

Design Methodology for Low Power Heterogeneous Reconfigurable Digital Signal Processors

Marlene Wan

Abstract

Wireless communication systems represent one of the largest consumer electronic markets, and the digital components in these systems provide the most challenge for digital designers. The wireless digital designs need to provide not only high performance, but also low power usage and small form-factor. Recent wireless applications add a new metric, flexibility, into the implementation requirement. This digital design paradigm calls for implementations that are heterogeneous (which includes ASIC, programmable processor, and reconfigurable components) to meet the requirement of providing performance, energy efficiency, and flexibility. A design methodology that takes all the design metrics and architecture styles into account is needed. This thesis proposes an energy-conscious methodology that can be used to synthesize or compile to such heterogeneous digital-signal processors. The goal of this research is to develop a design environment that takes the digital-signal processing algorithms specified in a high-level language and provide guidance to performs hardware-software-reconfigware partitioning and synthesis under real-time performance constraints and optimization objectives (such as low-energy consumption). This methodology is based on collecting quantitative power-delay-area information about the algorithm and architecture, then combining the information to guide architecture mappings and partitioning. A step-by-step procedure is presented (computational bottleneck extraction, mappings from algorithm to architecture, hardware-software-reconfigware partitioning and optimizations specific to reconfigurable architectures). A set of tools to assist in performing tasks in each step is

developed. At the end of the thesis, a design example that results in a low-power heterogeneous source-CODEC processor is presented.

Jan M. Rabaey, Chair of the Dissertation Committee

1 : Introduction	1
1.1 Motivation	1
1.1.1 Target Application Domain and Implementation Trend.....	1
1.1.2 Lack of Design Methodologies to Bridge the Algorithm-Architecture Gap	5
1.2 Contributions of this Thesis	6
1.3 Organization of the Thesis.....	8
2 : A Template for Heterogeneous Reconfigurable DSPs	10
2.1 Evaluation of Heterogeneous Reconfigurable Architectures	10
2.2 The Pleiades Heterogeneous Reconfigurable Architecture Template	13
3 : Design Methodology for Heterogeneous Reconfigurable DSPs	17
3.1 Survey of Hardware-Software-Reconfigurable Co-Design Methodologies	17
3.2 Overall Flow of the Design Methodology for Low-Energy Heterogeneous Reconfigurable DSPs	20
3.2.1 Assumptions.....	20
3.2.2 Overview.....	22
3.2.3 Description of Each Stage	24
4 : Algorithm Characterization and Architecture Modeling	28
4.1 Fusing Architecture and Algorithm Characterization for Estimation and Exploration	28
4.2 Algorithm Characterizations	31
4.2.1 Application 1: Algorithm Selection.....	32
4.2.2 Application 2: Kernel Detection	33
4.2.3 Application 3: Kernel Frequency Record for Design Evaluation at the System Level	34
4.2.4 Development Environment Details.....	34
4.3 Architecture Characterizations	35
4.3.1 Software on Embedded Programmable Processors	36
4.3.2 Satellite Modules (ASIC components).....	39
4.3.3 Satellite Modules (Embedded FPGA)	40
4.3.4 Interfaces	41

4.4 Starting Point of the Methodology for Heterogeneous Reconfigurable Architecture: A Software-Centric Approach.....	42
4.4.1 Results	43

5 : High-Level Simulation and Synthesis of Reconfigurable Satellite Co-processors..... 46

5.1 Data-driven Reconfigurable Satellite Co-processor Architecture	47
5.1.1 The Data-driven Protocol.....	49
5.1.2 Inter-Satellite Dedicated Connections	50
5.1.3 Libraries of Basic Satellite Modules	52
5.2 High-level Simulation Strategies.....	52
5.2.1 The C++ Intermediate Form (C++IF).....	53
5.2.2 Simulation Based on the C++ Intermediate Form	54
5.2.3 High-level Simulation Case Study.....	59
5.3 High-Level Synthesis Strategies.....	62
5.3.1 Basic Synthesis Flow	62
5.3.1.1 From SUIF to Control Dataflow Graph	63
5.3.1.2 From Control Dataflow Graph to Dataflow-driven Architecture ...	67
5.3.1.3 Useful Transformations	70
5.4 Case Study for the Synthesis Flow.....	71
5.4.1 Evaluation of Data Parallelism	71
5.4.2 Architecture and Algorithm Selection	72

6 Hardware/Software Partitioning and Architecture Instantiation..... 77

6.1 The State-of-the-Art Hardware-Software Partitioning Approaches.....	77
6.1.1 Review of the Computation Control-Flow	78
6.1.2 Formulation of the General Partitioning Problem	78
6.1.3 Solution to the Partitioning Problem.....	80
6.1.4 Examples for the Partitioning Solver.....	82
6.2 Architecture Instantiation	83
6.2.1 Data Allocation for Memory Satellite Instantiation	85
6.2.1.1 Data Allocation Case Study	87

7 Reconfigurable Architecture Specific Explorations and Optimizations..... 90

7.1 Two Design Challenges of Reconfigurable Architectures	90
7.2 The Reconfigurable Interconnect Architecture Exploration Environment	92
7.2.1 Overview of Reconfigurable Interconnect Architectures	94
7.2.2 The Evaluation Environment for Reconfigurable Interconnect Architectures	98
7.3 Reconfiguration and Interface Code Generation and Optimization	104
7.3.1 Basic Code Generation Techniques.....	106

7.3.2 Bottleneck Analysis for Configuration Code Optimization	110
7.3.3 Local Optimizations	111
7.3.4 Global Optimizations	112
7.3.5 Test Cases, Results and Discussion.....	114
8 Architecture Design Examples Using the Methodology	116
8.1 Design of a Reconfigurable CELP-based Digital Signal Processor	116
9 Conclusion and Future Research.....	126
9.1 Summary of Contributions	126
9.2 Future Research Directions.....	127
9.2.1 Heterogeneous Reconfigurable Architecture	128
9.2.2 Design Methodology for Heterogeneous Reconfigurable Architectures	128
9.2.3 Conclusions.....	129
10 Bibliography.....	130

1: Introduction

1.1 Motivation

1.1.1 Target Application Domain and Implementation Trend

Wireless communication systems represent a sector that is experiencing prominent growth in the electronic consumer communication market. To produce successful and competitive new wireless products, the design and implementation of wireless systems require careful consideration and constant re-evaluation. The key components of many wireless products are the digital embedded systems that perform communication and multimedia digital signal processing (DSP). The competitiveness of the product relies heavily on these embedded systems to follow stringent requirements such as low-power consumption, real-time performance and small form-factor (the Power-Delay-Area metric or P-D-A). The current trends in requiring wireless applications to support multiple standards show that another design aspect has become increasingly important: flexibility. An example exists in the infrastructure equipment and handheld terminals for Voice-over-IP (VOIP) applications. VOIP requires that processing elements support many voice channels simultaneously, with each channel supporting a variety of telephony algorithms such as VCELP, G.729 and G.726 [Ell98]. Another example is the much anticipated software radio for future generation of cellular telephone systems. The wireless cellular system is experiencing rapid evolution from the analog 1G standards and the 2G systems we have right now to the 3G standards that promise higher data rate on cell phones. Having multiple standards supported in the front-end [HHF99] and base-band digital signal processing can make the transition to the latest generation of cellular systems much smoother. Last but not least, advanced digital signal processing algorithms that can dynamically adapt to the wireless channel

environment to increase spectral efficiency are highly desirable in future wireless handheld systems. These new sets of applications suggest that the underlying embedded processing platform needs to be flexible as well as conform to the traditional P-D-A requirements.

Figure 1-1 shows different architecture platforms that can be used to implement embedded digital signal processing applications, such as VOIP, wireless cellular systems, and advanced communication algorithms. The different architectures are shown along with their flexibility and energy efficiency indices.

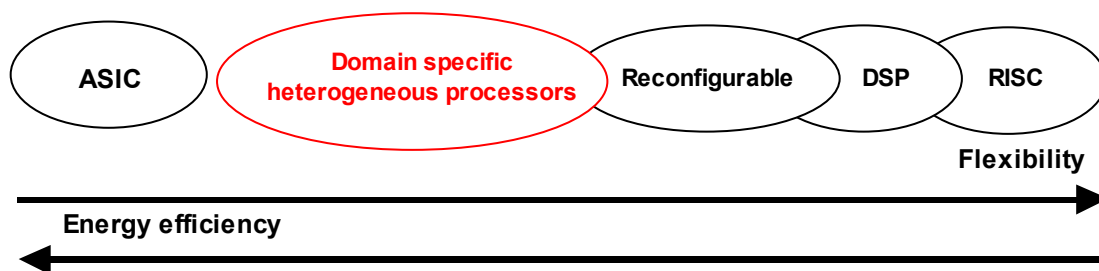


Figure 1-1 The Embedded DSP Implementation Spectrum

At one end of the spectrum, traditional programmable processors (DSP or RISC) provide the most flexibility, but the implementations are often too speed-inefficient or power-hungry. The inability to provide enough energy efficiency is shown by the information presented in Figure 1.2 and Figure 1.3 [GSM00]. Data in Figure 1.3 shows the total MIPS required for a set of existing and upcoming wireless cellular communication standards (GSM, GPRS, and 3G). Two digital signal processing domains, communication and wireless application (voice, image and video coding), are chosen to represent the key majority of the total MIPS requirement. For the most advanced standard shown in Figure 1.2 (3G), the total MIPS is about 5,000 and approaches 10,000, which exceed the complexity of many existing applications. From the implementation side, Figure 1.3 shows the trend in power dissipation for

programmable DSPs. Figure 1.3 shows that the lowest-power DSP processor provides about 0.8 mW/MMAC. Combining the information from the two figures using simple arithmetic (assuming 1 MIPS = 1 MMAC) suggests that the implementation of these communication algorithms can take around 40-80 Watts! The inefficiency is due to the sequential von Neumann or Harvard computation style of the programmable processors which support only limited parallelism required by the applications. At the other end of the spectrum in Figure 1.1, custom ASIC implementations can exploit application specific parallelism and control flow. Therefore, ASICs can offer order of magnitude improvement in terms of energy consumption [as shown in Table1-1] when compared to programmable processors. Table1-1 shows the power consumption of implementing a multi-user detection algorithm for CDMA systems on two programmable processors (StrongARM [ref] and Texas Instrument DSP [ref]) and on an ASIC. A two to three order of magnitude of power saving is shown in the ASIC implementation. However, they cannot provide the flexibility needed. To implement another variant of the multi-user detection algorithm, another ASIC has to be designed while the StrongARM and TI DSP can simply be reprogrammed.

