 | 12 | 92.61 | 7.04 |
| Geometric-transform | 12 | 82.56 | 10.09 | 2 | 50.27 | 8.29 | 5 | 43.13 | 7.92 |
| 1/2tanh(a-1)+ 1/2tanh(a+1) | 16 | 94.40 | 9.74 | 3 | 24.85 | 5.63 | 4 | 30.80 | 4.38 |
| PSK | 33 | 229.01 | 7.40 | 2 | 42.28 | 7.02 | 2 | 42.28 | 7.02 |
| Turbo decoder | 104 | 817.47 | 16.14 | 4 | 125.3 | 12.99 | 4 | 125.30 | 12.99 |
| Gabor-transform | 79 | 565.10 | 12.44 | 6 | 96.61 | 9.41 | 6 | 96.61 | 9.41 |

In the first set of results of Table 3.2 (lexicographical mapping), it is assumed that the polynomial representation is mapped only to multipliers and adders. This is same as the lexicographical component inference that is typical in commercial behavioral synthesis tools. The number of components column shows the numbers of adds and multiplies in the data-path polynomial. The area reported is the area of an adder multiplied by the number of adds, plus the area of a multiplier multiplied by the number of multiplies in the data-path polynomial. The critical path delay (CPD) reported is the cumulative delay of components on the critical path.

Next, the example dataflows are mapped and synthesized using SymSyn. The second set of results shown in Table 3.2 (*minimal component mapping*), are the results obtained from SymSyn by applying Algorithm 3.4. The mapping reported is the minimal component mapping. We have shown the number of library components Algorithm 3.4 has used in mapping each dataflow polynomial to the extended Synopsys DesignWare$^{®}$ library (DesignWare library [9] plus *tanh(x)*, *ln(x)*, and *exp(x)* operations). Area is the sum of areas of the components used by SymSyn in each data-path implementation.

Finally, the last set of results in Table 3.2 (*minimal CPD mapping*), are derived by SymSyn using Algorithm 3.5. The emphasis is to decompose each dataflow into the given library such that the critical path delay of the implementation is minimized. We have reported the number of components and the area and critical path delay of the implementation suggested by Algorithm 3.5. Note that Algorithm 3.5 maps for maximal parallelism and resource sharing is not used. The critical path delay reported is sum of the delays of components used in the data-path implementation in view of their data dependencies. Both mapping results shown are using the same component library.

In order to qualify the examples used in Table 3.2, the distribution of components used in SymSyn output is shown in Figure 3.10. Note that the components used most are the multiply/accumulate (MAC) operator and the square operator; this result is typical in data-intensive circuits.

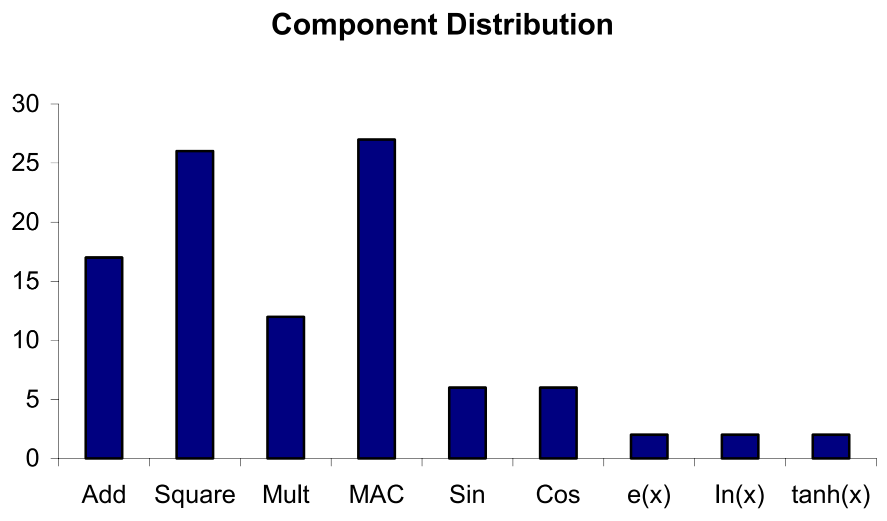


**Figure 3.10. Component Distribution in SymSyn Output**

In order to obtain more precise measurement of the critical path delay and area of our set of examples, Synopsys Behavioral Compiler™ and Synopsys Design Compiler™ are used to produce the set of results shown in Table 3.3. The examples in Table 3.3 are the subset of examples shown in Table 3.2 that did not need *tanh(x)*, *ln(x)*, and *exp(x)* operations. These operations are not available in the DesignWare library [9]. The *lexicographical* columns correspond to results reported by Synopsys Behavioral Compiler™ and Design Compiler™ without any mapping directives in the behavioral HDL code. The *SymSyn mapping* columns are the results reported for the same set of examples when mapping directives suggested by SymSyn are incorporated in the behavioral HDL code. It can be observed that actual performance and area improvements for these examples are inline and better than estimated by SymSyn in Table 3.2.

In summary, the results show that we can achieve an average performance improvement of 25% and an average area improvement of 60% over commercial behavioral synthesis flow. These improvements are the results of intelligent mapping algorithms implemented in SymSyn as opposed to the lexicographical mapping currently available in the commercial tools.

**Table 3.3. Area and Delay Reported by Synopsys Tools Using tsmc.35 Library**

| | **Synopsys BC results** | | | | **Synopsys DC results** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Lexicographical** | | **SymSyn Mapping** | | **Lexicographical** | | **SymSyn Mapping** | |
| **Dataflow Examples** | **Area** | **Est. Delay** | **Area** | **Est. Delay** | **Area** | **Delay** | **Area** | **Delay** |
| $a^2$-$b^2$ | 120295 | 11 | 83087 | 11 | 66760 | 11.21 | 54815 | 9.42 |
| $b^3$+$ba^2c$ | 469030 | 24 | 862816 | 24 | 285926 | 29.09 | 166303 | 25.44 |
| 1-$x0^2$/2+$x0^4$/24+x0+x1x2 | 395252 | 23 | 137139 | 16 | 146526 | 19.68 | 93538 | 14.39 |
| cos(sin(x0)) | 790784 | 39 | 163349 | 35 | 314776 | 38.17 | 140256 | 32.47 |
| IDCT | 456704 | 24 | 178177 | 18 | 323185 | 29.29 | 130753 | 20.52 |
| Anti-alias | 3387672 | 63 | 288373 | 48 | 1761169 | 69.43 | 102357 | 59.89 |
| Geometric-transform | 3051440 | 39 | 273340 | 30 | 1178868 | 54.07 | 190937 | 25.63 |
| PSK | 1812833 | 36 | 82705 | 24 | 1099991 | 33.33 | 80670 | 21.69 |

## 3.8. SUMMARY

This chapter has introduced two new decomposition algorithms to map dataflow to a set of complex arithmetic library components. These algorithms fit seamlessly in the high-level synthesis flow and enhance the quality of result of data intensive circuit synthesis. These methods take advantage of two previously developed concepts; one is the polynomial representation of library blocks and the second is symbolic computer algebra. Polynomial representation is used to represent the functionality of library components and the dataflow segment of the chip under design. Symbolic computer algebra is used to decompose the dataflow to a set of library components. From a practical standpoint, the contribution of this chapter is to make arithmetic library binding an automated process, and eliminate the need for user-specified synthesis directives.

Symbolic computer algebra is a powerful set of algorithms not previously used in the field of synthesis. These algorithms open a new set of opportunities in high-level synthesis research. Even though algebraic manipulations are best suited for combinational arithmetic designs, classical scheduling, resource sharing, and retiming algorithms can be applied to the data-path output to achieve optimized/pipelined designs.

The research presented here is especially promising in the fields of graphics, multimedia, and digital signal processing where there is a tolerance for computational error as long as the degradation in audio or video is limited [24][25][20]. This tolerance can be used to approximate non-polynomials dataflows to polynomial representations, which are well-suited inputs for our tool SymSyn. This chapter does not explain the approximation tools and truncation errors since there is a wide body of mathematical literature available on these topics [11].

# CHAPTER 4
# EMBEDDED SOFTWARE OPTIMIZATION

In embedded system design environment, the software portion of the system tends to change frequently as software changes are generally less costly than hardware changes. Therefore, system-level design tools and methodologies should facilitate embedded software optimization and support software engineering changes. Pre-optimized software libraries and complex processor instructions are often available for embedded system design. Compilers are proficient at optimizations such as dead code elimination, variable propagation, and loop unrolling. Nevertheless, most compilers are unable to use these complex assembly instructions and pre-optimized library elements efficiently while compiling C code for embedded processors. In this chapter, an embedded software optimization methodology is presented that uses the power of symbolic algebra and the dataflow decomposition algorithms used in previous chapters for hardware synthesis.

Currently, software engineers typically design key routines in assembly [21] or manually map a code section to a pre-optimized library element. Example of complex instructions available range from the simple multiply-accumulate (`MAC`) to a library of more complex instructions, such as those developed by Tensilica tools [50][26]. There are several pre-optimized software libraries commercially available. Intel recently released a library targeted at multimedia developers for StrongARM SA-1110™ embedded processor [34], and TI has a similar library for TI'54x DSP [35]. Embedded operating systems typically provide a choice from a number of mathematical and other libraries [36][37]. When a set of pre-optimized libraries is available, the designer has to choose the elements that perform best for a given section of code. For example, consider a section of code that calls the `log` function. The library used in mapping consists of four different `log` implementations: double, float, fixed point using simple bit manipulation algorithm [38], and fixed point using polynomial expansion. Each implementation has a different accuracy, performance, and energy trade-off. A designer would need to estimate which of the four implementations would work best, test the hypothesis, and iterate until the best result is found. Designers are faced with an even more complex problem when attempting to map a software implementation of IDCT already present in MP3 standards code into an embedded software library. There are many ways to implement IDCT on a given processor, and it may be difficult for a designer to determine which library element is most appropriate.

The objective of this research is to improve the quality of compiled code for embedded systems and facilitate the software design process. In this chapter, we propose a new methodology based on symbolic manipulation of polynomials and energy profiling which reduces manual intervention. This methodology automates the process of identifying the code sections that benefit from complex library mapping, and then performs the mapping using symbolic techniques. The set of techniques used in Chapter 3 for algorithmic-level hardware synthesis are combined with energy profiling, floating-point to fixed-point data conversion, and polynomial approximation to achieve a new embedded software optimization methodology. The combination of these tools and standard compiler optimization techniques allow novel automatic code transformations.

**Example 4.1**. As a ***motivating example***, consider the following code segment:

```
for i=1..3
    y = y + cos(i*x);
```

Using standard loop unrolling, the given code is transformed into the following:

```
    y = cos(x) + cos(2*x) + cos(3*x);
```

Now assume that for a given application cos(x) can be approximated into a Taylor series with three terms without noticeable degradation on the output. Many multimedia applications tolerate computational inaccuracy well, as long as the resulting effects (e.g. audio, video degradation) are limited. Therefore, $y$ can be approximated as a polynomial:

$$y = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 + 1 - \frac{1}{2}2^2x^2 + \frac{1}{24}2^4x^4 + 1 - \frac{1}{2}3^2x^2 + \frac{1}{24}3^4x^4$$

This polynomial can be further simplified using the expand routine in symbolic algebra:

$$y = 3 - 7x^2 + \frac{49}{12}x^4$$

Assuming that the embedded processor used to execute this code has a multiply accumulate (MAC) instruction, another symbolic routine called the Horner transform can be used on y:

$$y = 3 + (-7 + \frac{49}{12}x^2)x^2$$

The new equation can be mapped to one multiply instruction and two multiply-accumulates. Obviously, this mapping is much more efficient than three calls to the cosine library function. Unfortunately, to our knowledge, there is no available software optimization tool that performs this simple optimization automatically. Thus, it would be up to designers to manually implement such optimizations. ■

This chapter presents an algorithm and methodology, called SymSoft, that performs algebraic manipulations such as the ones shown in Example 4.1 simultaneous with automatic complex instruction and library mapping. First, a characterization function is derived for the pre-optimized library elements and complex assembly instructions. Then, the performance and energy critical code sections are identified using the energy profiler. If necessary, a tool such as Fridge [24] can be used to help transform floating-point data types into fixed-point. Next, complex nonlinear arithmetic functions in critical blocks are approximated as polynomials such that the final output is within the acceptable tolerance limits. Finally, symbolic algebra is used to map the polynomial representations of the critical basic blocks to the instruction set and library elements available automatically such that performance and power consumption are optimized.