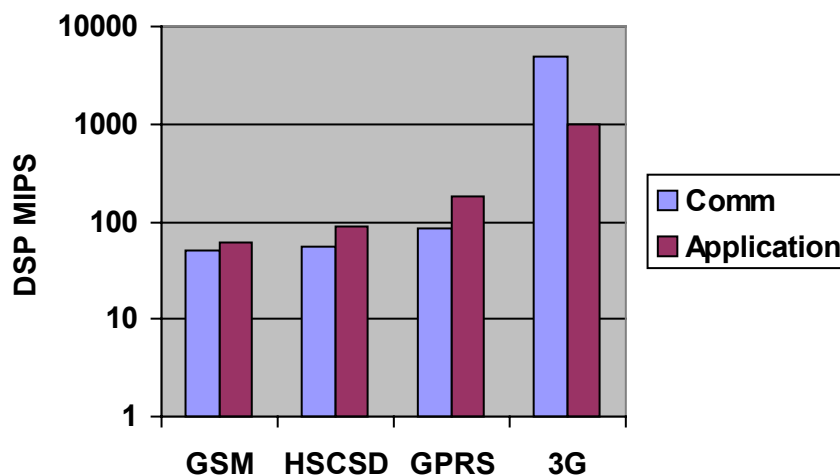


Figure 1-2 Computational Complexity for Various Wireless Standards

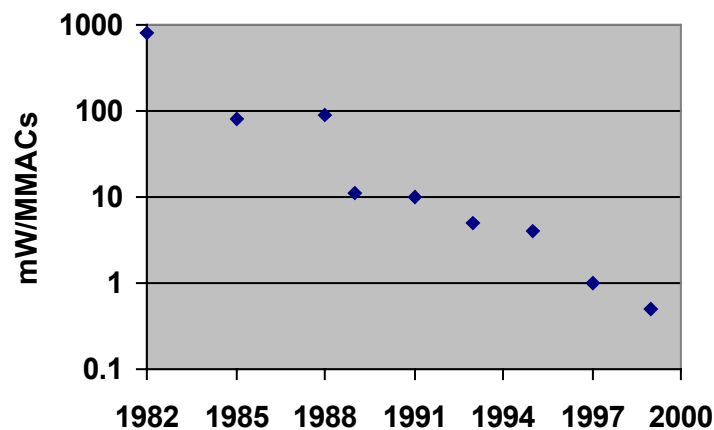


Figure 1-3 Power Dissipation Trends for DSPs

Recently, a new class of architecture, the reconfigurable architecture (or reconfigware), has emerged to be suitable for embedded digital signal processing by demonstrating performance superiority [Xilinx][PADDI] as well as energy efficiency [Nar99]. Reconfigurable architectures are characterized by the composition of active circuitry that can be configured to any of different operations during post-fabrication, which allows flexibility traditionally provided only by programmable processors. Another important feature of reconfigurable architecture is the ability to have application-dependent structure (data-flow and control-flow) that can achieve performance close to ASIC implementations. The combination of post-fabrication reconfigurability and application-dependent optimization is enabling the reconfigurable architectures to become a major implementation platform for embedded processing. However, as shown in Figure 1-1, the reconfigurable architecture is placed closer to the programmable processors since the energy efficiency of such architectures is still relatively low.

There gap between the processors on the two ends of the energy-flexibility spectrum in Figure 1-1 is still quite large. Given a set of embedded digital signal processing algorithms, it is advantageous to mix and match programmable processors,

ASIC, and reconfigurable processors depending upon the requirements of different parts of the algorithms. A good example of such a system on a board is the UCLA video compression system [SVM95], which consists of ASIC and FPGA elements to achieve performance and flexibility. The mixture of these implementation styles reveals a new kind of architecture: *heterogeneous reconfigurable architecture*. With the advance in silicon device technology, it is foreseen that such a heterogeneous reconfigurable computing device on a single chip has the potential to further reduce area and energy for embedded DSP applications.

Table 1-1 Power of Various Architectures Running Multi-user Detection for a DS-CDMA System

	Power
StrongARM	4.2 W
TMS320C54	603.0 mW
ASIC	3.1 mW

1.1.2 Lack of Design Methodologies to Bridge the Algorithm-Architecture Gap

While the concept of utilizing heterogeneous reconfigurable computing platforms on the same chip is attractive, it is quite difficult for a designer to partition and map the algorithms to realize the full potential of such an advanced system architecture, while keeping track of all of the P-D-A metrics. For a long time, the electronic product design process had relied on hardware and software design teams with very minimal interaction. It is not until recently does the importance of hardware-software co-design start to be recognized. As the concept of heterogeneous reconfigurable architectures is even newer than hardware-software co-design, most state-of-the-art design environments do not have the capability to cope with architectures ranging from programmable processors, reconfigurable processors to custom blocks. Therefore, it is critical to have such start building an analysis and compilation environment that can

account for all Power-Delay-Area metrics, and can encourage the co-design of hardware, software, and reconfigware. This thesis describes a design environment to perform such a task.

1.2 Contributions of this Thesis

The main contribution of this thesis is to provide a step-by-step path linking an algorithm specified in a high-level language to an implementation consisting of heterogeneous reconfigurable elements. To achieve an effective algorithm-architecture mapping that can meet all P-D-A constraints, several tools and environments have been developed to quantify effects of mapping algorithm to a specific architecture style.

The starting point of the methodology is a set of algorithms specified in a high-level or system-level language [Pet94] that targets heterogeneous architectures. There are two major approaches to close the algorithm-to-architecture gap (as shown in Figure 1-5). The first approach is the architecture synthesis path that derives a complete implementation from the algorithm. The other approach is the compilation path that assumes an underlying implementation and the goal is to “program” or “reconfigure” the existing architecture. The methodology described in this thesis builds the core (enclosed by the white box in Figure 1-5) that can be used for these divergent approaches.

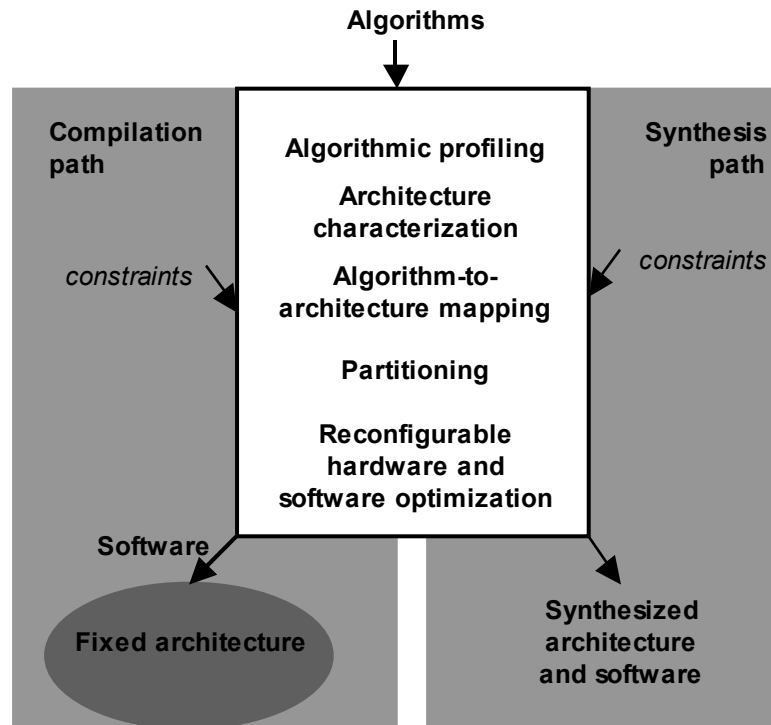


Figure 1-5 Compilation Path and Synthesis Path to Close the Algorithm-Architecture Gap and the Scope of Our Methodology

First, an estimator tool that combines both architecture and algorithm characteristics has been developed to perform first order estimation of the complexity of an algorithm to be mapped to an architecture style (algorithm-architecture co-design). The tool also helps to discover algorithm and architecture matching (hardware-software-reconfigurable co-design) by indicating parts of the algorithm that require more optimization.

Second, a synthesis tool that quickly maps the algorithm to the most promising reconfigurable architecture style is provided to enable rapid evaluation of the mappings. An executable intermediate form is used to represent the output of the synthesis tool as well as the input to the backend design flow. The intermediate form, based on the C++

language, also allows designers to input their own mappings to the reconfigurable architecture.

Finally, aspects that are critical to the reconfigurable architecture design are also addressed. Specifically, issues such as reconfigurable on chip interconnect evaluation and reconfiguration code generation and optimizations are addressed.

1.3 Organization of the Thesis

The rest of the thesis is organized in such a way: In Chapter 2, the concept of granularity and flexibility for computer architectures is defined so that the heterogeneous architectures can be classified in terms of granularity. A survey of reconfigurable architectures is given in this context. The importance of matching architecture and algorithm is described and the benefit of utilizing heterogeneous reconfigurable architecture is introduced. At the end of the chapter, a template for a coarse-grained heterogeneous reconfigurable architecture is given and its superiority in energy efficiency compared with other architectures is shown quantitatively.

Chapter 3 presents an overview of existing design tools for heterogeneous reconfigurable architectures. A design flow suitable for algorithm-architecture and software-hardware-reconfigurable co-design for reconfigurable architectures is presented at the end of chapter 3. The key steps of this methodology are discussed, and the remainder of the thesis gives details to each one of the steps.

In Chapter 4, algorithm characterization methods are introduced as a first step in algorithm partitioning across different architecture styles. Different characterization schemes of various architectures are introduced as a basis for architecture mapping estimation. The concept of quantitatively matching algorithm and architecture characteristics is introduced.

Chapter 5 covers a simulation and synthesis environment targeting a reconfigurable architecture suitable for accelerator functions. Since the target algorithm domain is in digital signal processing and wireless communication, the architecture is data-flow driven. Quantitative evaluations of the architecture granularity and flexibility are presented first. The intermediate forms and the synthesis tools used are described in detail.

Chapter 6 is devoted to system partitioning, satellite type and number instantiation, and memory allocation. The optimizations associated with the target architecture style are discussed.

Chapter 7 presents two important aspects of reconfigurable system design that generally do not exist in traditional IC design methodologies. First of all, an evaluation environment for heterogeneous reconfigurable network is introduced. Secondly, a configuration code generation and optimization process is described.

Chapter 8 presents the design and results of two heterogeneous reconfigurable systems using the methodology proposed in this thesis work. Examples drawn from speech coding and base-band digital signal processing applications are presented.

Chapter 9 summarizes the contributions of this research project, and gives potential future extensions of this work.

2: A Template for Heterogeneous Reconfigurable DSPs

This chapter describes the method used to evaluate different reconfigurable and programmable computational engines and the architecture's corresponding efficiency. Using the evaluation method, we present the benefits of composing architectures of different granularity and flexibility for application domains of our interest. A most promising architecture style is presented at the end of the chapter and it is the target of our design methodology.

2.1 Evaluation of Heterogeneous Reconfigurable Architectures

First of all, an analytical model is introduced based on [Deh96] to classify programming granularity of various traditional programmable and configurable processors. In Dehon's architecture evaluation methodology, three important variables are defined for an architecture: word length (w), program storage (c) and data storage [d] for each of the processing elements. By characterizing algorithmic properties such as computation word length (w_c) and path length (or program state, p), the resulting efficiency of applications implemented on specific architectures can be quantified (efficiency can be thought of as the implementation cost divided by the cost used directly to compute the algorithm). For example, FPGA's from [Xilinx] and [Altera] provide homogeneous and bit-level granularity [where $w=1$ and $c=d=1$]. State-of-the-art processors usually have $w=32$ or $w=64$ and $c=d=1024$. Figure 2-1 shows the efficiency of the two different kinds of architecture for a range of algorithm properties. The two graphs show that fine-grained architecture, such as FPGA (Figure 2-1a), gives the best efficiency for applications that have small word length (w_c) and small local states (p), and vice versa for programmable processors (Figure 2-1b).

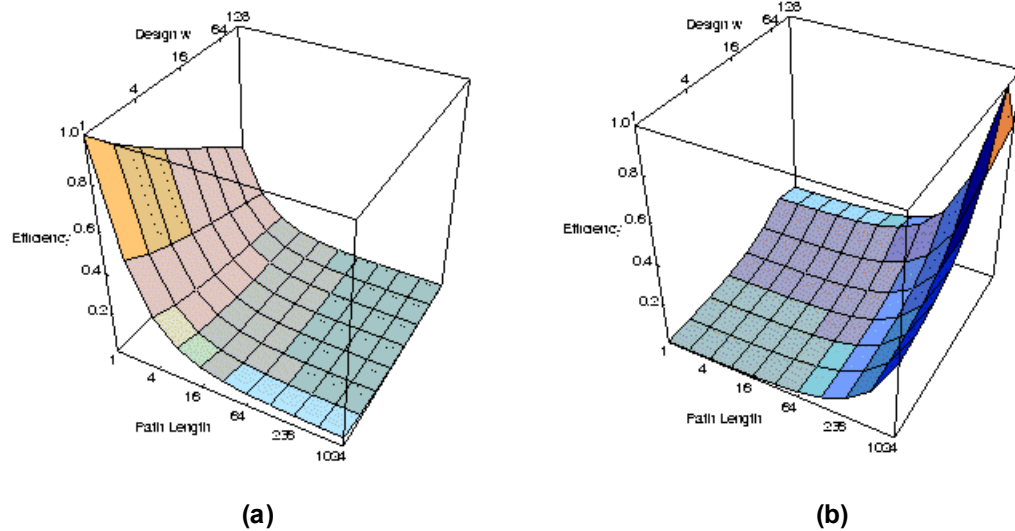


Figure 2-1 Efficiency Comparison of (a) a Reconfigurable Architecture with Small Granularity and (b) a Programmable Processor

This quantitative analysis of the architecture granularity demonstrates that it is important to understand the algorithm properties in order to choose a good underlying implementation platform. Current applications seldom possess homogeneous computational granularity or structure. For example, the base-band DSP processings for a wireless receiver often contain signal detection (word-level processing) followed by channel decoder (bit-level processing) and source coding (word-level processing). Even within a voice or video source coder, there are heterogeneous algorithmic structures (high-level control and regular data-flow computations) and computational granularities (address calculations versus word-length computations on data). Such heterogeneity in the algorithm encourages composition of heterogeneous reconfigurable architectures. To see this quantitatively, Figure 2.2 shows the efficiency of a heterogeneous architecture with a FPGA module (analyzed in Figure 2.1a) and a programmable processor core (from Figure 2.1b). Figure 2.2 only shows a cross section of the design efficiency assuming the

design word length is 4. The efficiency for each program state (or path length) is generated by taking the maximum of the two efficiencies (for FPGA and programmable processor) given the word length and program state. As the figure suggests, while FPGAs work best for algorithms with small program state and programmable processors achieve better efficiency for algorithms with large program state, the heterogeneous architecture can achieve best of the two worlds by providing higher efficiency in both cases.

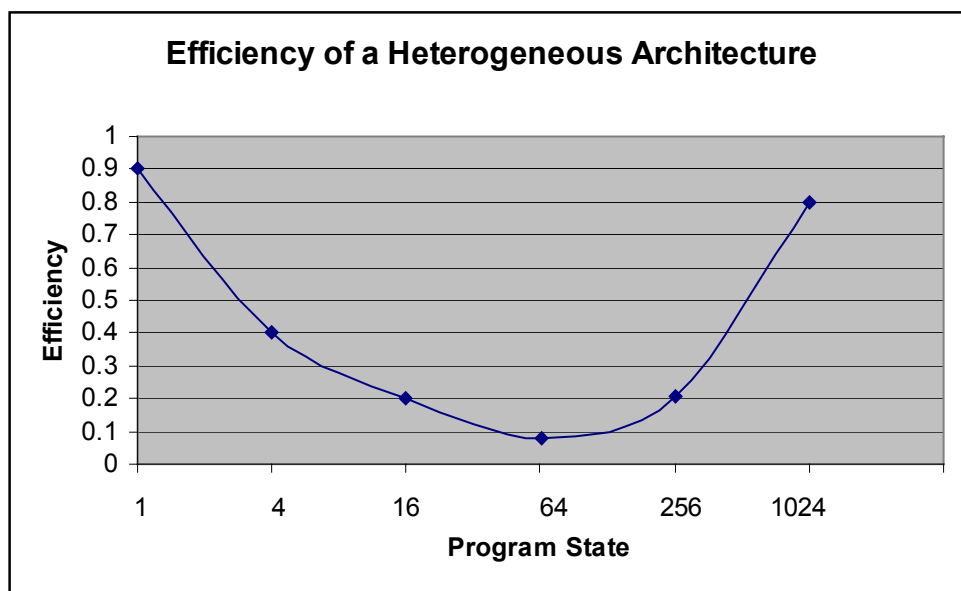


Figure 2-2 Design Efficiency of a Heterogeneous Architecture (Mixture of a FPGA component and a Programmable Processor Core)

In actual implementations of various digital signal processing algorithms, board-level heterogeneity already exists [RSF96] [CA96] [Villar95] to achieve better speed performance. As integration capability allows different IP cores to coexist on the same IC [Borel97], a new breed of heterogeneous reconfiguration processor ICs is emerging. Indeed, there are already significant research activities devoted to heterogeneous architectures: (NAPA1000) [GARP]. Commercial products, such as [Chameleon] and

[MorphICs), also present ICs that have mixed granularity. However, most of these architectures only target at most two kinds of granularities (general-purpose programmable processor and FPGAs) and both of them target general-purpose computing.

As shown in Chapter 1, ASIC components provide superior energy and performance efficiency. Therefore, if general-purpose flexibility is not needed, using application specific implementation gives better efficiency. Therefore, another important factor should also be considered in order to perform better in the P-D-A metrics: providing architectures of varying degrees of flexibility.

In the embedded DSP processing, many algorithms obey the 90-10 rule [Hen98], which states that approximately 90 percent of execution time is spent in about 10 percent of the code (often residing in tight loops). The flexibility required for the wireless communication or DSP applications is usually across algorithms within the same domain (for example, CELP-based speech coders, different filter algorithms for signal detection, and DCT based image/video processing). Within one domain, the 10% bottleneck usually possesses the same computational structure as well as granularity. Therefore, it pays to provide application-specific specialization (less flexible), especially within the same domain. Instead of targeting a general-purpose heterogeneous architecture with a general-purpose core and a general-purpose reconfigurable fabric, a new heterogeneous reconfigurable architecture style, which also includes application-specific computing elements in the architecture, is advocated in this thesis.

2.2 The Pleiades Heterogeneous Reconfigurable Architecture Template

The Pleiades [Abn96] architecture template is a representative of heterogeneous reconfigurable architectures with mixed granularity and flexibility. The architecture (shown in Figure 2-4) is composed of a programmable microprocessor and

heterogeneous computing elements (referred to as satellites in the rest of the thesis). The satellites can be of different granularity and flexibility (custom ASIC blocks, micro-programmed processor, or FPGA). The architecture template fixes the communication primitives between the microprocessor and satellites and among the satellites. The heterogeneous nature of the architecture decides that the architecture is “domain-specific”. For each algorithm domain (communication, speech coding, video coding), an architecture instance can be created that specifies the satellite types and the numbers of each satellite.

To reduce overhead of instruction fetch and global control for satellite computations, the architecture uses distributed control and configuration. To achieve distributed control, each satellite is equipped with an interface that enables it to exchange data streams with other satellites efficiently, without the help of a global controller. The communication mechanism between each satellite is data driven. The control means available to the programmer are basic satellite configurations to specify the types of operation to be performed by the satellite, and configurations for the reconfigurable interconnect to connect a cluster of satellites. All configuration registers are part of the processor’s memory map and configuration codes are set via memory writes from the processor’s point of view.

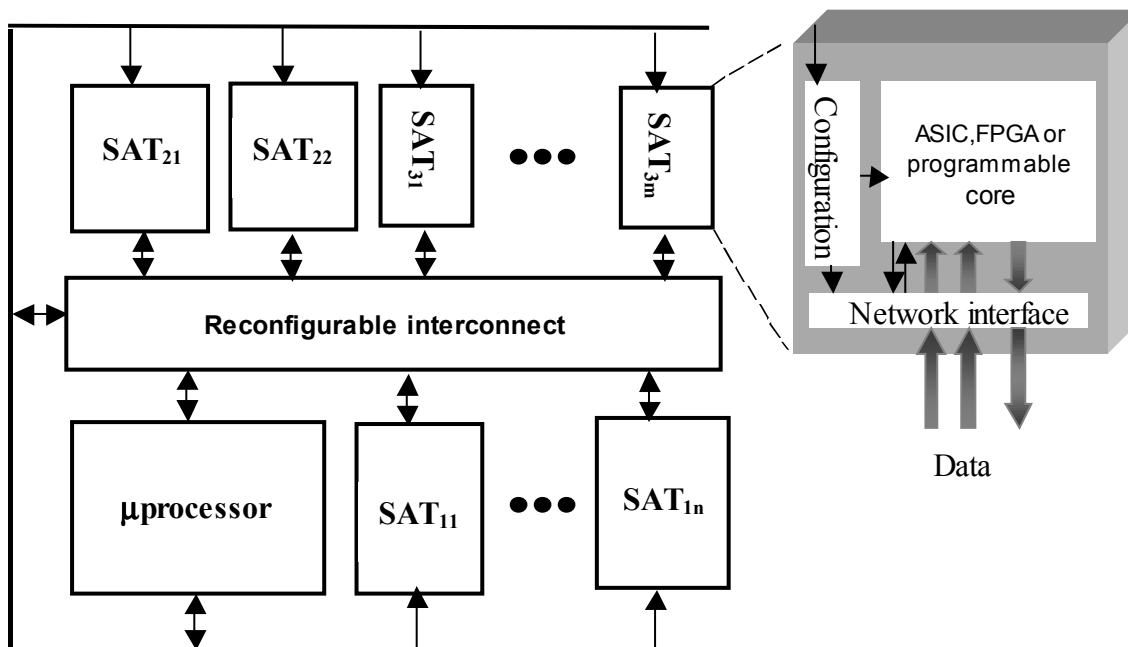


Figure 2-4 Heterogeneous Reconfigurable Architecture Template

Empirical quantitative results presented in Figure 2-5 give a first order idea of the superiority of the coarse-grained reconfigurable architecture in terms of an important metric (energy-delay product) [Abn98]. A variety of programmable/reconfigurable processors (ranging from RISC, CISC, fine grained FPGA to coarse-grained reconfigurable processor, such as the Pleiades) are evaluated using two typical DSP benchmarks: IIR and FFT. To give meaningful architectural comparison in this case, all energy and delay data are normalized to a reference of 0.6 μ m process with 1.5V as the supply voltage. As shown by the results in the figure, the Pleiades architecture gives more than one order magnitude of efficiency when compared to programmable RISC and DSP processors. These results show the energy efficiency of coarse-grained reconfigurable architecture for computations with mostly dataflow operations. However, these results did not demonstrate the effectiveness of heterogeneous reconfigurable architecture for more complex applications. In this thesis, we demonstrate a design methodology that allows us to map an entire application (including dataflow and

control-flow) to such a heterogeneous architecture, and show the energy efficiency superiority in this case as well.