This chapter is organized as follows: Section 4.1 discusses previous work in software optimization for energy and performance. Section 4.2 describes the software and hardware platform and the measurement setup we are using in our experiments. Section 4.3 presents the SymSoft flow, and gives an overview of each of its steps and components. The results of SymSoft optimizations on several software examples for the SmartBadgeIV system are presented in Section 4.4. SymSoft lowers the execution time and energy consumption of these examples by using a pre-optimized software library available for StrongARM and the StrongARM instruction set. Finally, Section 4.5 summarizes the contributions of this work.

## 4.1. RELATED WORK

Designers have used software performance and size optimization methodologies and tools of for many years. Generally, compilers are used to translate a high-level specification into optimized machine code for a target processor. Several researchers have worked on optimizing compilers in last few years [27]. Prototype research compilers have shown impressive results [28]. Most optimizing compilers target high-performance and/or general-purpose computers. Relatively little effort has been dedicated to create powerful optimizing compilers for embedded processors. Several researchers are studying automatic code retargeting techniques for embedded processors [29][30] using graph-covering methods. Graph covering methods have limited knowledge of algebra. Using algorithms from symbolic algebra, as explained in this chapter, enables simultaneous code generation and algebraic manipulations. Currently, most embedded processors (or DSPs) are programmed directly by expert programmers and code optimization is mostly based on human intuition and skills. In addition, due to recent growth in market demand for portable devices, optimization of software for power consumption is gaining importance. As a result, one of the primary requirements for system-level design methodology of embedded devices is to effectively support code performance and energy consumption optimization.

Several optimization techniques for lowering energy consumption have been presented in the past. Numerous methodologies for optimizing memory accesses have been introduced that combine automated and manual software optimizations [31]. Tiwari et al. [32][33] used instruction-level energy models to develop compiler-driven energy optimizations at assembly level such as instruction reordering, reduction of memory operands, operand swapping in the Booth multiplier, efficient usage of memory banks, and a series of processor specific optimizations. Several other optimizations such as energy efficient register labeling during the compile phase [39], procedure inlining and loop unrolling [40] as well as instruction scheduling [41] have also been suggested. In addition, various compiler optimizations have been applied concurrently and the resulting energy consumption was evaluated via simulation [42]. All of these techniques focus on automated instruction-level optimizations driven by the compiler. Unfortunately, current available compilers have limited capabilities. Specifically, they are incapable of handling arithmetic optimizations such as shown Example 4.1.

In the previous work [49], MP3 audio decoder software available from the standards body [23] was manually optimized for the SmartBadge embedded system [22]. This work required the designer to first implement a fixed-point library and then to replace all floating-point operations with fixed point. Then, the designer needed to fully understand the details of the SmartBadge's design, so that the critical arithmetic operations can be manually optimized with inline assembly code. The manual optimization process lasted several days. This experience is similar to the typical industrial settings, where the software needs to be ported and optimized to the newer versions of hardware.

The proposed methodology and tool flow uses profiling to identify the code sections that would benefit most from algebraic optimizations, and then automatically performs the optimizations using symbolic techniques. Such symbolic techniques have been previously used in algorithmic level synthesis of data intensive circuits as described in Chapter 3. SymSoft uses the same principles previously used for high-level component mapping of hardware and applies them to the software optimization problem. The outcome of our mapping algorithm is software that runs faster and consumes less energy on the SmartBadgeIV [22] embedded system while compared to the output of the commercial StrongARM compiler.

## 4.2. EXPERIMENTAL SETUP

SymSoft is used to optimize a set of examples on the SmartBadgeIV [22]. SmartBadgeIV, as shown in Figure 4.1, is an embedded system powered by batteries through a DC-DC converter. It consists of StrongARM SA-1110™ processor with StrongARM SA-1111™ companion chip, audio CODEC with microphone and speakers, Lucent's WLAN card, sensors and three types of memory: SRAM, SDRAM and FLASH. SmartBadgeIV currently runs eCos [36] and an embedded version of the Linux operating system [37]. In this work, the Linux operating system was used since the software library available to us is implemented for Linux. SmartBadgeIV 's Linux has the main parts of the operating system, including a small file system, residing in the SRAM. The larger file system is remotely mounted from the server via the WLAN card. In our experiments, the program files and their input data reside in the directory structure on the server. These files are accessed via the wireless link on the SmartBadgeIV.

All of the measurements were performed using National Instruments Data Acquisition (DAQ) measurement system capable of 1.25 Msamples/second. We found a sampling speed of 1 kHz to be sufficient. In our setup, we used one PC to measure system, processor, and WLAN currents via the DAQ interface, and the other PC to act as a remote file server for the SmartBadge IV. The execution time of the code was measured by accessing StrongARM SA-1110™ on-board timer.

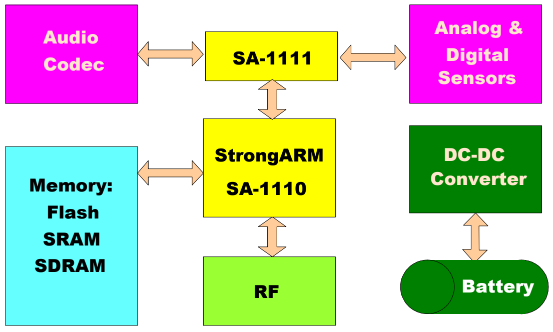


**Figure 4.1. SmartBadgeIV Architecture**

## 4.3. SYMSOFT METHODOLOGY AND TOOL FLOW

Ideally, the software designer would write an algorithmic-level description of the software and have a compiler-like tool optimize it for the given hardware platform.

However, optimum implementation of calculation intensive routines for the particular hardware design is not possible with traditional compiler optimizations alone. Commonly, the designer does most of such optimizations by hand. Automating even a portion of this process can save much design time.

We present a methodology and a tool flow, SymSoft, which facilitates embedded system software optimization with automating library and complex instruction mapping for a given embedded processor. Figure 4.2 shows the SymSoft flow. The mapping methodology consists of three main steps: library characterization, target code identification, and mapping.

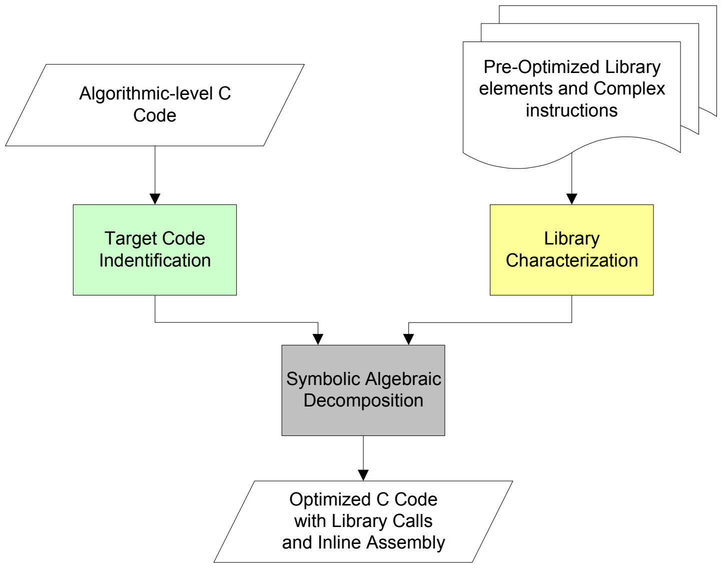

Algorithmic-level C Code
Pre-Optimized Library elements and Complex instructions
Target Code Indentification
Library Characterization
Symbolic Algebraic Decomposition
Optimized C Code with Library Calls and Inline Assembly

**Figure 4.2. SymSoft Tool Flow**

The first step is to characterize the library elements. The characterization not only includes performance and energy consumption of the complex element for a given hardware architecture, but also the expected input and output format, accuracy and a polynomial representation.

The next step identifies the target code for optimization. In this step, an initial check is performed to see whether data representation used in the algorithmic-level C code matches the target hardware. Most embedded processors support only fixed point computation, but many multimedia algorithms utilize floating-point operations. The profiler, described in Section 4.3.2.2, detects if data representation is an issue within several seconds. Then, if needed, floating-point operations are replaced with fixed-point operations with the help of a floating-point to fixed-point converting tool [20][24][25]. The profiler also reports the performance and energy critical functions of the code. The polynomial representations of the arithmetic sections of the critical routines are calculated with help of traditional compiler techniques such as loop unrolling. When necessary, polynomial approximation techniques are used. Accuracy is checked at the end of the target code identification step to make sure that the code still meets the specifications, as some rounding occurs both during the data representation conversion and during the polynomial formulation.

Finally, the target code represented by polynomials is automatically mapped into the library elements and complex processor instructions. The key contribution in SymSoft is a new method to map critical code segments into pre-optimized software library elements and complex assembly instructions using symbolic polynomial manipulation. The mapping process selects the solution that offers best performance with sufficient accuracy. Since the SymSoft methodology is compliant with other software optimization techniques, additional benefits are gained by combining it with traditional complier optimization algorithms, such as constant and variable propagation, dead code elimination, and loop unrolling. The next sections describe each part of the SymSoft flow in detail.

### 4.3.1. Library Characterization

The target library consists of pre-optimized software libraries and complex arithmetic instructions available for the target processor. Complex arithmetic instructions vary from the simple multiply-accumulate (`MAC`) to more complex instructions, such as those developed by Tensilica tools [26]. Pre-optimized software libraries include traditional embedded system libraries, such as the IEEE floating-point mathematical library for Linux operating system [37], commercial libraries available for the particular processor, such as Intel's integrated performance primitives (IPP) [34], and a set of in-house pre-optimized routines. Table 4.1 shows a sample of elements of the IPP library. Library characterization is done on element-by-element basis. Each element is labeled with the type of inputs and outputs, performance, accuracy, energy consumption, and finally its polynomial representation.

**Table 4.1. Sample of IPP Library Elements**

| Library Elements | Description |
|---|---|
| Exp | Exponentiation |
| Ln | Natural logarithm |
| DotProd | Vector dot (inner) product |
| Mean | Vector arithmetic mean |
| FIR | Finite impulse response filter |
| IIR | Infinite impulse response filter |
| Conv | Convolution |
| WinHamming | Hamming window |
| FFT | Fast Fourier transforms |
| HuffmanDecode | Decodes Huffman symbols |
| SubBandSynthesis | Stage two of hybrid synthesis filter bank |
| IMDCT | Inverse modified discrete cosine transform |

The format of library element inputs and outputs is determined from the library include files or documentation available with the library element. Techniques discussed in Section 4.3.2.3 can be used to extract the polynomial representations from the source code if the code is available. Otherwise, either the distributor needs to provide the equivalent polynomial representation or it might be obtained from the documentation. Important part of library characterization is the determination of accuracy, performance and energy consumption. This information is used to guide the selection process when more than one library element has same functionality. Most embedded systems have operating system timers that can be used for fine-granularity performance measurements on hardware. However, often there is not an easy way to measure processor and memory power consumption. Alternatively, a cycle-accurate energy consumption simulator [44] easily provides energy and performance estimates of library elements. Note that the library characterization step is yet to be automated.

Examples of two characterized complex library elements, SubBand Synthesis and IMDCT, are shown in Table 4.2. The library has three different versions of each library element: the open-source floating point version from the MP3 standards library [23], fixed-point in-house pre-optimized routine, and a version from Intel's integrated performance primitive (IPP) library for StrongARM SA-1110™ processor [34]. For each library element, we have measured its performance on the SmartBadgeIV hardware. All entries in Table 4.2 are represented using polynomials. Since polynomials for complex library elements can be quite large, we show only a critical portion of IMDCT polynomial in Equation 1. Equation 1 shows how *n/2* windowed samples, $y_k$, are transformed into *n* $x_i$ samples. Note that this is just a first order polynomial, since $\cos(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1))$ can be calculated in advance for all *i*, *k* and *n*.

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} y_k \cos(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1)) \qquad \textbf{(1)}$$

**Table 4.2. Characterized Complex Library Elements**

| Library Element | Execution time | Input Type |
|---|---|---|
| float SubBandSyn | 0.95 | 64 bit float |
| fixed SubBandSyn | 0.01 | 32 bit fixed |
| IPP SubBandSyn | 0.002 | 32 bit fixed |
| float IMDCT | 0.39 | 64 bit float |
| fixed IMDCT | 0.014 | 32 bit fixed |
| IPP IMDCT | 0.0002 | 32 bit fixed |

### 4.3.2. Target Code Identification

The input to the target code identification step is the algorithmic-level C code of the embedded software. The output of this step is a set of polynomial representations of the critical code segments that would benefit most from mapping to complex instruction and pre-optimized library elements. Target code identification consists of three stages as shown in Figure 4.3. First, the profiler checks to see whether floating point operations are on the critical path. If needed the floating-point operations are transformed into fixed-point operations by data representation conversion. Next, the energy and performance critical procedures are identified. This step can be done either with simulation using the energy profiler [44] or by profiling directly on the hardware. Finally, when the power and performance critical procedures are identified, they are formulated as polynomials suitable for mapping into library elements. In the next sections, we will take a closer look at each stage of the target code identification step.

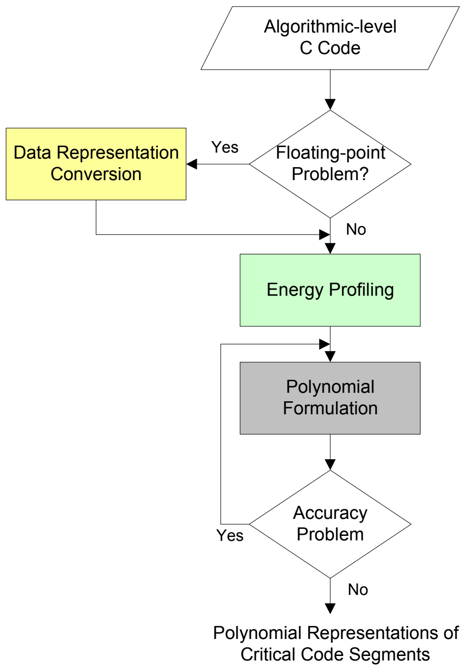


**Figure 4.3. Target Code Identification**

4.3.2.1. Data Representation Conversion

Signal processing algorithms are generally developed using ANSI-C with IEEE floating-point data types. However, these algorithms are often implemented in embedded systems using fixed-point data types in order to meet the power, cost, and performance requirements. In this stage, it is checked whether floating-point operations are capturing most of the execution time and power consumption of the algorithmic-level C code. In that case, floating-point operations are considered critical and they must be converted to fixed-point operations. Converting a floating-point algorithm to a fixed-point algorithm is a time consuming and error prone task. Facilitating and semi-automating this conversion has been the target of many research projects [20][24][25]. Such tools use interpolative analysis or analytic techniques to convert floating-point operations into appropriate fixed-point operations while reducing the manual work and the number of simulations required. In our tool flow, we opt to use a tool like Fridge [24] (a.k.a. CoCentric fixed-point designer) to automate this stage of optimization.

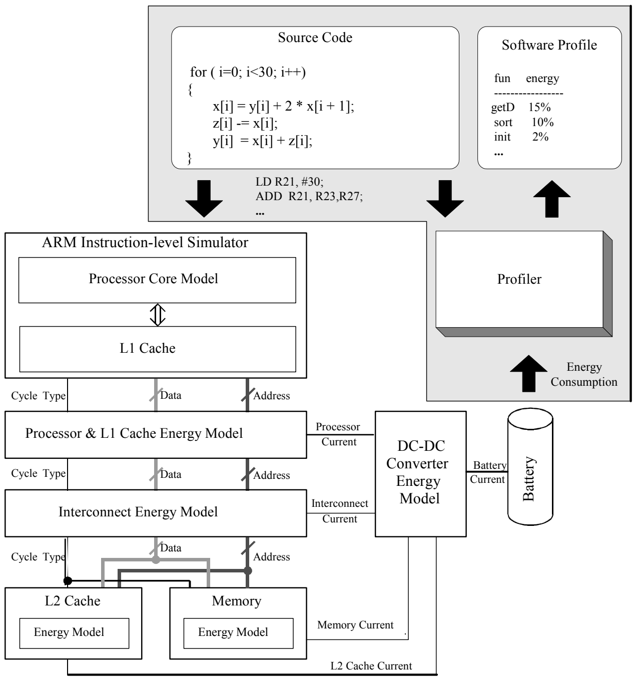
#### 4.3.2.2. Energy Profiling

Code optimization requires extensive program execution analysis to identify performance and energy-critical bottlenecks and to provide feedback on the impact of code transformations. Profiling is typically used to relate performance to the source code for CPU and L1 cache [43]. An energy profiler enables easy identification of energy-critical procedures. It also facilitates analysis of code transformations' impact on the processor energy consumption, the memory hierarchy, and the system busses.

The profiler exploits a cycle-accurate energy consumption simulator [44] to relate the embedded system energy consumption and performance to the source code. Thus, it can be used for analysis (i.e., to find energy-critical sections of the code), and for validation (i.e., to assess the impact of each code optimization).

The profiler architecture [44] is shown in Figure 4.4. Source code is compiled using a compiler for a target processor. The output of the compiler is the executable represented as assembly code and a map of locations of each procedure in the executable. The profiler of the cycle-accurate simulator periodically samples the simulation results (by user defined sampling interval) and maps the energy and performance to the function executed using information gathered at the compile time. Sampling is used to improve profiling speed while maintaining accuracy. Once the simulation is complete, the energy consumption and execution time of each function are displayed.

With the profiler, SymSoft can obtain energy consumption breakdown by procedures in the source code and thus can quickly identify the sections of the source code whose optimization can provide the largest execution time and energy savings. In addition, with the cycle-accurate simulator that is at the heart of the profiler, SymSoft can get detailed information about performance and energy consumption of smaller subsections of code. The identified critical code segments are then passed as inputs to polynomial approximation and symbolic mapping tools that can optimally map the code section into complex library elements and assembly instructions in few minutes.



```
for ( i=0; i<30; i++)
{
      x[i] = y[i] + 2 * x[i + 1];
      z[i] -= x[i];
      y[i]  = x[i] + z[i];
}
```

```
LD R21, #30;
ADD  R21, R23,R27;
...
```

```
fun    energy
-----------------
getD   15%
sort    10%
init     2%
...
```

**Figure 4.4. Profiler Architecture**

4.3.2.3. Polynomial Formulation

Our goal is to automatically map the critical code segments selected by the profiler into pre-optimized library elements or complex assembly instructions such that optimum execution time and power consumption are achieved. The symbolic mapping algorithm, described in Section 4.3.3, takes as input the polynomial representations of the critical code segments and the polynomial equivalence of complex arithmetic assembly instructions and pre-optimized library elements. The polynomial formulation step prepares the first set of inputs required by the symbolic mapping algorithm by calculating the polynomial representations of the critical code segments. The second set of inputs is calculated in the library characterization step as described in Section 4.3.1.

The polynomial representation of a basic block can be directly extracted from the C code if the basic block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [46][47] can be used to formulate the equivalent polynomial representation. When the basic block implements a transcendental function, we use an approximation, such as the Taylor or Chebyshev series expansion, as its polynomial. The chosen polynomial approximation has to be verified by simulation to ensure that the software constraints, such as audio quality, are satisfied. A good approximation can result in large performance and power improvements for multimedia applications, since these applications can tolerate a slight degradation in the output. For example, to verify the accuracy of the MP3 decoder we have used the compliance test provided by the MPEG standard where the range of RMS error between the samples defines the compliance level [45]. If the approximation is not sufficient to satisfy the accuracy constraints, the quality of approximation is changed and verified again through simulation.

The objective of this step is to formulate polynomials that cover as much of the source code as possible. Consecutively, the likelihood of finding a more complex library element that matches at least a portion of the formulated polynomial increases. This objective can be accomplished by using code transformation techniques such as loop unrolling, constant and variable propagation to form larger basic blocks.

### 4.3.3. Symbolic Mapping Algorithm

The symbolic mapping algorithm requires two sets of inputs: a set of polynomials representing the critical code segments and another set of polynomials representing the pre-optimized library elements and complex instructions. The former has been generated in the target code identification step and the latter is the output of the library characterization step. The goal of the symbolic mapping algorithm is to decompose the polynomial representations of the critical code segments (CCS) into the polynomial representations of the target library such that execution time and power consumption are minimized. The power consumption and execution time of each library element are provided to the mapping algorithm as constants by the library characterization step as described in Section 4.3.1. As opposed to tree covering based algorithms, in our algorithm, mapping is performed simultaneously with algebraic manipulations.

The symbolic mapping algorithm uses multivariate polynomial manipulation algorithms from symbolic computer algebra. The theory behind these algorithms is described in Chapter 2. Namely, symbolic techniques used are factorization, expansion, Horner transform, multivariate polynomial substitution, and variable elimination. In this section, these routines are described by a set of simple examples.

**Example 4.2.** ***Factor*** and ***expand*** are inverse operations. Consider using Maple to factor and expand the following polynomial:

```
>  S := x^2*(x^14+x^15+1);
>  P := expand(S);
   P = x^16+x^17+x^2
>  factor(P);
   x^2*(x^14+x^15+1)
```
■

**Example 4.3.** ***Horner*** form of a polynomial is a nested normal form with minimal number of multiplications and additions. Any polynomial can be rewritten in Horner, or nested, form. An example of Horner form polynomial for multiple variables is shown below:

```
>  S:= y^2*x+y*x^2+4*x*y+x^2+2*x;
>  convert(S, 'horner', [x,y]);
   (2+(4+y)*y+(y+1)*x)*x
```
■

**Example 4.4.** ***Simplify*** implements substitution and variable elimination for multivariate polynomials:

```
>  S:= x + x^3*y^2 -2*x*y^3;
>  simplify(S, {p = x^2-2*y}, [x,y,p]);
   x+y^2*x*p
```
■

The core of the library-mapping algorithm is the simplification modulo set of polynomials (*simplify*) routine. The polynomial representations of critical code blocks are simplified modulo a subset of polynomials representing the library elements called the side relation set. Choosing the side relation set is a non-trivial and important task, especially since different side relation sets results in different solutions. In Chapter 3, an algorithm was introduced that selects the side relation set such that the hardware implementation of a (portion of) data path with a given component library has minimal critical path delay. In this chapter, a similar algorithm is used to optimize the execution time of the critical code segments of software by mapping to pre-optimized library elements and complex assembly instructions. Since evaluating all subsets of the library is exponentially expensive, the library-mapping algorithm uses the branch-and-bound method with execution time and energy consumption as bounding functions to prune the search space. All previously described symbolic manipulations except *simplify* are used as guidelines in formulating different side relation sets to speed up the mapping algorithm.

Figure 4.5 gives an overview of the mapping algorithm. Inputs to the algorithm are the polynomial representations of the critical code segments (CCS) and the polynomial representations of the target library elements. Initially, tree-height reduction, factorization, expansion, and Horner-based transform are applied to the polynomial representation of the CCS resulting in several different polynomials representing the same code segment. Each of the different polynomial representations is used to select a side relation from the target library. These guidelines are used to increase the speed of finding the desirable mapping. The polynomial representation of the CCS is simplified modulo the selected side relation sets in parallel. If the result of *simplify* matches a library element then the CCS is mapped. Otherwise, we need to continue to add to the side relation set until the CCS is fully mapped to our library. The iterative part of the algorithm, denoted in Figure 4.5 as *main loop*, is implemented using branch-and-bound algorithms.