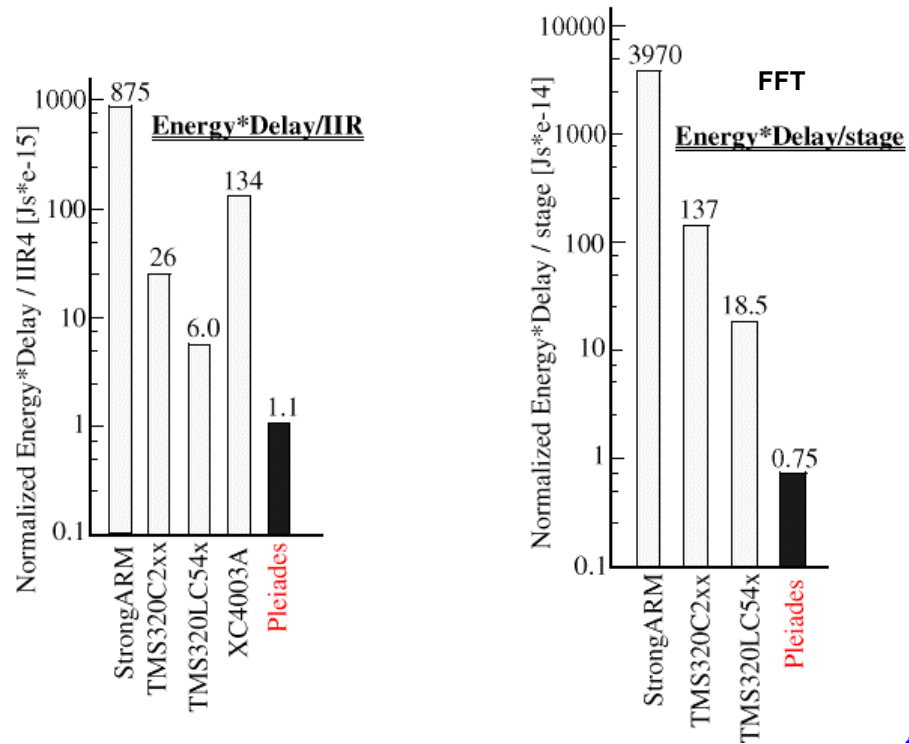


Figure 2-5 Energy-delay Products for DSP Benchmarks Implemented on Architectures of Different Granularity and Flexibility

3: Design Methodology for Heterogeneous Reconfigurable DSPs

Chapter 2 has established that heterogeneous reconfigurable architecture is a promising implementation platform for the application domains of interest: wireless communication and digital signal processing algorithms. In this chapter, we first investigate a plethora of design environments that have the potential to assist in using heterogeneous reconfigurable architectures in the embedded digital-signal processing domain. We evaluate each of the environments in terms of its capability to deal with architectures of different granularity and flexibility as well as its ability to evaluate energy and performance metrics for various mappings. We show that there is no existing design methodology to bridge this algorithm-architecture gap. Therefore, at the end of the chapter, a proposal of such a methodology is given for the design of such an architecture as well as compilation to an existing architecture.

3.1 Survey of Hardware-Software-Reconfigurable Co-Design Methodologies

The goal of this research is to develop a design environment that takes the digital-signal processing algorithms specified in a high-level language and performs hardware-software-reconfigurable partitioning and synthesis under real-time performance constraints and optimization objectives (such as low-energy consumption).

Part of this problem involves co-design of hardware and software, which has drawn much attention in the past few years in the VLSI computer-aided design (CAD) community. The examples are commercial tools such as Virtual Component Co-Design (VCC) from Cadence and Napkin-To-C (N2C) from Coware and research work done by [Kal95] and [BPM97]. VCC provides a good encapsulation environment for designers to

incorporate architectures of their own choice, and the tool provides performance feedback for the hardware-software partition provided by the designer. However, the designers have to do the partitioning and estimations themselves. The designers still need faster ways to characterize their specified algorithms and target architectures. N2C provides a good way to synthesize interfaces between hardware and software, but currently does not address anything else. Existing hardware-software co-design methodologies in the research world [Ka195] [BPM97] attempt to partition and synthesize hardware and software for systems specified in system-level languages using custom developed co-simulation and estimation methods. However, the partitioning and synthesis part of these hardware-software co-design tools do not target reconfigurable components. In addition, the design metrics of interest are mainly timing and area, but the energy metric is not addressed.

Recently, several VLSI CAD co-design tools [Dav99][DiJ98] have considered reconfigurable architectures as part of the heterogeneous components to target. The main approach is to have the algorithm divided into tasks, and the assumption is that the implementation (in several different architecture styles, such as ASICs or programmable processors) of each task is present. The focus of these environments is to perform task-level scheduling, and to conduct quantitative partitioning across ASIC, processors, and FPGAs accordingly. However, the reconfigurable architectures addressed are mainly off-the-shelf fine-grain FPGA components. In addition, these methodologies are constrained, because no synthesis path is provided to map tasks without an implementation library.

Besides the research efforts from the VLSI CAD field, many research activities involving heterogeneous reconfigurable processors are ongoing in the general-purpose computing community. The focus of the research activities here is somehow reversed. The target architecture has always been a composition of processor with reconfigurable

accelerators. Recent efforts attempting to solve the design problem have concentrated on fast compilation from high-level language to the target architecture. For example, [CAL98] has proposed a fast compilation path to a MIPS processor with reconfigurable data-path coprocessors [Hau97]. [KYI97] has introduced a similar method to [CAL98] for a heterogeneous processor with a fine-grained FPGA as the coprocessor. However, none of the approaches performs analysis on the resultant partitioning or synthesis, so the advantage of moving a computation to a reconfigurable architecture is often offset by the overhead of reconfiguration. In addition, the usual assumption is that there is already an underlying architecture causing the design aspect of an architecture to not be addressed at all.

The desired methodology must meet the following needs:

1. Targeting architecture with mixed granularity and flexibility (reconfigurable modules are a must).
2. Addressing the energy metric.
3. Providing optimizations for reconfigurable modules.
4. Having the ability to quickly synthesize to an architecture style to obtain fast performance feedback.

Table 3-1 shows whether or not many current system-level tools have to ability to address these needs.

Table 3-1 Summary of Features of Selected System-level Design Tools

	[Ka195] [BPM97]	[Dav99] [DiJ98]	[CAL98] [KYI97]	[This Thesis]
Target Reconfigurable Modules		√	√	√
Use Energy as Metric				√
System Optimizations	√	√		√
Synthesis of Tasks			√	√

Therefore, it can be deduced that no methodology exists that allows the designers to take an algorithm specified in a high-level language, partition it to architectures of various flexibility and granularity, get the power as well as delay feedback, and then perform system-level optimizations accordingly. A methodology that will address all of these issues is introduced in the following section, and is described in more detail in the rest of the thesis.

3.2 Overall Flow of the Design Methodology for Low-Energy Heterogeneous Reconfigurable DSPs

3.2.1 Assumptions

In the methodology proposed by this thesis, the applications of interest are real-time periodic digital signal processing for embedded applications. Therefore, meeting the real-time constraints while reducing total power (or energy) consumption is an important goal.

As mentioned in the end of last chapter, the target architecture style is a set of domain specific heterogeneous reconfigurable architectures. The Pleiades architecture described in Chapter 2 is used as our demonstration vehicle.

We envision an instance of the Pleiades architecture to have the ability to support multiple application threads (see Figure 3-1). In this case, a real-time operating system will run on the microprocessor to take care of scheduling computations to the reconfigurable resources, and to make sure that real-time constraints are met for all threads. In Section 3.1, several existing methodologies already investigate the scheduling problem, assuming that each task has been mapped to different architectures. Therefore, the focus of this research will be on mapping and synthesizing one single thread to a heterogeneous architecture instance. Figure 3-1 shows the model of computation that is supported by our methodology. We assume that an application thread is initiated on a programmable processor, and computations can be spawned off to clusters of reconfigurable satellite processors (the “split” point in Figure 3-1). After all computations are finished on the satellites, control is transferred back to the processor at the join point. Multiple spawning occurs within an application thread.

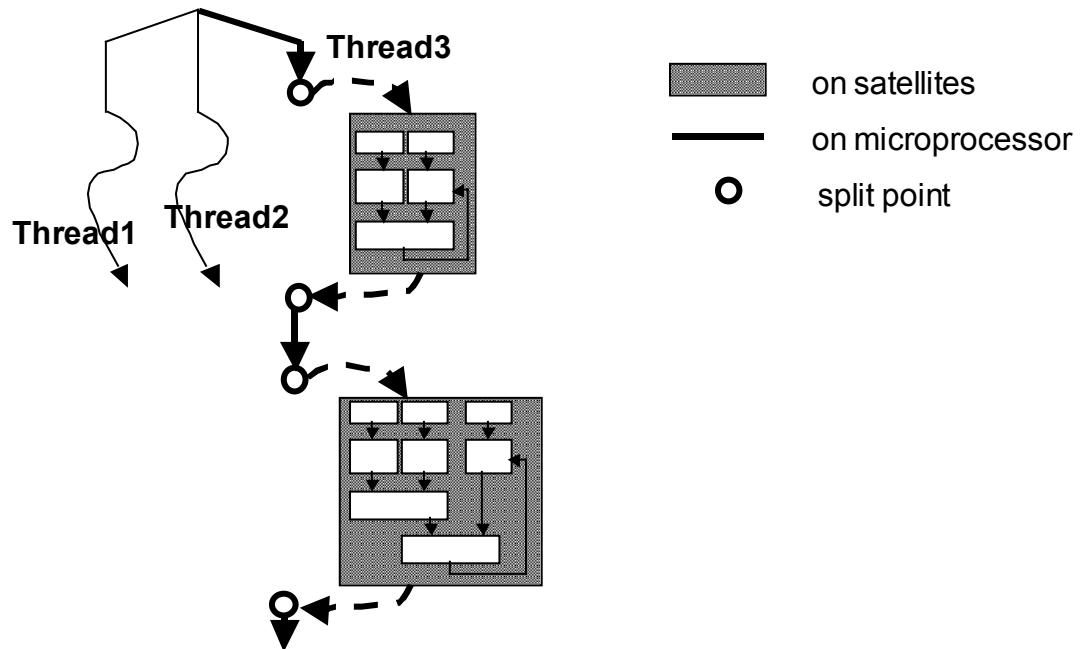


Figure 3-1 Model of Computation for the Architecture

3.2.2 Overview

Given the model of computation and the underlying architecture template, two key issues must be resolved in order to make a methodology useful for the designers. First, the architecture combines two very distinct models of computation, control-driven computation on the general-purpose microprocessor, and data-driven computing on the clusters of reconfigurable satellites. Therefore, the goal of the architectural exploration process is to partition the application over these two computing paradigms so that performance and energy dissipation constraints are met (referred to as *issue 1* later). Second, optimizations related to reconfigurability have to be supported at both the architecture instantiation path as well as compilation path (*issue 2*).