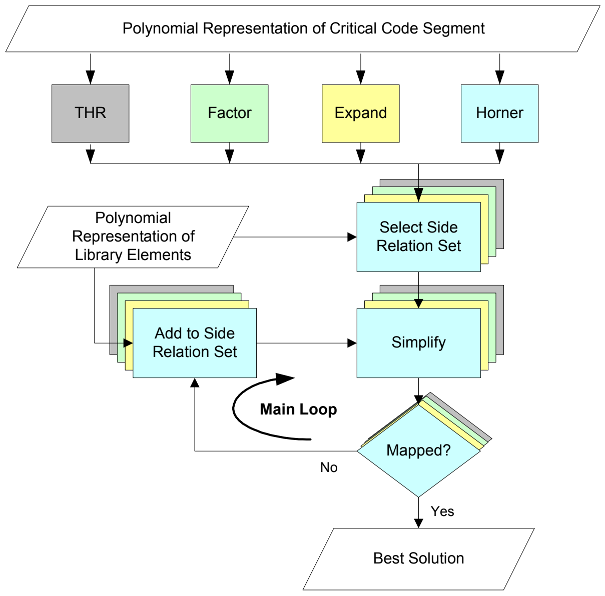


**Figure 4.5. Overview of the Library Mapping Algorithm**

Algorithm 4.3.3 shows the pseudo-code of the library-mapping algorithm. Inputs to this algorithm are the polynomial representation of the critical code section (*CCS*) and the polynomial representations of the library elements (*L*). The bounding function is defined as the best execution time for *CCS* seen so far. The lower bound computed at each decision branch is the execution time of the library elements in the side relation set in view of data dependencies. If this lower bound is greater than the best execution time seen so far, the corresponding decision branch is pruned. Decision tree (*decision_tree*) implements the branch-and-bound algorithm. The algorithm starts by initializing the root of *decision_tree* to the polynomial representation of *CCS* and calculating an initial bound. The bounding variable is initialized to the execution time of calculating the *CCS* polynomial solely with add and multiply instructions, the lexicographical mapping (*LexMap*). Nodes are added to this tree in breadth-first manner. These nodes store the polynomial result of *simplify* of their parent node and the chosen side relation set. When a simplification result corresponds to a polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the execution time of the solution is less than previously encountered solutions, we set the bounding variable to the current value. In case the simplification result stored in a tree node does not correspond to any library elements, we apply the same steps to the new tree node until either a solution is found or the corresponding branch is pruned. Since *CCS* is a polynomial and add and multiply instructions are always available in our library, we are guaranteed to have a solution. However, our mapping algorithm searches for a solution that best exploits the given software library.

**Algorithm .4.3.3.** Decompose *CCS* into elements of library *L*

```
function Decompose (exp_tree, boundVal, L) {
  // initialize the decision tree
  decision_tree ← tree (exp_tree)
  Depth ← 0
  Bound ← boundVal
  for all n ∈ decision_tree with depth == Depth  do{
      if Depth == 0
            choose sr ∈ L to preserve the exp_tree structure
       else for all sr ∈ L {
            result = simplify (n, sr);
            AddChild (n, result)  // make result a child of node n
            if result ∈ L   // solution is found
               Bound = Min(cost of node result, Bound);
       }
        if no more n ∈ decision_tree with depth == Depth
           Depth ← Depth + 1
 }
 return the best solution
end Decompose
procedure main (CCS,L)
  exp_tree [1 .. NoManipulations] = AllManipulations (CCS);
  for i = 1 to NoManipulations {
     boundVal[i]=LexMap(exp_tree[i]);
     solution[i] = Decompose(exp_tree[i],boundVal[i]) }
  return the best solution in solutions[i]
end main
```

The branch-and-bound algorithm in Algorithm 4.3.3 is applicable to most practical problems and its runtime is in the matter of minutes. Nevertheless, as for all branch-and-bound algorithms, the worst-case complexity remains exponential. The speed of this algorithm depends on the initial polynomial and the initial side relation set. Here, we use a set of library independent symbolic manipulations on the original *CCS* polynomial to help with the selection of initial side relation element. These manipulations improve the execution time without hampering the quality of the solution. First, we apply tree-height reduction, factorization, expansion, and Horner-based transform to *CCS* in the *AllManipulations* function. As a result, we have several different polynomials (*exp_tree*) representing the same code section. Each of these representations can result in the desirable implementation based on the available library elements.

To select the initial member of side relation sets, we start with the primary inputs and cover the expression tree with the library elements. We choose all library elements that cover the primary inputs and a portion of the expression tree as initial elements of the different side relation sets used to simplify the root of the *decision_tree*. If the result of simplify is not a library element, we add more elements to the side relation set without further guidance from the expression tree and decompose the result. Note that in selecting the side relations from the library, all different permutations of the variables with the same data-type are considered. This algorithm is implemented in C with calls to Maple V for the symbolic manipulations.

**Example 4.5.** In order to demonstrate the power of our ***library mapping*** algorithm, consider a basic block implementing Equation 2:

$$d = \cos\left(\frac{\pi}{72}\left(2p + 1 + \frac{N}{2}\right)(2m + 1)\right) \qquad \textbf{(2)}$$

Equation 2 is approximated using Pade approximation to the polynomials shown in Equation 3 in the previous step of the SymSoft flow as described in Section 4.3.2.3.

$$x=\frac{\pi}{72}(2p+1+\frac{N}{2})(2m+1)$$
$$d\cong\frac{1-\frac{3665}{7788}x^2+\frac{711}{25960}x^4-\frac{2923}{7850304}x^6}{1+\frac{229}{7788}x^2+\frac{1}{2360}x^4+\frac{127}{39251520}x^6} \quad \textbf{(3)}$$

The simplification modulo set of polynomials routine can be used to map the numerator and denominator of Equation 3 to the available instruction set. Let dn be the numerator of Equation 3 with a, b, and c the constants of the polynomial. In addition, we define siderels as a subset of the available instructions with renamed variables. We have:

```
> dn:=1+a*x^2+b*x^4+c*x^6: siderels:={w=x^2,y=b+c*w,z=a+y*w};
> simplify(dn, siderels,[x,w,y,z]);
  1+z*w
```

Note that the first element of the side relation set (w=x^2) corresponds to the square or multiply instruction and the other two elements of the set (y=b+c*w, z=a+y*w) and the result of simplify (1+z*w) correspond to the MAC instruction. The side relation set can be any subset of the available instruction set with proper renaming of the variables. Different side relation sets result in finding other possible solutions for the specification. The above implies:

```
dn=1+a*x^2+b*x^4+c*x^6=1+z*w
    =1+(a+y*x^2)*x^2=1+(a+(b+c*x^2)*x^2)*x^2
```

Therefore, the numerator of Equation 3 can be mapped to one square and three MACs instructions. Assuming R1, R2, R3, R4, and R5 hold 1, a, b, c, and x, respectively, the resulting assembly code is:

```
MULT R6, R5, R5
MAC  R7, R3, R4, R6
MAC  R8, R2, R7, R6
MAC  R7, R1, R8, R6
```

In the MP3 decoder program, the basic block evaluating Equation 2 uses floating-point and takes 2124 cycles to run on the StrongARM SA-1110™ processor. The approximation represented in Equation 3 calculates `x` using floating-point and `d` using fixed-point arithmetic and nested MACs as suggested by the symbolic optimization. This approximation executes in 901 cycles. Thus, we have achieved an improvement of 57% for this simple example. The fixed-point version with no symbolic optimization executes in 1367 cycles. Thus, approximately 50% of the improvement achieved is due to use of fixed-point arithmetic and 50% is due to smarter use of processor instructions. ■

## 4.4. RESULTS

We have tested the effectiveness of SymSoft using the experimental embedded system SmartBadgeIV and a wide range of code examples used in communication, digital signal processing, and streaming media. The SmartBadgeIV system and our experimental setup for hardware execution time and energy consumption measurement were described in Section 4.2.

The first six software examples are obtained from a DSP software benchmark suite [48]. The first two examples are software programs that perform common digital signal processing computations; discrete convolution and dot (inner) product. Convolution is the linear operator can compute the output of a linear time-invariant (LTI) system in response to an input sequence given the system impulse response sequence. The dot (inner) product of two vectors is the summation of the products of the two input sequences; i.e. $z = \sum_i x[i] \cdot y[i]$.

The next four examples are different digital filters used in digital signal processing and communication applications. The first filter is a finite impulse response (FIR) filter. The next two filters are biquad infinite impulse response (IIR) filters. A single IIR filter of arbitrary order is often decomposed into equivalent cascades of 2nd-order IIR sections known as biquads. Although the biquad cascade is analytically identical to the single filter of higher order, the biquad filter realization is more stable and less sensitive to quantization errors. The last filter is a least-mean-square (LMS) FIR adaptive filter. The LMS filter is a time-varying linear system for which the filter coefficients are adjusted at each time step to minimize the error between the actual output and a given desired output.

Finally, the last example is a full MPEG Layer III (MP3) audio decoder implementation that streams MP3 encoded files from a server to a client (SmartBadgeIV).

**Table 4.3. Results of SymSoft Optimization on a Set of Examples**

| | **Execution time in microsecs** | | |
|---|---|---|---|
| **Examples** | **Original** | **SymSoft** | **improvement (%)** |
| Convolution | 667 | 627 | 6.01 |
| Dot product | 358 | 267 | 25.42 |
| FIR filter | 2418 | 1170 | 51.61 |
| IIR filter (4 biquads) | 5079 | 4355 | 14.25 |
| IIR filter (1 biquad) | 1396 | 1250 | 10.46 |
| Least Mean Square | 1200 | 1000 | 16.67 |
| MP3 decoder | 5470000 | 1430000 | 73.86 |

Table 4.3 summarizes the results of applying SymSoft tool flow to the set of examples discussed above. In each case, we start with the fixed-point implementation of the algorithm and use profiling to select the critical code sections. Optimizing a critical code section results in noticeable improvement on any given example. Next, the critical code sections are automatically mapped to the instruction set available on the StrongARM SA-1110™ processor and Intel's integrated performance primitives (IPP) library for StrongARM SA-1110™ processor [34]. Table 4.3 shows the execution time of each example before and after the optimization with SymSoft. Note that the original execution time column, reports the execution time of the examples when all possible optimizations available with the ARM compiler are used.

The improvements demonstrated in Table 4.3 indicate that by using SymSoft we can obtain significant execution time improvement for a range of applications over commercial compilers. The amount of improvement achieved is dependent on the number of critical blocks that are optimized and the library implementations available for the given block. Examples in Table 4.3 show improvements in the range of 6% to 73% with an average of 28% improvement.

In the next section, we will go through all the steps of the SymSoft flow using the MP3 decoder software as an example.

### 4.4.1. The MP3 Optimization Results

We start with an algorithmic level description of the MPEG Layer III (MP3) audio decoder obtained from the International Organization for Standardization (ISO) [23]. Our design goal is to accelerate the MP3 decoder and lower its energy consumption while keeping full compliance with the MPEG standard. The first step in decoding the MP3 stream is synchronizing the incoming bitstream and the decoder. Huffman decoding of the SubBand coefficients is performed before requantization. Stereo processing, if applicable, occurs before the inverse mapping which consists of an inverse modified discrete cosine transform (IMDCT) followed by a polyphase synthesis filterbank. During the optimization process, we used instructions available on the StrongARM SA-1110™ processor, a mathematical library available with Linux operating system [37], Intel's integrated performance primitives (IPP) library for StrongARM SA-1110™ processor [34], and a library populated with in-house pre-optimized routines. The library elements ranged from simple mathematical functions such as `MAC` to as complex elements as `IMDCT` routine.

The SymSoft flow, as described in Section 4.3, consists of library characterization, target code identification, and the final library mapping step. The library characterization step uses hardware measurements for performance and simulations for energy consumption [44]. The polynomial representation is obtained either from the source code (Linux mathematical and in-house libraries), or from documentation (IPP library).

The target code identification consists of three important steps: data type conversion, code profiling, and formulating polynomials to be mapped. The first step is to check if floating-point data types are suitable for the given platform. Since SmartBadgeIV 's processor, StrongARM SA-1110™, can only emulate the floating-point operations, there is a need for data representation transformation. The code was converted to use fixed-point arithmetic. It was verified through simulation that 27-bit precision fixed-point data-types are sufficient to meet the compliance test provided by MPEG standard [45]. Automating floating-point to fixed-point data type conversion has been targeted by the tool Fridge [24]. Profiling the original source code highlights the critical code segments. Table 4.4 shows the results of profiling original MP3 decoder software we obtained from the standards body. All profiling reported in Table 4.4 is using hardware measurements. The results are shown for one frame and represent only the most significant functions in terms of their performance impact. Next, we formulate equivalent polynomial representation of each of the critical functions shown in Table 4.4. We use polynomial approximations for the non-linear calculations in the critical basic blocks. Once more, we validate that these approximations satisfy the MPEG compliance test [45]. The output of the target code identification step is a set of polynomials representing the critical sections of the code.