The basic flow of the design methodology is presented in Figure 3.3. Two terminologies used in the methodology stages are defined here:

Kernel: A computationally intensive part of the algorithm that often resides in nested loops.

Macro-model: A quantified view of the cost functions of the design module in terms of its parameters and constraints.

In stage 1 of the methodology, potential kernels of the algorithms are identified. All kernels are mapped to different architectures and their timing and energy costs are recorded during stage 2. In stage 3, a global partitioning across software-hardware-configware is determined based on the information gathered at stage 2. These first three stages in the methodology ensure that a good partitioning is obtained (addresses *issue 1*). In the final step, optimizations concerning reconfigurable architectures are carried out (addresses *issue 2*). Both of these issues require careful modeling of the algorithms and the underlying heterogeneous architectures, which are given by the combination of the profiling and the architecture macro-model libraries (these two modeling techniques will be discussed in Chapter 4).

Given the system-level description as input, the architecture template given in Chapter 2 allows the designer to have two kinds of output. In the first case, when a set of algorithms are given, the architecture instantiation path is followed to create an instance of an architecture. In the second case, the compilation path is used to generate software to be run on a given instance of an architecture. The proposed basic methodology flow is used as the core for both paths. An expanded overview of the basic design methodology is presented in section 3.2.3. The detailed description of each one of the methodology stages is given in Chapter 4 through Chapter 7 of this thesis. Steps that are specific to the architecture instantiation path or the compilation path will be noted.

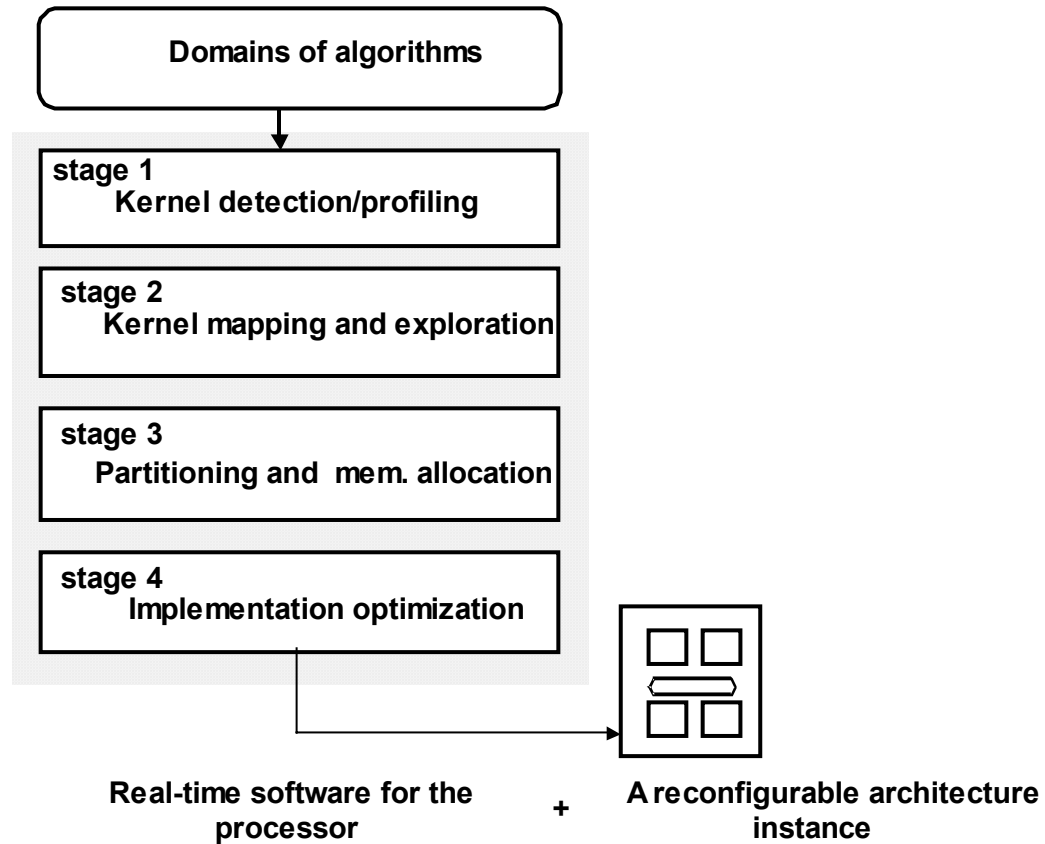


Figure 3-2 The Steps of the Design Methodology

3.2.3 Description of Each Stage

The important steps of the methodology are further expanded in Figure 3.3 and are described in detail in the remainder of this section.

The methodology flow takes DSP or communication algorithms specified in a high-level language (for example, C) as input. The initiation of the design process requires the establishment of a first-order baseline model of the algorithm complexity and bottlenecks. Such a model allows for the selection and execution of architecture-independent optimizations (stage 1A). As architectural choices have not yet been made,

this model assumes the presence of a “virtual architecture” with some generic operator costs attached to it. Optimizations at this stage only address either win-only situations or order-of-magnitude improvements, so absolute accuracy is not that important.

After a satisfactory algorithm formulation is obtained, the architectural mapping and partitioning process is entered. To be meaningful, the partitioning process should be based on realistic bottom-up information regarding the cost of implementing functions and operations on the different architectural choices. Our design-exploration methodology relies extensively on the availability of Power-Delay macro-models for all components in its architectural library (stage 1B). The estimation methods employed in each of these models vary depending upon the type of the module and the desired accuracy. While the absolute accuracy of these characterizations is not crucial, it is important that bounds on the prediction accuracy are known. “Improvements” that fall within the noise level of the estimations should be treated warily.

The architecture partitioning and mapping process is started by establishing an initial solution. Given the implementation simplicity of a pure software implementation, we have adopted a “software-centric” approach that assumes that the whole algorithm is initially mapped onto the core processor (stage 1C). The software-centric approach establishes how close such a solution adheres to the design specifications and helps to establish the design bottlenecks. A rank ordering of the dominant compute kernels is determined. Dominant kernels are evaluated in order of importance. If a hardware implementation is deemed worthwhile, a repartitioning of the design is established (stage 2 and stage 3). At stage 2, a synthesis tool is provided to map a kernel to the reconfigurable satellites. The synthesis tool performs a direct mapping of the kernel and gives the power, delay and area estimates of the mapping. Several algorithm-level transformations can be utilized at this stage (such as software-pipelining and loop unrolling) to improve the final implementation.

After all costly kernels are mapped to accelerators, a final partition of the algorithm across different architectures is obtained. While the rest of the algorithm remains as high-level language, the portions of the algorithm to be implemented by satellites are specified in an intermediate form that is capable of modeling the structure of the reconfigurable satellite operations (i.e. as a net-list). Based on this conceptual net-list, implementation optimizations (stage 4) are invoked to choose a good reconfigurable interconnect architecture (during architecture design path) and to generate efficient configuration and interface code (during compilation path).

At different phases of the design phase, it is important to show the impact of particular design choices on the overall performance and energy of the application in order to give meaningful design guidance. A spreadsheet-like environment proposed in [Lid98] does precisely that and it is utilized in our methodology.

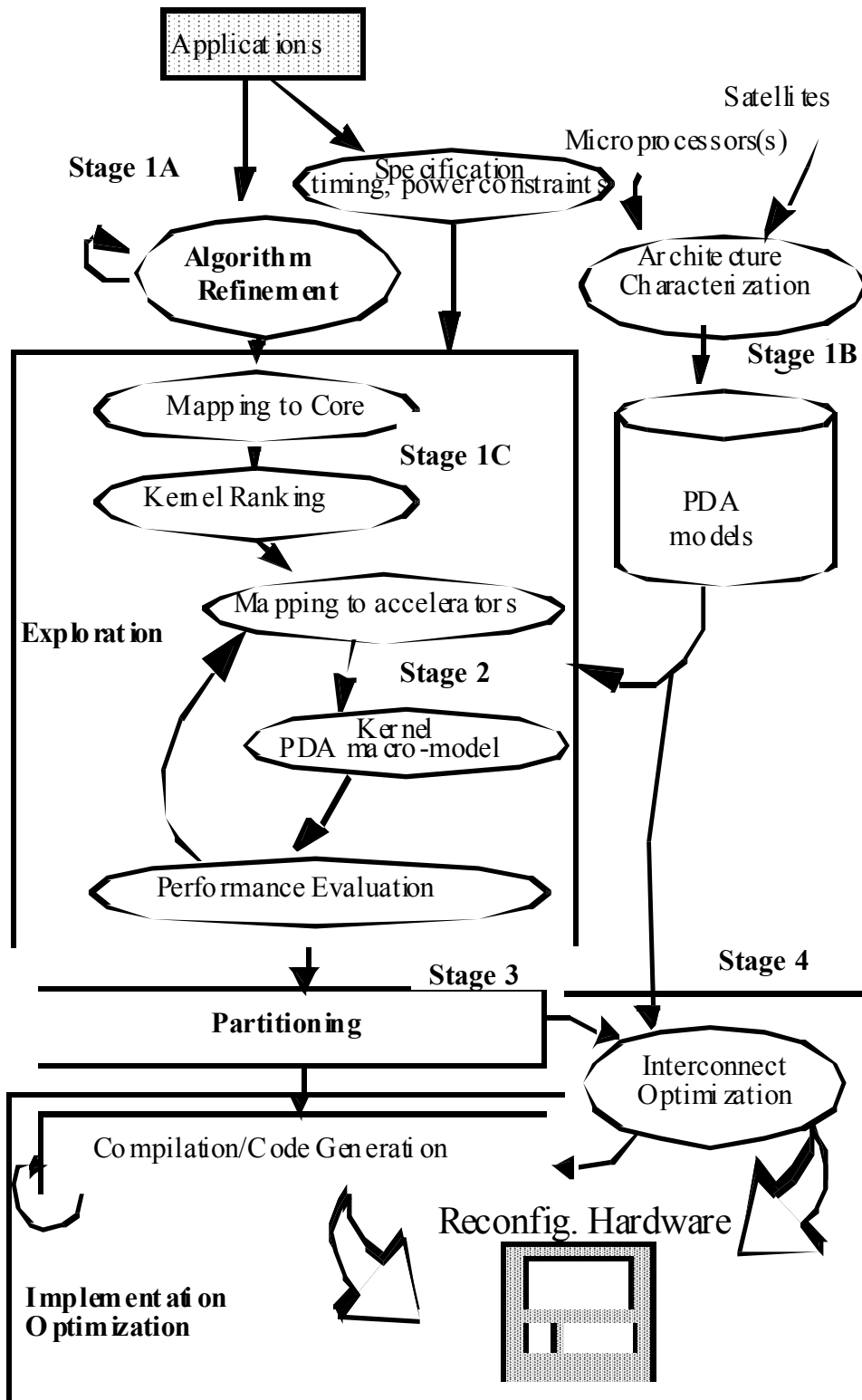


Figure 3-3 Design Methodology Flow for Low-Power Heterogeneous Reconfigurable DSPs

4: Algorithm Characterization and Architecture Modeling

Since more and more designs of complex embedded digital systems are composed of mixed ASIC, programmable cores, and reconfigurable hardware, the conventional design methodology that starts with an already hardware-software-reconfigurable partitioned system is no longer effective. It is necessary for the new methodology to have an architecture-independent specification of the algorithm first. The next step is to explore different possible architecture partitions until final hardware-software-reconfigurable boundaries are decided. The importance of using a methodology that combines algorithmic and architectural information in this exploration and design process is emphasized in this thesis. This chapter discusses the foundation upon which the central part of the design methodology is based. First, this chapter introduces the concept of using efficient and effective estimation of different accuracy levels to achieve better overall system architecture design (hardware-software-reconfigurable partitioning, software, hardware architecture optimizations etc). At the end of this chapter, an example that demonstrates how the various techniques in this chapter are used to achieve stage 1 and stage 2 of the design methodology is given.

4.1 Fusing Architecture and Algorithm Characterization for Estimation and Exploration

During the design of complex embedded hardware or software systems, the decisions made at higher levels (algorithm or system level) have a larger impact on the final implementation [Lid98][Kie99] in contrast to changes made at the RTL or circuit level. Ironically, the most difficult part of the design process is to discover the best transformations and optimizations needed at these high levels. The main reason is that

the design space covered at these earlier stages is large, which makes searching for a good solution a time-consuming task. In addition, the resultant effect on the final implementation is often difficult to predict at the system or algorithm level if good architectural and algorithm mappings are not provided. Therefore, the best way to an optimized final implementation is to first devise good estimation techniques to guide design explorations at higher levels of design abstraction, instead of brute-force search and optimization. Several previous research projects that have the similar concepts are the Y-chart approach proposed in [Kie99], design guidance using algorithm property extractions for high-level synthesis in [Gue98], and compiler transformations based on loop properties described in [KeP94].

The work by [Kie99] explores the design space by rapidly mapping tasks in the algorithm to nodes in a dataflow architecture instance, and collects performance numbers by instantiating architecture-based simulators based on the mappings. [Gue98] and [KeP94] employ analytical and empirical models based on static properties of the algorithm. While all of these works provide efficient high-level design guidance, each of them only addresses one type of architecture (custom ASICs, or instruction-set based processors). For the problem that this thesis addresses—the design of heterogeneous processors, the final implementation of interest contains both hardware and software, thus the design space is even larger. This problem translates into a stronger need for better guidance during the design process. In this methodology, it is proposed to divide the entire design flow into four steps to ease this algorithm to architecture transition (described in Chapter 3 and Figure 3-3). The steps are namely algorithm selection/improvement, algorithm exploration, algorithm partitioning, and eventually architecture improvement steps. To achieve the goal effectively, it is important to *quantify* the effects of mapping an algorithm down to different target hardware, software and reconfigware implementations by characterizing properties of the algorithm and architecture.

One important thing to note is that the amount of information and accuracy level needed from algorithm and architecture characterizations are different at each step. Figure 4-4 shows that the amount of information needed from the algorithm characterization and the architecture modeling stage is depending on the design abstraction level (where thicker arrows imply more information with higher accuracy). As we go down the design abstraction, we need more detailed information from algorithm (for example, the amount of parallelism in addition to operation counts) and architecture (for example, detailed interconnect model in addition to satellite models). The key is to have a method to get the high-level algorithm and architecture information as fast as possible so we can explore different options at step 1 and step 2, where most improvements can be made.

The remainder of this chapter concentrates on the techniques and models used for algorithm and architecture characterizations at the early stages. Specifically, in this chapter, we talk about the algorithm profiling techniques used at the algorithm selection and improvement stage, and the kernel identification stage. Different architecture modeling techniques used at the kernel extraction and kernel mapping level are introduced as well. At the end of the chapter, the starting point (step 1 and part of step 2) of the design methodology is presented. The more detailed characterizations are presented in Chapter 5 and Chapter 7 when kernel mapping and architecture optimizations are discussed. Much of the information generated at these steps is also used while making decision at the later steps since all optimizations made later have to be justified and triggered by system bottlenecks to avoid unnecessary optimizations.

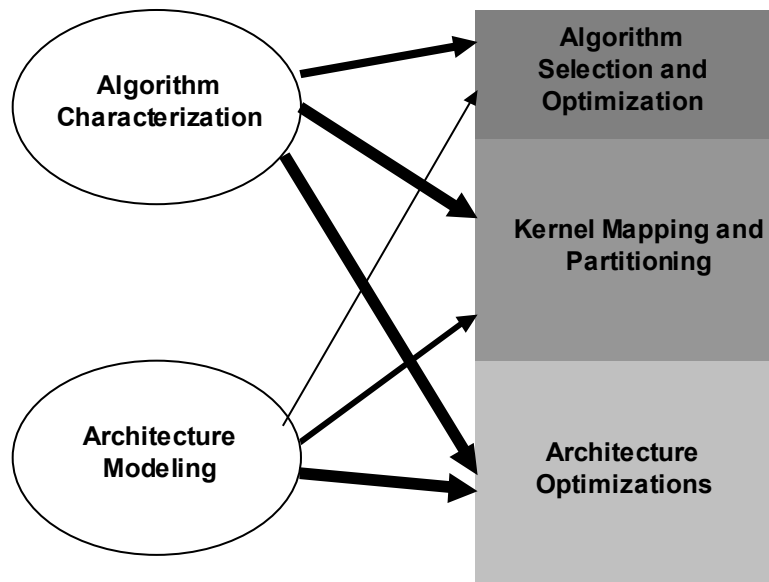


Figure 4-4 Use of Different Characterization and Modeling Techniques for Different Design Stages

4.2 Algorithm Characterizations

After an algorithm is decided upon, the first useful implementation-dependent information to know about the algorithm is the *inherent complexity* (multiplication, addition, array accesses, or some other operations defined by the user) depending on some virtual machine model (costs on each operation). The necessary information is best obtained through a combination of dynamic profiling (loops and conditionals) and static analysis (within basic blocks) of the high-level specification. *Static analysis* has been used in performance estimation to find out the worse case timing within blocks without too much control-flow. *Dynamic profiling* (through simulation) has been used widely to detect hot spots in an algorithm running on a specified target machine. For example, a utility provided in commercial tools, such as Purify, gives the program designer a percentage breakdown of cycles for each function using dynamic profiling.

The drawback of these tool is that the results are “host machine” dependent. Recently, this information has been used in collecting basic block frequency information in a more architectural independent way [Cal97][Wan98].

A tool, `source_profile` [Wan98], which combines both dynamic and static profiling of algorithms, is developed in conjunction with the methodology of this thesis. Instead of dynamically collecting cycles, the tool records the number of invocations of each function and each basic block. This dynamic information is combined with static analysis of the operations in each basic block to provide overall operation counts for the entire application. This algorithmic characterization technique is used in several places in our design methodology. Three of the most important applications of the tool are detailed in the following sections (Section 4.2.1 to Section 4.2.3). The details of the tool is described in Section 4.2.4.

4.2.1 Application 1: Algorithm Selection

The first application of algorithm profiling is to provide inherent computational complexity for algorithm selection. In the system design process of a signal detection processor [Zha99], five multi-user-detection (MUD) algorithms are considered with varying error convergence rate. In the traditional system design process, the algorithm designers either do not consider the implementation complexity during algorithm selection, or have to rely on manual mapping of each algorithm. The tool, `source_profile`, is used in this algorithm selection stage to provide insight for the designers. The output of the profiler is shown in Table 4.1. As shown in Table 4.1, algorithm 4 is a more complex algorithm and should be chosen only if the system performance (convergent rate for a single system) is orders of magnitude higher.

Table 4-4-1 Complexity Profiling for Algorithm Selection

	Algorithm1	Algorithm2	Algorithm3	Algorithm4	Algorithm5
Computational Complexity (operation counts)					
MULT	124	248	496	228,656	736
ALU	124	252	502	237,708	800
MEM	248	620	1,240	642,754	2,120
Total	496	1,120	2,238	1,109,118	3,656

4.2.2 Application 2: Kernel Detection

Modern DSP algorithms such as voice and video CODECs or communication algorithms combine regular but computation-intensive kernels (typically the dominant factor) with irregular control functions. While the control functions are best implemented on a traditional programmable processor architecture, the kernels present ample opportunity for design optimization by exploiting hardware accelerators.

It is important to identify potentially power-hungry portions of an algorithm as early as possible. The inherent computational complexity (counts of basic operations and memory accesses) provided by `source_profiler` is a meaningful measure to identify dominant kernels [Wan98]. Since operations do not have a uniform cost, a weighted sum (where the weights indicate the energy allocated to each operator in the abstract implementation model) is calculated and aggregated at either the function or basic block level to indicate the power consumption trend within the application.

Based on this information, the designer can rewrite the code to either reduce the inherent cost of the algorithm, or extract kernels into procedure calls. These procedure calls are then considered for potential hardware acceleration.

4.2.3 Application 3: Kernel Frequency Record for Design Evaluation at the System Level

As mentioned in Section 4.1, it is also important to evaluate effects of architectural optimizations on the overall system performance. The profiler tool is used to record kernel frequency, and the information is preserved and incorporated into the overall system macro-model to help the system level evaluation during later stages of the methodology.

4.2.4 Development Environment Details

This algorithmic refinement and kernel extraction process is based on a combination of commonly available tools. The algorithm is simulated with appropriate input vectors (representing standard operation), and profiling information is gathered at the basic block level (for example, using the `gcc -a -g` options combined with post-processing). The profiling frequency is back-annotated to the source code to provide a first-order complexity-breakdown. The basic-block execution frequencies obtained from the profiler are combined with a breakdown of the blocks into basic operations, as provided by standard compiler parser front-end, the Stanford Unified Intermediate Form (SUIF). The SUIF compiler is used because of the modularity of the design. The SUIF compiler (as shown in Figure 4.2a) consists of several modular passes which include parser, high-level optimizer, machine code optimizer and code generator. The intermediate format that our tool, the `source_profiler` module, works with is the high-SUIF format. `Source_profiler` (Figure 4.2b) treats source-level procedures, basic blocks, instructions, and expressions as traversable trees (Figure 4.2c) and performs a depth-first search on the tree to cover the entire source representation.