**Table 4.4. Profiling the Original MP3 Code**

| **Function name** | **Execution time (s)** | **%** |
|---|---|---|
| III_dequantize_sample | 1.1754 | 45.33 |
| SubBandSynthesis | 0.9481 | 36.56 |
| Inv_mdctL | 0.3872 | 14.93 |
| III_hybrid | 0.0670 | 2.58 |
| III_antialias | 0.0131 | 0.51 |
| III_stereo | 0.0010 | 0.04 |
| III_hufman_decode | 0.0007 | 0.03 |
| III_reorder | 0.0005 | 0.02 |
| Total for one frame | 2.5931 | 100.00 |

In the first phase of optimization, the polynomial representations of the critical code sections of the first three function shown in Table 4.4 are mapped into the StrongARM assembly instructions by algorithm described in Section 4.3.3. It is important to note that StrongARM compiler was not capable of using the MAC instruction effectively. However, our symbolic algorithm was able to use this instruction efficiently. Automatically generated inline assembly was inserted in the C code as the result of the decomposing algorithm. The results of optimizing critical functions of the MP3 code by SymSoft are compared with the original results from straightforward compilation in Table 4.5. The numbers reported in Table 4.5 are obtained using the cycle accurate energy simulator described in Section 4.3.2.2. The first, third, and fifth row in Table 4.5 correspond to the first three rows of Table 4.4. The second, fourth, and sixth row in Table 4.5 are functions related to the function in the previous row. As we can see, 12-70% improvement has been achieved using the SymSoft methodology. Such improvement was previously possible only thorough manual optimization with inline assembly. The automation introduced by SymSoft drastically reduces the embedded software optimization cycle.

**Table 4.5. Comparing SymSoft Instruction Mapping and a Commercial Compiler**

| | **Execution time (#cycles)** | | | **Energy Consumption (mWhr)** | | |
|---|---|---|---|---|---|---|
| **Function** | **original** | **optimized** | **%imp** | **Original** | **optimized** | **%imp** |
| III_dequantize_sample | 650894 | 421976 | 35.2 | 0.940 | 0.747 | 20.5 |
| PowThreeFourth | 14135 | 5380 | 61.9 | 0.040 | 0.009 | 76.6 |
| SubBandSynthesis | 155204 | 70633 | 54.5 | 1.015 | 0.306 | 69.8 |
| generateFilterS | 5263831 | 4196853 | 20.3 | 3.630 | 3.319 | 8.6 |
| Inv_mdctL | 63583 | 31954 | 49.7 | 0.101 | 0.051 | 49.6 |
| generateMDCTTable | 1454550 | 957051 | 34.2 | 1.051 | 0.922 | 12.2 |

Next, we profile the MP3 decoder that results from this phase of optimization on the hardware and measure the execution time of each function while decoding one frame of the MP3 stream. The resulting performance profile is shown in Table 4.6. Although the execution time per frame is drastically reduced (by two orders of magnitude compared to Table 4.4), we can see that still almost 85% of the execution time is spent in the IMDCT and SubBand synthesis functions.

**Table 4.6. MP3 Profile After First Phase of Optimization**

| Function name | Execution time (s) | % |
|---|---|---|
| Inv_mdctL | 0.0144 | 49.54 |
| SubBandSynthesis | 0.0103 | 35.30 |
| III_dequantize_sample | 0.0013 | 4.33 |
| III_stereo | 0.0008 | 2.83 |
| III_reorder | 0.0007 | 2.28 |
| III_antialias | 0.0006 | 2.15 |
| III_hufman_decode | 0.0007 | 2.48 |
| III_hybrid | 0.0003 | 1.10 |
| Total for one frame | 0.0291 | 100.00 |

In the second phase of optimization, the code is mapped to Intel's IPP library using the SymSoft methodology. Here we find two primitives that match the two critical procedures shown in Table 4.6. The resulting performance profile is shown in Table 4.7. Our method automatically uses two of the IPP routines. While the new profile shows that SubBand synthesis still takes roughly 35% of the execution time for each frame, we see that MDCT is no longer a critical portion of the code. Notice that the execution of the IPP SubBand synthesis routine is one order of magnitude faster than the previous version and the total time for decoding one frame is reduced by a factor of 5.

**Table 4.7. MP3 Profile After Second Phase of Optimization**

| Function name | Execution time (s) | % |
|---|---|---|
| ippsSynthPQMF_MP3_32s16s | 0.00176 | 35.242 |
| III_dequantize_sample | 0.00124 | 24.79 |
| III_stereo | 0.00082 | 16.46 |
| III_hufman_decode | 0.00067 | 13.416 |
| IppsMDCTInv_MP3_32s | 0.00047 | 9.4113 |
| III_get_scale_factors | 3.4E-05 | 0.6808 |
| Total time for one frame | 0.00499 | 100.00 |

Table 4.8 summarizes the performance and the energy results of the overall optimization process we described in this section. All measurements are performed on the SmartBadgeIV while running at maximum processing speed and voltage. We start from the original source code obtained from the standards web site that runs roughly two orders of magnitude slower than real-time playback. The next two rows show the results of mapping only into Intel's IPP library; more specifically, we are able to automatically use IPP's SubBand Synthesis and IMDCT in the original code. However, the rest of the code remains intact and still operates on floating-point data. StrongARM SA-1110™ cannot perform floating-point operations natively. As a result, the execution time of the code is still far from real-time playback.

**Table 4.8. Execution Time and Energy of Different Versions of the MP3 Decoder**

| Code version | Execution time (s) | Improvement factor | Energy (mWhr) | Improvement factor |
|---|---|---|---|---|
| Original | 503.92 | 1.0 | 509.6 | 1.0 |
| Original + IPP SubBand | 301.43 | 1.7 | 292.5 | 1.7 |
| Original + IPP SubBand & IMDCT | 211.27 | 2.4 | 199.1 | 2.6 |
| SymSoft first phase (FPh) optimization | 5.47 | 92.1 | 4.47 | 114.2 |
| FPh + IPP SubBand | 3.33 | 151.4 | 2.78 | 182.3 |
| SymSoft final optimization (FPh + IPP SubBand & IMDCT) | 1.43 | 352.4 | 1.17 | 435.2 |
| IPP MP3 (Best possible) | 0.41 | 1240.8 | 0.31 | 1626 |

The fourth row corresponds to the result of the first phase of optimization using SymSoft methodology (without using the Intel library). In this phase, the target libraries used in the mapping step consist of the assembly instructions available on the StrongARM and a set of in-house fixed-point routines. As shown, we have achieved an improvement of two orders of magnitude in both performance and energy for this mapping. The improvement is because of effective use of the MAC instruction available on StrongARM and conversion of most floating-point operations to fixed point. Fixed-point accuracy is verified through simulation.

An additional saving of a factor of four is obtained by further optimizing the code and adding Intel's IPP library to the target libraries in the mapping step. The improvement of factor of four is solely due to automatic use of complex library elements that have been pre-optimized for the given processor. Full compliance to the standard of each version of MP3 code is ensured by checking the accuracy at each mapping step with MP3 compliance test [45]. Note that even larger energy savings are possible by using processor frequency and voltage scaling, since the final MP3 code optimized by SymSoft runs almost four times faster than real-time playback.

The last row in the table, IPP MP3, represents fully hand-optimized MP3 code for StrongARM available from Intel. The final optimized version by SymSoft is a factor of 3.5-3.7 times worse than the IPP MP3. The lower bound on execution time (IPP MP3) is achieved by full manual optimization, which is an error-prone and tedious task. Our methodology reduces the manual intervention of software designers in the optimization process and its results are still faster than real-time playback. Such improvements were previously only possible by skilled designers, familiar with the hardware and software, hand optimizing the code for a given embedded system platform.

As it can be observed from Table 4.8, the reported optimization space for the MP3 decoder spans over three orders of magnitude. The major contribution of this work is to provide a semi-automated optimization flow that closely approaches the lower bound of the optimization space within the limitations of polynomial representation for code sections. Our approach is particularly suitable for data intensive algorithms such as DSP and multi-media applications, since large portions of these software codes can be easily represented by polynomials.

## 4.5. SUMMARY

The contribution of this chapter is a symbolic mapping algorithm and methodology, SymSoft, for energy and performance optimization of algorithmic level software code to execute on a given embedded processor. There are three main steps in our methodology: library characterization, target code identification, and library mapping. The library characterization step finds a polynomial to represent the functionality of each library element and associates a set of parameters such as execution time, energy consumption, and input/output type with each library element. In the target code optimization step, our tool uses execution time and energy profiling to automatically identify need of automated data representation conversion and the critical sections of the code that would benefit most from optimization. For transcendental arithmetic functions, approximation into a polynomial representation is needed in order to enable symbolic algebra techniques. Finally, the library-mapping step uses symbolic computer algebra to automatically decompose the polynomial representations of the critical code sections into a set of library elements available for the embedded processor.

We demonstrated application of our tool, SymSoft, to the optimization of several examples on the SmartBadgeIV [22] embedded system. Using SymSoft for source code optimization, we have been able to increase performance and energy consumption of these examples dramatically while satisfying the output accuracy requirements. These improvements are achieved by the use of pre-optimized software library functions, conversion of critical floating-point operations to fixed point, and reducing the number of memory accesses and instructions executed in critical code segments. The technique presented in this chapter can be easily used in conjunction with other compiler optimization techniques [27].

# CHAPTER 5
# INSTRUCTION SET SELECTION AND USAGE

In the previous two chapters, the focus has been on the design and optimization of hardware and software sections of an embedded system independently. In this chapter, software and hardware co-design is addressed. The co-design methodology presented in this chapter, identifies sections of the software that are critical and are more appropriate for hardware. Mapping these segments to hardware can greatly reduce the execution time of the application. Application-specific instruction-set processors (ASIPs) are suitable for such embedded systems. ASIPs have time-to-market advantage over custom design ASICs and performance and power advantages over traditional fixed instruction set processors. These processors are microprocessors where the instruction set is specialized based on a given application. ASIPs are tailored to include new ad-hoc functional units and instructions that calculate the critical sections of the application software.

Typically, the specialization of embedded ASIPs in a manual task. Our objective is to facilitate the specialization of application-specific processors and to automate the use of the new complex instructions added to the processor. In this chapter, we propose a new specialization methodology based on extracting multiple-input single-output dataflow graphs and symbolic manipulation of polynomials. First, we automatically identify clusters of combinatorial operations that can be grouped into single operations implemented in new functional units of the ASIP. Next, we use symbolic algebraic algorithms to map dataflow sections of our software to the potential new complex instructions. The combination of algorithms from symbolic computer algebra and standard compiler optimization techniques allows novel automatic code transformations that are hard to find by traditional graph covering methods.

**Example 5.1.** As a motivating example, consider the code segment shown below:

```
int foo(int a, int b, int c, int d){
  return a*b+c*d;
}
```

Assume that for a digital video application `foo` is a critical function. Therefore, we add a functional unit to the application-specific processor that calculates `foo`. Next, consider a basic block of the same digital video application calculating:

```
Y1 = a * R1 + b * G1 + c * B1 + d;
Y2 = a * R2 + b * G2 + c * B2 + d;
Y  = Y2 + q * (Y1 - Y2);
```

With proper variable renaming and the algebraic knowledge that `d = 1*d`, we can calculate `Y1` and `Y2` using the new `foo` instruction added to the processor as follows:

```
Y1 = foo(a, R1, b, G1) + foo(c, B1, d, 1);
Y2 = foo(a, R2, b, G2) + foo(c, B2, d, 1);
```

By using the *expand* routine in symbolic algebra, Y can be transformed and mapped to the `foo` instruction:

```
Y = Y2 + q * Y1 - q * Y2;
Y = Y2 + foo(q, Y1, -q, Y2);
```

Thus, the number of instructions used to calculate this basic block is reduced from 15 instructions to 9 instructions that include 5 `foo` instructions, 3 adds, and 1 negate. Now, we formulate one polynomial for the original basic block:

```
Y = a*R2+b*G2+c*B2+d+q*a*R1+q*b*G1+q*c*B1-q*a*R2-q*b*G2-q*c*B2;
```

By applying the *collect* symbolic polynomial manipulation on Y, and substituting `1-q` by `p`, we have:

```
Y = ((1-q)*R2+q*R1)*a+((1-q)*G2+q*G1)*b+((1-q)*B2+q*B1)*c+d;
Y = (p*R2+q*R1)*a+(p*G2+q*G1)*b+(p*B2+q*B1)*c+d;
Y = foo(foo(p,R2,q,R1),a,foo(p,G2,q,G1),b)+foo(foo(p,B2,q,B1),c,d,1);
```

As shown, the new equation can be mapped to 7 instructions: 5 `foo` instructions, 1 add, and 1 subtract. The new mapping is even more efficient than the previous one. This complex solution is hard to find in tree mapping methods since the number of operation to calculate `Y` initially increases. ■

Currently, no tool can perform these kind of algebraic optimizations automatically. Furthermore, utilization of additional ASIP instructions is the responsibility of the designers. Thus, designers manually implement such optimizations using their knowledge of algebra and insert the proper intrinsic function calls.

This chapter presents a methodology that combines detection of potential new instructions for application-specific processors with algebraic manipulations such as the one shown in the previous example. The result of this combination is automatic instruction set selection and mapping. First, a set of potential functional units is extracted from the application software by the multiple-input single-output (MISO) dataflow extraction tool. Next, symbolic algebra is used to map the polynomial representations of the basic blocks to the new instruction set. The new instruction set is determined based on the usage frequency, cost, and possible execution time improvement. Finally, symbolic algorithms are used once more to map the basic blocks to the new instruction set.

The application of this methodology spans from pure ASIP design with extensible functional units to processors equipped with embedded reconfigurable arrays. To minimize microarchitectural and technological assumptions, the analysis of the results focuses on the former type of designs without excluding the viability for the latter implementation. As an example of an extensible ASIP, we are using a Tensilica [50][26] core in our experimental setup.

The chapter is organized as follows: Section 5.1 discusses previous work in software optimization for configurable processors. Section 5.2 presents our proposed methodology, and explains each of its steps. The results of several examples and the improvements achieved by automatically specializing the Tensilica core are presented in Section 5.3. Finally, Section 5.4 summarizes contributions of this work.

## 5.1. RELATED WORK

This chapter combines two related areas of research: automatic identification of instruction-set extensions and use of symbolic algebraic manipulations to map dataflow sections of the code to complex instructions available on the processor. We will discuss the latter area first and then the state-of-the art in instruction identification.

Advanced compilers integrate some tree restructuring capabilities based on algebraic properties to reduce the execution time of complex calculations [27][58]. The goal of such restructuring is typically very precise; for example isolating constants to minimize the amount of address calculation at runtime. In such cases, a number of basic tree transformation rules, applied recursively, result in the desired optimized tree. Similarly, research compilers [51] for SIMD architectures use a fixed set of algebraic tree restructuring rules for associativity and commutativity to improve the quality of SIMD instruction selection. Our approach is more general in that it explores all restructuring possibilities of a dataflow section derived from results of elimination theory and Gröbner bases [7]. A comprehensive set of algebraic manipulations becomes necessary, as defining a straightforward series of transformations is not possible in the general case of complex instruction selection.

The problem of identifying instruction-set extensions consists of detecting clusters of operations that, if implemented as a single complex instruction, maximize a metric—typically performance. Previous works [54][55] have combined template matching (or *instruction mapping*) and template generation (*instruction identification and selection*, in this text) for ASIPs. Kastner et al. [54] cluster operations based on the frequency of node types successions—e.g., multiplications followed by additions. The authors observe that the number of operations per cluster is typically small and conclude that simple pairs of operations appear to be the best candidates. In addition, their work does not account for constraints on the number of inputs and outputs of the clusters. Arnold et al. [55] propose a very similar method from the identification perspective, although the overall goal and architectural context is rather different. Work in reconfigurable computing (e.g., [56][57]) also tackles instruction-set identification. Algorithms are relatively simple and typically result in small suboptimal instructions.

In the experiments presented in this chapter, we have designed specialized Xtensa processors for a set of different applications using the Tensilica toolset [50]. Tensilica provides an extensible core architecture and a methodology and toolset to specialize the core architecture for a given application. The Xtensa architecture can be extended by adding new function units to the microprocessor. These extensions are described in the Tensilica instruction extension (TIE) language. The TIE descriptions are compiled into hardware and integrated in the core Xtensa processor. However, identifying suitable extensions to the core Xtensa processor is a manual task and based on designers creativity. In addition, the TIE instructions added a processor should be used manually in the application software through intrinsic function calls. Both these limitations are automated by the methodology presented in this chapter.

## 5.2. METHODOLOGY

Here we present a methodology that aims to automate specialization of an application-specific processor by adding new functional units to an extensible basic ASIP core. Our methodology also targets automatic use of the new functional units in the application software. Ideally, designers can profile the software code to find the critical sections of the code. These sections are then added to the ASIP core as new functional units and instructions. In addition, a compiler-like tool would optimize the algorithmic-level description of the software to use the new instructions automatically. However, in reality, optimum selection of new instructions is not possible solely by traditional profiling tools. Moreover, designers need to manually modify their original code to use the new functional units or complex instructions. Usually, designers use the profiling information as a guideline for selecting new instructions. Along with that, the software code is manually restructured to find common blocks or functions that could be mapped to new functional units or instructions. Automating selection and use of new instructions can save much design time.

Figure 5.1 shows the overview of our two-step methodology. We start with the high-level C code describing our application and the instructions available on the base core. In the first step, we combine multiple-input single-output (MISO) dataflow extraction with symbolic algebraic mapping to define the new instruction set. Note that the base instruction set of extensible ASIPs can generally execute control segments of the application efficiently. Thus, we focus on dataflow sections of the code and add new combinational functional units to the base ASIP to accelerate critical basic blocks of the code.

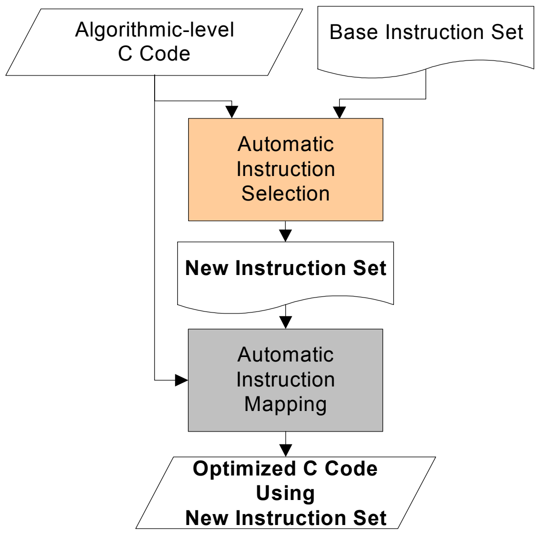

Algorithmic-level C Code
Base Instruction Set
Automatic Instruction Selection
New Instruction Set
Automatic Instruction Mapping
Optimized C Code Using New Instruction Set

**Figure 5.1. SymASIP Methodology**

Once the instruction set is selected, decomposition algorithms empowered by symbolic algebra are used in the second step to automatically map the basic blocks to the new instruction set. Each basic block is modeled by its polynomial representation. These polynomials are decomposed into a sequence of instructions available on the new customized ASIP by polynomial manipulation techniques.

The key contribution is in the use of symbolic algebra combined with dataflow extraction technique to automate instruction set selection and use of the new instruction set for basic block optimization. Note that our methodology is compliant with other software optimization and processor customization techniques. Additional benefits are gained for example by customization of the register file and cache units or by compiler optimizations such as dead code elimination and constant propagation. The next sections describe steps of our methodology in detail.

### 5.2.1. Automatic Instruction Selection

In this section, we will explain the details of the automatic instruction selection step that corresponds to the first shaded box in Figure 5.1. As mentioned earlier, the base instruction set of an extensible ASIP is usually sufficient for control flow. Therefore, our focus is on adding new instructions that effectively execute critical dataflow sections of the code. Figure 5.2 shows the different steps necessary for automatic instruction set selection. We start by extracting *multiple-input single-output* (MISO) dataflow segments from the high-level C code description of our application. A MISO is a set of nodes and edges of a *directed acyclic graph* (DAG) that except for one node all destination nodes of the edges belong to the MISO [52]. Identification of MISOs within a DAG requires an algorithm of only linear complexity. The set of extracted MISOs represent the potential new instructions.

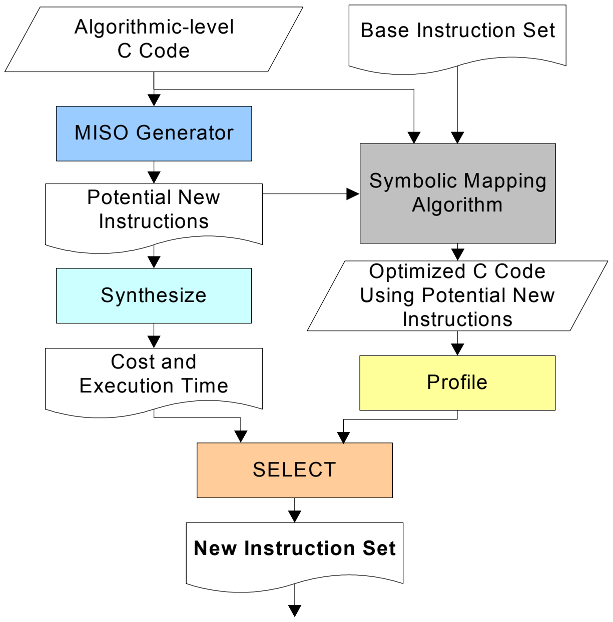


**Figure 5.2. Automatic Instruction Selection**

Using symbolic algebraic algorithms, the original software application is optimized and mapped to the union of the base instruction set and the potential new instruction set.

The potential new instructions are synthesized in order to extract their cost and execution time. The next step is to profile the optimized software code. The output of the profiler indicates the frequency that each potential new instruction can be used in the application. Note that the symbolic decomposition step automatically identifies all possible sections of the code that can be mapped to extracted MISOs without manual intervention or the need to restructure the original program. Using the frequency of each instruction and their associated cost and execution time, a set of most promising MISOs are selected as instructions to be added to the base ASIP. In the next section, we will describe the MISO extraction and symbolic decomposition steps in more detail.

#### 5.2.1.1. MISO Extraction

The multiple-input single-output (MISO) dataflow extractor tool described in this chapter implements two different algorithms. Both algorithms extract single output subgraphs from the basic blocks of the embedded application that correspond to potential new instructions. The analysis starts with the *directed acyclic graph* (DAG) representation of the basic blocks of the given application. Nodes of the DAG are assembler-like instructions and edges represent data dependency among instructions.

The first method is a greedy algorithm of linear complexity that extracts maximal single output subgraphs from basic blocks. The extracted subgraphs are called *MaxMISO*s (maximal multiple-input single-output). The algorithm starts from each exit node of the basic block and constructs a subgraph by trying to recursively include the parent nodes [52]. Subgraph formation never stops for excess of inputs—inputs are unlimited in a MaxMISO—but it stops if inclusion of a further parent node violates the output constraint. Therefore, MaxMISOs are maximal in the sense that adding any further node would fundamentally violate the single output constraint. The algorithm complexity is linear in the number of nodes of the initial graph. Extracted subgraphs produce a single result, while their number of inputs is unlimited. If the resulting number of inputs is unpractical, the MaxMISO is ignored. The MaxMISO algorithm represents a good tradeoff between complexity of exploration and effectiveness of the resulting extracted instructions.

The second algorithm used in this chapter is called *Optimal* [53]. It extracts instructions satisfying user-given input/output constraints that result in maximal speedup. *Optimal* analyses the dataflow graphs of the basic blocks of the application and considers all possible subgraphs. The input and output requirement of the subgraphs is calculated and only those satisfying all constraints are selected for further consideration. The number of subgraphs being exponential in the number of nodes of the graph, *Optimal* has an exponential worst case complexity; yet, it exploits some graph characteristics which allow significant pruning of the search space and, in practice, it exhibits a subexponential complexity. Graphs with up to a couple of hundreds of nodes can be processed in matter of hours. While *Optimal* is designed to satisfy any user-given output constraint, it has been used here only with a single output.

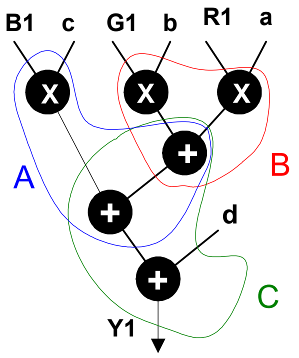


**Figure 5.3. MISO Extraction on Y1 = a*R1 + b*G1 + c*B1 + d**

Speedup estimation is then performed for the potential instructions extracted by either the *MaxMISO* or the *Optimal* algorithm. The estimation consists in comparing the approximate subgraph execution time in software, as a sequence of instructions, with the approximate time the subgraph takes if implemented in hardware, as a single special instruction. The most promising candidates are then passed on to the *Symbolic Mapping* phase.

The behavior of *Optimal* is now shown on the motivational example seen at the beginning of this chapter, for an input/output constraint of 4/1. Part of the DAG of the main basic block is shown in Figure 5.3. *Optimal* identifies subgraphs within constraints, and estimates their gain by using a rough estimation model, described in [53]. The subgraphs A, B and C, are finally selected by *Optimal* to be passed on to the next phase, as the most promising candidates for new instructions. Candidate B corresponds to the `foo` instruction shown in the motivating example of this chapter. Next, we will show how Candidate B is chosen by the symbolic mapping technique as a new instruction.