The tool generates a detailed and illustrative overview of the distribution of the algorithm complexity over basic operators and memory accesses.

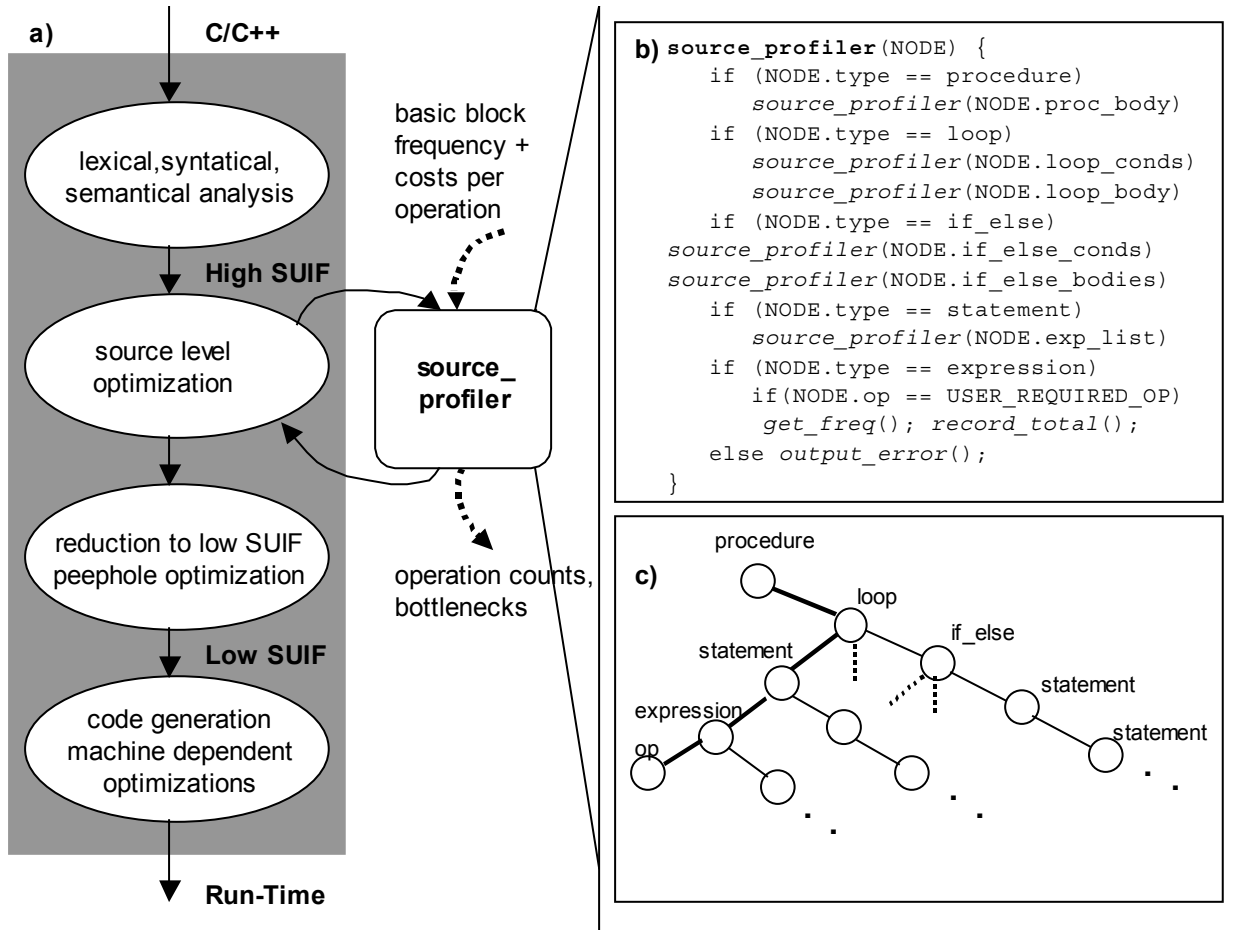


Figure 4-4-1 a) The Interaction between the Source Profiler and the SUIF Environment
b) The Pseudo Code c) The High SUIF format that the Profiler works with

4.3 Architecture Characterizations

Another important aspect of the methodology proposed in this thesis is the ability to have good architecture characterizations combined with effective techniques to bind mappings to heterogeneous architectures in a system model. In this section, we are only concerned with power models for the different architectures, since significant research exists that investigates the area and timing metrics.

The technique of macro-modeling is used since macro-models allow any estimation technique to be encapsulated, and they allows us to deal with an underlying architecture that includes heterogeneous components. A macro-model produces a series of outputs as a function of a series of inputs. To get the output, the model can be in the form of either a fixed number, a table, a set of parameterizable equations, an invocation of a tool or can be composed of a set of other macro-models. Macro-models serve as the system-level building block structure to enable estimation of heterogeneous designs.

A successful system modeling technique relies not only on the use of macro-modeling (it provides us with the encapsulation tool), but also on a careful characterization of the underlying architectural choices. It is important to have power and performance model of different architectures in the template. For the rest of the chapter, we introduce several modeling methods for the various components in the target architecture and discuss how macro-models can be built using these primitives. The components are: instruction-set based core processor architectures, ASIC satellite computational modules, reconfigurable interconnection, and reconfiguration/interface between the processor and the satellites.

4.3.1 Software on Embedded Programmable Processors

Processor simulation performed at the Register Transform Level (RTL) has been used to get a very accurate power estimate for an application [PBB99]. However, RTL simulation is time consuming and is always about orders of magnitude slower than an instruction-set simulator [GFT99] written in a high-level language.

Instruction-level modeling has emerged as the preferred method for characterizing the energy consumption of software running on general-purpose cores [Tiw94] because of its reduced time-to-result and reasonable estimation accuracy. The technique is to

first build up a database of energy base cost for each instruction. The energy base cost of each instruction is obtained by either physical measurements (for example [Tiw94]), physical or RTL-level simulations, or vendor supplied technology notes [Texas Instrument]. After the total energy is collected, the model includes a scaling factor that introduces the effects of frequency and voltage to allow architecture exploration. Given a piece of assembly code, the total energy consumed is approximately the sum of the base costs of the executed instructions and other overheads (Equation 4.1). In the equation, B_i is the base cost of an instruction i , and N_i represents the number of invocations of instruction i .

More accuracy can be obtained by adding inter-instruction effects (O_{ij} between instruction i and j), as well as correction factors for pipeline stalls and cache misses (E_k).

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \quad (\text{Eq 4.1})$$

To capture the dynamic behavior of an algorithm, these instruction-level based energy and the correction factors can then be stored in a database and used by a profiler. The profiler can trace executed instructions, record the base energy for each instruction and perform book-keeping of the energy for not only the entire application but also each function. After some post-processing, an illustrative breakdown of energy cost for each high-level function is presented. Figure 4.3 shows the flow and the important components for such an energy-profiling framework.

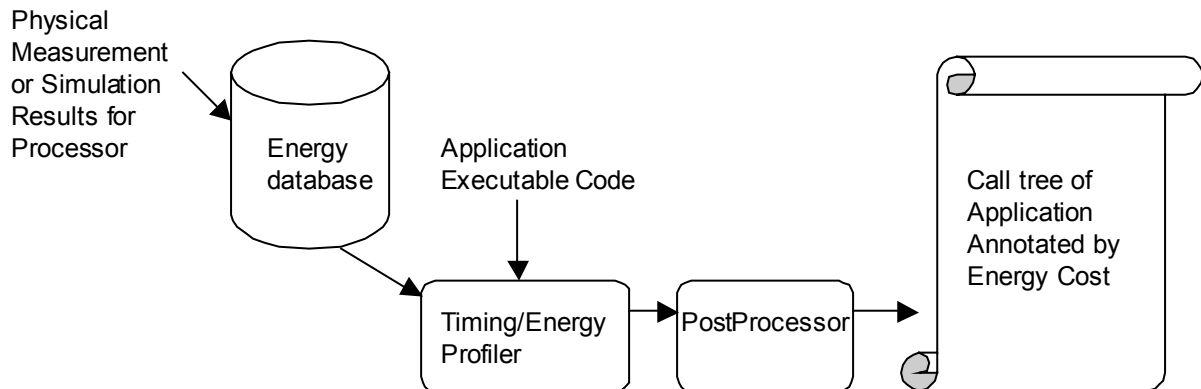


Figure 4-4-2 Profiling Environment for Programmable Processor Cores

Currently, as part of the development of this framework, we have evaluated the instruction base-costs and pipeline stall effects for several ARM cores [StrongARM and embedded ARM8] through physical measurements performed on ARM evaluation boards as well as simulations in PowerMill [reference] with processor net-lists. Table 4.1 shows a sample list of base energy costs for instructions of an embedded ARM8 microprocessor at 1V running at 25 MHz [Omad99]. In addition, a profiler (provided with free distribution of ARM SDK2.1) is modified to record the energy at each instruction boundary to trace the dynamic behavior of the algorithm.

Table 4-2 Basic Energy Cost and Cycle Counts for Each ARM Instruction

Instruction Type	Base Energy Cost (pJ/instruction)	Cycles per Instruction
ADD	39.56	1
CMP	47.60	1
MUL	100.62	2
MLA	102.24	2
LDR/STR	87.98	2

For RISC processors, the overhead factor (O_{ij} in Equation 4.1) can be considered as a fixed number and added as an application-level overhead after the total energy has been computed. Since the StrongARM and ARM8 processors evaluated in this thesis work are both RISC processors, we verified that the overhead is indeed a fixed number (less than 10%) [Lar98] and used it in our energy profiler. The cache miss factor is ignored since predictability of memory access is required for most DSP applications. Therefore, even though the error factor varies depending on the type of processors, in our target architecture, we are only concerned with the base energy and a fixed overhead.

4.3.2 Satellite Modules (ASIC components)

The power characterization of the satellite modules is somewhat more complex due to the variability of the units, differences in levels of programmability, and influence of factors such as signal correlations. However, moving components from software to hardware is only worthwhile when it results in substantial improvements, making some estimation inaccuracy acceptable. Adequate macro-modeling techniques have been developed for parameterizable functional modules, such as memories, multipliers and ALUs [Lid97]. To develop a macro-model for a satellite module, analytical models of the effective switching capacitance C_{eff} are derived from circuit-level simulations, hence including effects such as short-circuit currents. Given the unknown nature of the signal statistics in a given accelerator configuration, a (pessimistic) white-noise signal distribution is assumed during characterization. The total energy spent by a given functional module in a system is then modeled as:

$$Energy_{FU} = C_{eff} \times N \times V_{dd}^2 \quad (\text{Eq 4.2})$$

where N equals the number of accesses of the module. The units for C_{eff} and V_{dd} are pF and V respectively.