#### 5.2.1.2. Symbolic Mapping and Optimization

The symbolic mapping algorithm requires two sets of inputs: a set of polynomials representing the basic blocks of the application and another set of polynomials representing the complex dataflow instructions. The goal of the symbolic optimization step is to decompose the polynomial representations of the basic blocks into a minimum number of polynomial representations of available instructions. Such decomposition is done with the help of symbolic computer algebra routines and algorithms. Functional units added to an extensible ASIP execute in one cycle or they are automatically pipelined [50]. Thus, using a minimum number of instructions to calculate a given basic block improves its execution time.

The polynomial representation of a basic block can be directly extracted from the C code if the basic block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [47] can be used to formulate the equivalent polynomial representation. Note that Boolean functions and polynomial functions accelerate greatly when mapped to hardware. Therefore, these basic blocks are the excellent candidates to be mapped in new functional units of the processor. Approximation, such as Taylor or Chebyshev series expansion, can also be used to extract polynomial representation for basic blocks that calculate a transcendental function. In this chapter, we will not use approximation techniques.

This section gives a brief overview of the mapping algorithm. The core of the library-mapping algorithm is the simplification modulo set of polynomials (*simplify*) routine, described in Chapter 2. The polynomial representations of the basic blocks are simplified modulo a set of polynomials. This set, a.k.a. the *side relation set*, represents a subset of the instruction set. Next, we describe the *simplify* routine by means of the motivating example of this chapter. The polynomial representation of the basic block is equal to (`p=q-1`):

```
Y=a*q*R1+a*p*R2+b*q*G1+b*p*G2+c*q*B1+c*p*B2;
```

Side relation sets are selected from the polynomial representations of the MISOs reported by the MISO extraction tool (Section 5.2.1.1) with proper variable renaming. In this example, we show two side relation sets, `siderel` and `siderel2`, with elements whose polynomial representations match Candidate B shown in Figure 5.3. Next, we apply simplify on `Y` modulo the selected side relation sets. The result reported by Maple is shown in bold-italic. A mapping is found when the result of simplify is only one variable or corresponds to a polynomial representation of an instruction in our instruction set. This process is repeated for different combinations of the MISOs and the instructions available on the base core. Candidate B is chosen as a new instruction as it is more frequently used in execution (or mapping) of `Y`.

```
>  siderel:={s1=q*R1+p*R2, s2=q*G1+p*G2, s3=q*B1+p*B2};
>  z := simplify(Y, siderel, [R1, R2, G1, G2, B1, B2]);
   z = a*s1+b*s2+c*s3
>  siderel2:={s4=s1*a+s2*b, s5=s4*1+c*s3};
>  simplify(z, siderel2, [s1, s2, s3]);
   s5
```

As shown, side relation set selection is a non-trivial task. Therefore, to find the best possible mapping, the side relation set should be set equal to all subsets of the instruction set with all possible permutations of the input variables. Algorithm 5.2.1.2 is used to prune the search space efficiently. Let `S` be the polynomial representation of the basic block to be decomposed into complex dataflow instructions. We start by simplifying `S` modulo each instruction as the side relation. The simplification results are stored in a tree data structure. If a simplification result is identical to the polynomial representation of an available instruction, a possible solution is found and the corresponding tree node is marked accordingly. If the simplification result stored in a tree node does not correspond to a library element, we recursively apply the same steps to the new tree node.

**Algorithm 5.2.1.2.** Decompose S into the instruction set L

```
procedure Decompose(S, L)
  # Given a polynomial representation of the basic block S
  # and a set of polynomials L corresponding the instruction set
  # decompose S into elements of library L.
  # initialize tree
  treeroot(S);
  depth ← 0
  bound ← -1
  while depth ≠ bound do {
     bound ← Explore(S, L, depth) # Explore is defined below
     depth ← depth +1
  }
  report best solution in tree
end
# used in Decompose procedure
int function Explore(S, L, d)
  bound ← -1
  for all n ∈ in tree with depth d do{
    for all sr ∈ L do{
      result = simplify(n, sr);
       # make result a child of node n
      addchild(n, result);
       if result ∈ L
         # solution is found
         bound = treedepth(result);  }}
  # returns –1 if no solution is found yet.
  return(bound)
end
```

The bounding function used to reduce the search space is the number of instructions used to calculate the basic block. In other words, if we find a solution that calculates the basic block with two instructions we will not explore solutions requiring more than two instructions. Nevertheless, we will uncover all two-instruction solutions and choose the one with optimal cost or execution time. The number of instructions used is equivalent to the depth of the simplification tree. Therefore, the tree is bounded by the depth of the first solution found. This algorithm was implemented in C with calls to Maple V for symbolic manipulations.

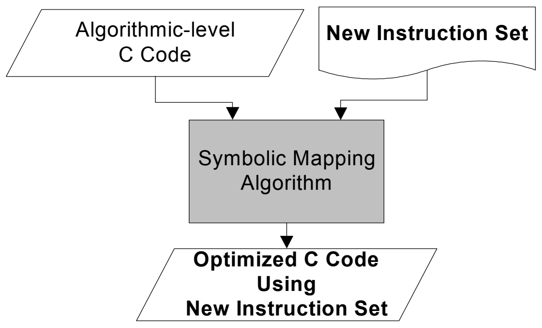

Algorithmic-level C Code
New Instruction Set
Symbolic Mapping Algorithm
Optimized C Code Using New Instruction Set

**Figure 5.4. Automatic Instruction Mapping**

### 5.2.2. Automatic Instruction Mapping

The new instruction set of the ASIP has been chosen by the step described earlier. The original software code is now automatically transformed to use the new instruction set assisted by symbolic polynomial manipulation algorithms. Figure 5.4 gives an overview to the automatic instruction-mapping step. This step also corresponds to the second shaded box of Figure 5.1. The polynomial representations of basic blocks of the software application and the new instruction set of the ASIP are available to the symbolic mapping algorithm. As opposed to tree covering based algorithms, in our algorithm, mapping is performed simultaneously with algebraic manipulations.

The automatic instruction-mapping step uses Algorithm 5.2.1.2 described in Section 5.2.1.2. The output of this step is optimized C code with intrinsic function calls automatically inserted. The optimization criteria consist of using a minimum number of instructions to calculate a basic block of the original code. Since added functional units either are pipelined or execute in one cycle, this mapping greatly reduces the execution time.

## 5.3. RESULTS

We have optimized several Tensilica [50] cores for a set of software examples using our automatic instruction selection and mapping methodology. In the first step, the MISO extraction tool selects a set of possible complex instructions for each software application. The symbolic mapping technique is used to map the code to the new instructions available. At the end of this step, a subset of the MISO set is selected and implemented as new functional units of the ASIP core under design. The selection is based on the cost of each MISO and the frequency of its use.

**Table 5.1. Execution Time Improvements Reported by the ISS**

| **Examples** | **Base core Execution time (cycles)** | **Extended core Execution time (cycles)** | **Improvement (%)** |
|---|---|---|---|
| dot_product | 72990 | 54011 | 26.00 |
| Iir | 88838 | 20652 | 76.75 |
| Fir_2dim | 168978 | 114488 | 32.25 |
| convolution | 182492 | 123035 | 32.58 |
| Fir | 268228 | 158642 | 40.86 |
| DES crypt | 1118570 | 916884 | 18.03 |
| Adpcm | 22514517 | 14176587 | 37.03 |
| MP3 | 2224094335 | 745248522 | 66.49 |
| **Average** | | | 41.25 |

In the next step of the flow, we take the new instruction set and automatically use the complex instructions available to optimize our original software code. We have used the Tensilica [50] software toolchain to measure the execution time improvement of each of our examples as result of the new instructions added to the base core. The base core is the default 32-bit Tensilica core plus a 32-bit multiplier. The functional units selected are added to the base core using the Tensilica instruction extension (TIE) language. Table 5.1 reports the execution time of each example (measured in cycles) as reported by the Tensilica instruction set simulator.

The first five examples in Table 5.1 are simple filters and dot-product examples from a DSP benchmark. The *DES crypt* example is the MD5 message-digest algorithm that produces a 128-bit fingerprint for an arbitrary length message or file. The *adpcm* example is an adaptive differential pulse code demodulator software used for speech compression/decompression. Finally, the last example is a fixed-point MP3 decoder software decoding a 5 second long stream. By applying our methodology to the MP3 decoder and adding only three new instructions to the base core, the decoder executes three times faster.

**Table 5.2. Area Cost of the Added Instructions**

| **Examples** | **Number of Instructions Added** | **Area of Instructions Added (mm$^2$)** | **Area Increase of the Base Core (%)** |
|---|---|---|---|
| dot_product | 1 | 0.211 | 7.3 |
| Iir | 1 | 0.503 | 17.3 |
| fir_2dim | 1 | 0.211 | 7.3 |
| convolution | 1 | 0.211 | 7.3 |
| Fir | 1 | 0.199 | 6.8 |
| DES crypt | 3 | 0.142 | 4.9 |
| Adpcm | 1 | 0.103 | 3.5 |
| MP3 | 3 | 0.564 | 19.5 |
| **Average** | | 0.268 | 9.2 |

To estimate the cost associated with the execution improvements reported in Table 5.1, we have synthesized the new functional units added for each example using Synopsys Design Compiler and a 0.35-micron CMOS technology library. The area of the base core is approximately 0.29 mm$^2$ in this technology. Table 5.2 shows the number of new instructions added to the base core, the area of the new instructions, and the area increase of the base core. As it is observed from Table 5.2, our methodology selects a small number of instructions to be added to the base processor that result in modest area increase. Nevertheless, due to its strong instruction selection and mapping engine, the instructions added are key instructions that can be used in many sections of the code, and thus significantly decrease the execution time. Note that the area reported in Table 5.2 is an upper bound, as the new instructions are synthesized separate from the base core and resource sharing is not considered. For all examples shown in this section, we have added a total of ten different instructions to different cores. The complexity of the added instructions ranges from two operations to twenty operations.

## 5.4. SUMMARY

The contribution of this chapter is a new methodology that automates the selection of very complex instruction set extensions for ASIPs together with aggressive techniques to map the basic blocks to such complex instructions. This work focuses on arithmetic intensive applications such as multi-media processing. A basic ASIP core is extended automatically to include ad-hoc functional units that accelerate the dataflow sections of the software application. A set of potential instructions is generated by the multiple-output single-input (MISO) dataflow extraction tool. Symbolic computer algebra is used to discover transformations that expose unintuitive opportunities for mapping basic blocks of an application into the potential instructions. The most frequently used MISOs by the symbolic mapping tool are selected and added to the base ASIP processor. Symbolic algebra automates very smart instruction mapping previously only possible by designer's manual intervention.

We demonstrate the application of our tool to a set of arithmetic intensive examples including an MP3 decoder software. A Tensilica core was optimized for each application using the Tensilica tool set. We have achieved an average of 41% improvement in the execution time of our examples, while paying only an average of 9.2% penalty in area cost.

Another possible application of our technique is to facilitate reuse of an ASIP in future generations of an application. While hard-wired ad-hoc functional units present the risk of inflexibilities towards subsequent changes of an application, our smart symbolic mapping techniques increase the possibility of using instructions tailored for a previous generation of the application. In future work, we also plan to find dataflow instructions with more than one output.  Such sections can be selected by the Optimal [53] algorithm and represented by a set of polynomials for the symbolic mapping step.

# CHAPTER 6
# CONCLUSION

Embedded systems are now in every corner of our world and their presence is constantly increasing. Due to their high complexity and short turn around time, embedded-system design automation is now a necessity. This thesis presents a set of algorithms and methodologies for design and optimization of different components of an embedded system. The tools, methodologies, and algorithms presented in this thesis increase designer's productivity and reduce to design cycle of an embedded system. In addition, they provide a better quality of result due to a wide design space exploration at a high level of abstraction. This thesis starts with the algorithmic-level description of designs from the multimedia and digital signal processing (DSP) domain of applications. Multimedia and DSP algorithms are mostly arithmetic intensive descriptions that result into designs with considerable data-path components.

This thesis leverages from results of research and development in the field of symbolic computer algebra. By using routines from symbolic computer algebra, the described design algorithms are capable of algebraic manipulation and arithmetic optimization. To our knowledge, using symbolic algebra in optimization and synthesis of systems was not previously explored by other design tools.

## 6.1. SUMMARY OF CONTRIBUTIONS

In this thesis, symbolic polynomial manipulation techniques are used to develop algorithms, tools, and methodologies that cover all aspects of embedded systems design including hardware, software, and processor design.

To design a data-intensive hardware block, a set of algorithms, tools, and methodologies are presented that automatically map the basic blocks of the algorithmic-level description of a design to pre-optimized arithmetic library elements. The mapping and component selection is performed simultaneous with arithmetic manipulations on the given basic block. These manipulations are possible by using algorithms from symbolic computer algebra. Since different variation of a dataflow may result in different library component selection, a wider design space is explored. The result is a data path that implements the given dataflow optimally using the available library. Our method eliminates the need for synthesis directives from hardware designers.

Software changes are frequent in embedded systems. Multimedia and DSP applications have very complex software components. In this thesis, a methodology, tool, and algorithm are presented to optimize the execution time and energy consumption of an embedded software program. Energy profiling and symbolic mapping algorithms are used to select and optimize critical section of an embedded software program respectively. The symbolic mapping algorithms map the critical section of the software to complex microprocessor instructions or embedded software library functions. The results associated with the software optimization methodology show dramatic improvement on the execution time and energy consumption of a set of programs running on a prototype embedded system hardware.

Software/hardware co-design is more important for embedded system design as the software and hardware blocks are more tightly coupled. This thesis presents a co-design methodology based on application specific processors. A set of functional blocks is added to the base processor to accelerate the critical sections of the given application.

This defines a new instruction set for the application specific processor. The original application automatically optimized and mapped to the instruction available on the processor using an algorithms based on symbolic computer algebra. New hardware is added to the application specific processor to execute the new instructions defines. The software executing on this platform is automatically optimized and co-designed. The method was tested on different applications and a set of specialized processors was automatically generated. Results show significant execution time improvement achieved with smart and negligible extra hardware added to the base processor.

## 6.2. FUTURE DIRECTIONS

Symbolic computer algebra is a powerful set of algorithms not previously used in the field of system design and optimization. These algorithms open a new set of opportunities in for future research.

One of these possibilities is automatic algorithm optimization. Currently most algorithms are designed manually by skilled engineers. Ideally, an algorithm specified by a designer can be converted by a tool it to an optimum implementation based on a set of constraints. For example, a Fourier transform may be automatically changed to a fast Fourier transform algorithm. Most of the skills necessary for this transformation are implemented in mathematical tools such as Matlab and Maple. Using these algorithms and a guided search over the solution space can effectively synthesizes a new and improved algorithm.

On another note, many embedded system applications can tolerate a given degradation in their output result. For example, an audio decoder satisfies compliancy test when the root mean square of the difference signal between the output of the decoder and the supplied reference less than a given number. In other words, in multi-media applications a notion of arithmetic “don’t care” exists. In this thesis, such “don’t care” conditions were used to reduce the cost of the system. However to automate such task, one should leverage results from approximation theory.

The methodology and algorithms presented in this thesis to automate instruction set selection and usage can be extended to configurable computing. The cost of silicon is decreasing and hybrid FPGA components are now available on the market. These components have a microprocessor and configurable fabric on the same chip. An embedded application can use a similar methodology as the one proposed in this thesis to efficiently use the processor and FPGA. The computational intensive sections of the application can be automatically mapped to the FPGA. These blocks can then accelerate the application code automatically using a symbolic decomposition algorithm.

# BIBLIOGRAPHY

[1] "International Technology Roadmap for Semiconductors", *http://public.itrs.net*, 2001.

[2] *Maple*, Computer Software, Waterloo Maple Inc., *http://www.maplesoft.com/*, 1988.

[3] *Mathematica*, Computer Software, Wolfram Research Inc., *http://www.wri.com/*, 1987.

[4] B. Buchberger, "Some Properties of Gröbner Bases for Polynomial Ideals", *ACM SIG-SAM Bulletin*, 10/4, 1976, 19-24.

[5] K. Geddes, S. Czapor, and G. Labahn, *Algorithms for Computer Algebra*. Boston: Kluwer Academic Publishers, 1992.

[6] T. Becker and V. Weispfenning, *Gröbner Bases*. New York: Springer-Verlag, 1993.

[7] D. Cox, J. Little, and D. O'shea, *Ideals, Varieties, and algorithms*. New York: Springer-Verlag, 1997.

[8] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.

[9] *DesignWare Library*, Synopsys Inc., *http://www.synopsys.com/*, 1994.

[10] J. Smith and G. De Micheli, "Polynomial Methods for Allocating Complex Components", in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 217-222, March 1999.

[11] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall, *Computer Approximations*. New York: John Wiley & Sons, 1968.

[12] D. J. Kuck, *The Structure of Computers and Computations Vol. I*. New York: John Wiley and Sons, 1978.

[13] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup", *IEEE Transactions on Computers*, Vol. C-21, pp. 1293-1310, December 1972.

[14] A. Nicolau and R. Potasman, "Incremental Tree Height Reduction for High Level Synthesis", in *Proceedings of the 28th Design Automation Conference*, pp. 770-774, June 1991.

[15] D. Kolson, A. Nicolau, and N. Dutt, "Integrating Program Transformations in the Memory-Based Synthesis of Image and Video Algorithms", in *Proceedings of the International Conference on Computer Aided Design*, pp. 27-30, November 1994.

[16] H. Wang, A. Nicolau, and K. Siu, "The Strict Time Lower Bound and Optimal Schedules for Parallel Prefix with Resource Constraints", *IEEE Transactions on Computers*, Vol. 45, No. 11, pp. 1257-1271, November 1996.

[17] R. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions", in *Proceedings of the IEEE International Symposium of Circuits and Systems,* pp. 49-54, May 1982.

[18] R. Brayton, G. Hachtel, C. McMullen, and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston: Kluwer Academic Publishers, 1984.

[19] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS: A Multiple-level Logic Optimization and the Rectangular Covering Problem", in *Proceedings of the International Conference on Computer Aided Design*, 1987.

[20] D. Menard, D. Chillet, F. Charot, and O. Sentieys, "Automatic Floating-Point to Fixed-Point Conversion for DSP Code Generation", in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 270-276, October 2002.

[21] P. G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends", *Proceedings of the IEEE*, vol. 85, no. 3, pp. 419-435, March 1997.

[22] G. Q. Maguire, M. Smith, and H. W. Peter Beadle, "SmartBadges: A Wearable Computer and Communication System", in *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, Invited talk, March 1998.

[23] *Coded representation of audio, picture, multimedia and hypermedia information*, ISO/IEC JTC/SC 29/WG 11, Part 3, International Organization for Standardization, May 1993.

[24] M. Willems, H. Keding, T. Grötket, and H. Meyr, "Fridge: An interactive Fixed-Point Code Generation Environment for HW/SW CoDesign", in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 687-690, April 1997.

[25] G. Constantinides, P. Cheung, and W. Luk, "The Multiple Wordlength Paradigm", in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, March 2001.

[26] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/Software Instruction Set Configurability for System-on-Chip Processors", in *Proceedings of the 38th Design Automation Conference*, pp. 184-190, June 2001.

[27] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann Publishers, 1997.

[28] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", *IEEE Computer*, vol. 29, no. 12, pp. 84-89, December 1996.

[29] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*. Boston: Kluwer Academic Publishers, 1995.

[30] R. Leupers, Retargetable *Code Generation for Digital Signal Processors*. Boston: Kluwer Academic Publishers, 1997.

[31] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vanduoppelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*, Boston: Kluwer Academic Publishers, 1998.

[32] V. Tiwari, S. Malik, A. Wolfe, and M. Lee, "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing Systems*, vol. 13, no. 2, pp. 223-238, August 1996.

[33] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization", *IEEE Transactions on VLSI Systems*, vol. 2, no.4, pp.437-445, December 1994.

[34] *Integrated Performance Primitives for the Intel StrongARM SA-1110 Microprocessor*, Intel Corporation, *http://www.intel.com*, 2000.

[35] *TI'54x DSP Library*, Texas Instruments Inc, http://www.ti.com, 2000.

[36] *eCos$^{TM}$ Reference Manual*, Cygnus Solutions, 1999.

[37] *Linux-arm math library reference manual*, RedHat Inc., 2000.

[38] J. Crenshaw, *Math Toolkit for Real-Time Programming*. Kansas: CMP Books, 2000.

[39] H. Mehta, R. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software", in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 72-75, August 1997.

[40] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", in *Proceedings of the 35th Design Automation Conference*, pp.188-193, June 1998.

[41] H. Tomyiama, H., T. Ishihara, A. Inoue, and H. Yasuura, “Instruction Scheduling for Power Reduction in Processor-Based System Design”, in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 23-26, February 1998.

[42] M. Kandemir, N. Vijaykrishnan, M. J. Irwin and W. Ye, “Influence of Compiler Optimizations on System Power”, *IEEE Transactions on VLSI Systems*, vol. 9, no. 6, pp. 801-804, December 2001.

[43] *ARM Software Development Toolkit*, Version 2.11, Advanced RISC Machines (ARM) Ltd., 1996.

[44] T. Simunic, L. Benini, and G. De Micheli, “Energy-Efficient Design of Battery-Powered Embedded Systems”, *Special Issue of IEEE Transactions on VLSI Systems*, pp. 18-28, May 2001.

[45] *Information Technology, Generic Coding of Moving Pictures and Associated Audio: Conformance*, ISO/IEC JTC 1/SC 29/WG 11 13818-4, International Organization for Standardization, 1996.

[46] J. Smith and G. De Micheli, “Polynomial Methods for Component Matching and Verification”, in *Proceedings of the International Conference on Computer Aided Design*, pp.678-685, November 1998.

[47] J. Smith and G. De Micheli, “Polynomial Circuit Models for Component Matching in High-Level Synthesis”, *IEEE Transactions on VLSI Systems*, vol. 9, no. 6, pp. 783-800, December 2001.

[48] V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr, “DSPstone: A DSP-Oriented Benchmarking Methodology”, in *Proceedings of the International Conference on Signal Processing Applications and Technology*, October 1994.

[49] T. Simunic, L. Benini, G. De Micheli, and M. Hans, “Source Code Optimization and Profiling of Energy Consumption in Embedded Systems”, in *Proceedings of the International Symposium on Systems Synthesis*, pp. 193–198, September 2000.

[50] *The Xtensa Processor Generator*, Tensilica Inc., *http://www.tensilica.com*, 1997.

[51] R. Leupers, *Code Optimization Techniques for Embedded Processors*. Boston: Kluwer Academic Publishers, 2000.

[52] C. Alippi, W. Fornaciari, L. Pozzi, and M. G. Sami, “A DAG Based Design Approach for Reconfigurable VLIW Processors”, in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 778-780, March 1999.

[53] K. Atasu, L. Pozzi, and P. Ienne, “Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints”, in *Proceedings of 40th Design Automation Conference*, June 2003.

[54] R. Kastner, A. Kaplan, S. Memik, and E. Bozorgzadeh, “Instruction Generation for Hybrid Reconfigurable Systems”, *ACM Transactions on Design Automation of Embedded Systems*, vol. 7, no. 4, pp. 605-627, October 2002.

[55] M. Arnold and H. Corporaal, “Designing Domain Specific Processors”, in *Proceedings of the 9th International Workshop on Hardware/Software CoDesign*, pp. 61-66, April 2001.

[56] B. Kastrup, A. Bink, and J. Hoogerbrugge, “ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator”, in *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 695-706, April 1999.

[57] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “CHIMAERA: A High-Performance Architecture with a Tightly Coupled Reconfigurable Functional Unit”, in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 225-235, June 2000.

[58] J. Zory and F. Coelho, “Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Code”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* pp. 376-384, October 1998.

[59] A. C. Parker, M. Mlinar, and J. Pizarro, “MAHA: A Program for Data Path Synthesis”, in *Proceedings of the 23rd Design Automation Conference*, pp. 252-258, June 1985.

[60] T. J. Kowalski and D. E. Thomas, “The VLSI Design Automation Assistant: Prototype System”, in *Proceedings of the 20th Design Automation Conference*, pp. 479-483, June 1983.

[61] Z. Yu, K. Y. Khoo, and A. N. Willson, “The Use of Carry-Save Representation in Joint Module Selection and Retiming”, *Proceedings of the 37th Design Automation Conference*, pp. 768-773, June 2000.