4.3.3 Satellite Modules (Embedded FPGA)

For fine-grained programmable units such as embedded FPGAs, accurate energy estimations can only be obtained after mapping, placement, and routing are performed. Currently, all of the functions we are implementing on FPGAs are small data-path elements (such as arrays of adders and comparators). Therefore, manual placement and routing are performed and energy is collected from simulating the functions on a low energy FPGA [Var00] using Powermill. In general, for larger functions, it is advisable to develop characterized libraries of macro-modules to avoid time-consuming mapping and analysis steps to be performed during high-level exploration. One way to use such macro-modules is to perform fast placement and mapping at the algorithmic level [Cal98].

Table 4-3 shows a sample set of characterized satellite modules. All computational satellites operate on a data bit-width of 16-bit. The data is collected for satellites running at 1V, so the energy model includes the C_{eff} and V_{dd} factor.

Table 4-3 Energy and Timing Statistics of Satellites

Satellite Type	Operation	Energy (pJ/op)	Throughput (ns)
MAC	Multiplication	21	24
MAC	MAC	26	24
ALU	ADD	7	20
Memory (512K)	Read	7	11

Address generator	Sequential address	6	20
Embedded FPGA	Theta	14	25

4.3.4 Interfaces

While the processor core and satellites contribute to the computation engine of the architecture, the interfaces that connect the processor and satellites have to be carefully modeled, since they act as communications between the major computation engines. There are two main interfaces that are modeled—the processor-satellite interface and the intra-satellite interface.

Processor Satellite Interface and Configuration

Another important cost is the cycle time and energy spent in reconfiguring each kernel and interfacing between hardware and software modules. Reconfiguration overhead is modeled as memory writes from the processor's perspective.

Intra-Satellite Interconnect

The reconfigurable interconnect module between satellite processors can consume considerable energy. Models for estimating interconnect power have been proposed at the algorithmic level [Meh94]. Given the size and number of the functional modules in the accelerator network, the average length and capacitance of each wire is computed. The total energy is estimated using the following expression:

$$Energy_{INTER} = \left(\sum_{i \in network} C_i \times V_{dd}^2 \right) \times N_{inter} \quad (\text{Eq 4.3})$$

with N_{inter} , the number of accesses to the network. The units for C_{eff} and V_{dd} are pF and V respectively.

While the above interconnect model is very suitable for algorithmic-level estimation and optimization, it is not adequate for architecture-level optimizations. Therefore, a more sophisticated interconnect model is developed based on the detailed module floor-plan and reconfigurable interconnect architecture, which will be addressed in detail in Chapter 7.

4.4 Starting Point of the Methodology for Heterogeneous Reconfigurable Architecture: A Software-Centric Approach

As shown in the overall design flow in Figure 3.6, a refined algorithm design (selected among different algorithms and with kernels extracted into procedures) is expected after stage 1. Algorithm profiling tools described in Section 4.2 are provided for this purpose.

After an optimized algorithm is obtained and potential kernels have been identified, the path down to implementation is started. Given the desirability of a software implementation, it is a natural choice to first explore the power and timing performance of the application when implemented entirely on the processor core. The power and timing information is obtained by combining the refined algorithm obtained from stage 1 with the microprocessor energy (timing) estimation provided by tools from stage 2. These costs establish a baseline to measure the impact of architectural optimizations. It is also used to evaluate compliance with timing and energy constraints and the range of improvement that has to be obtained.

Because of the inefficiency of most general-purpose processor cores, some of these constraints may not be satisfied. To identify areas of substantial improvement, a relative ranking of the dominant kernels is established. Using the dynamic instruction-

level code profiler (described in Section 4.3.1), a percentage breakdown of the energy consumed by each function (and basic block) is extracted and presented in a hierarchical call-tree format [Figure 4-4]. A kernel can only reside at leaf nodes and its energy and performance models can be changed to represent different implementation choices.

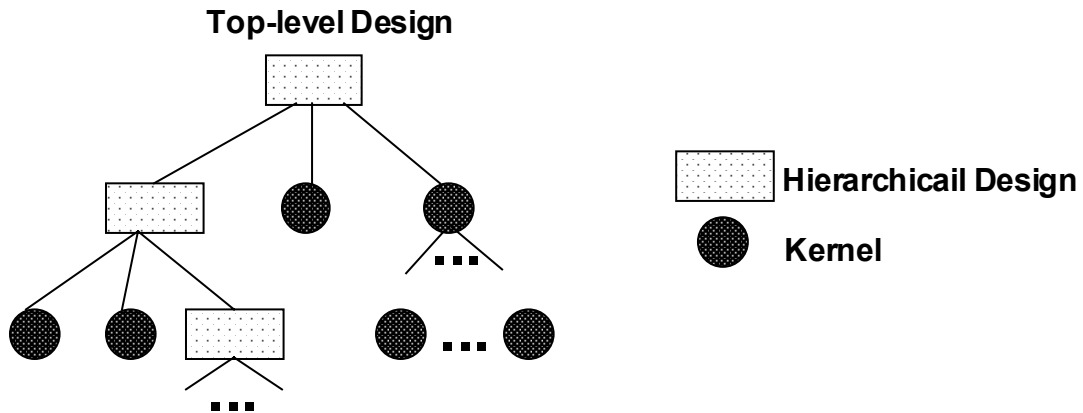


Figure 4-4 Conceptual Representation of the Design

4.4.1 Results

This section shows an example of stage 1 and stage 2 of the methodology proposed in this thesis. An application is first profiled to discover the kernels within the algorithm. Next, the algorithm is rewritten to extract the kernels into function calls. The algorithm is profiled again on a general-purpose processor to obtain the time and energy of a software-centric implementation of the algorithm. The topic of exploring different hardware and reconfigurable architectures for the kernels is the focus of Chapter 5.

The application of interest is an adaptive least mean square algorithm for multi-user detection of a CDMA system. The chip-rate is 3.2 MHz and 15 users and one pilot user are to be supported in the application. The algorithm written in C is profiled using

the source_profiler (described in Section 4.2.4) for one second of antenna data. Figure 4.5a shows the result of the total basic operations collected for one run of the algorithm. Figure 4.5b shows the percentage breakdown of each basic block.

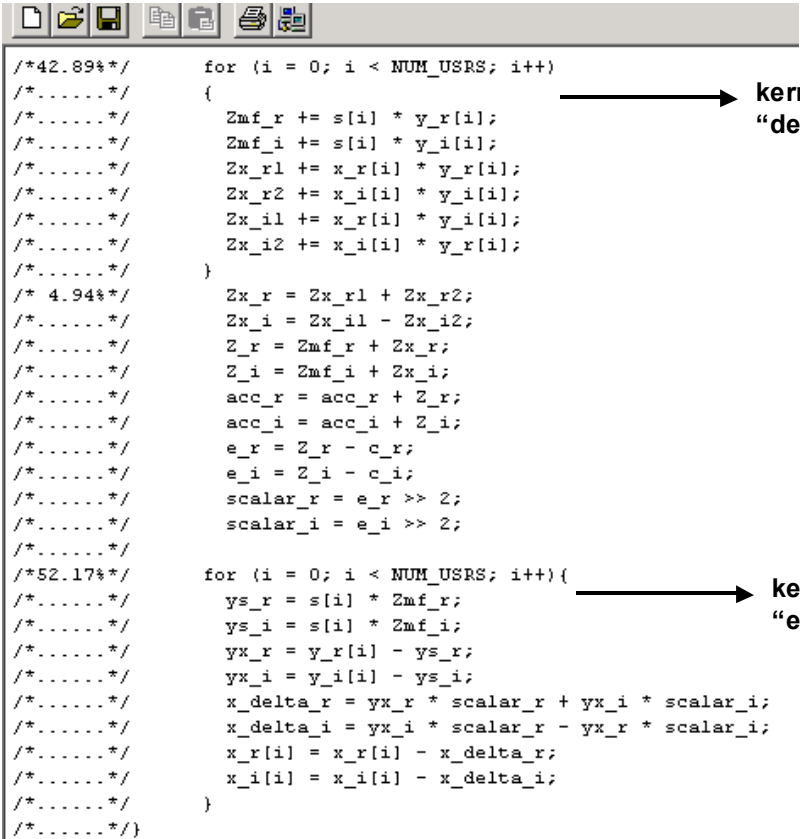
a)

```

... Profiling
...
MEM: 39,000,000 x 1 = 39,000,000 basic ops
MUL: 36,000,000 x 2 = 72,000,000 basic ops
ALU: 44,400,000 x 1 = 44,400,000 basic ops
CMP: 6,400,000 x 1 = 6,400,000 basic ops
TOTAL: 161,800,000 basic ops

```

b)



```

/*42.89%*/      for (i = 0; i < NUM_USRS; i++)
/*.....*/      {
/*.....*/          Zmf_r += s[i] * y_r[i];
/*.....*/          Zmf_i += s[i] * y_i[i];
/*.....*/          Zx_r1 += x_r[i] * y_r[i];
/*.....*/          Zx_r2 += x_i[i] * y_i[i];
/*.....*/          Zx_i1 += x_r[i] * y_i[i];
/*.....*/          Zx_i2 += x_i[i] * y_r[i];
/*.....*/      }
/* 4.94%*/      Zx_r = Zx_r1 + Zx_r2;
/*.....*/      Zx_i = Zx_i1 - Zx_i2;
/*.....*/      Z_r = Zmf_r + Zx_r;
/*.....*/      Z_i = Zmf_i + Zx_i;
/*.....*/      acc_r = acc_r + Z_r;
/*.....*/      acc_i = acc_i + Z_i;
/*.....*/      e_r = Z_r - c_r;
/*.....*/      e_i = Z_i - c_i;
/*.....*/      scalar_r = e_r >> 2;
/*.....*/      scalar_i = e_i >> 2;
/*.....*/
/*52.17%*/      for (i = 0; i < NUM_USRS; i++){
/*.....*/          ys_r = s[i] * Zmf_r;
/*.....*/          ys_i = s[i] * Zmf_i;
/*.....*/          yx_r = y_r[i] - ys_r;
/*.....*/          yx_i = y_i[i] - ys_i;
/*.....*/          x_delta_r = yx_r * scalar_r + yx_i * scalar_i;
/*.....*/          x_delta_i = yx_i * scalar_r - yx_r * scalar_i;
/*.....*/          x_r[i] = x_r[i] - x_delta_r;
/*.....*/          x_i[i] = x_i[i] - x_delta_i;
/*.....*/      }
/*.....*/

```

kernel function
"demodulate"

kernel function
"estimate_update"

Figure 4-5 Source Profiling Results for a MUD Algorithm

The first and the third basic blocks constitute the majority of the complexity. Therefore, they are extracted into function calls to denote that they are kernels. The algorithm is profiled again using an instruction set simulator for ARM8 to collect the software implementation cost of each kernel. The cycle and energy are both collected. The performance data is post-processed and imported into Excel. Figure 4.6 shows the results from this profiling.

Total cycles: 251.2 M cycles	
Total energy: 3,400 mW	
cycles	
	body (2.07%)
main (100 %)	demodulate (48.96%)
	estimate_update (48.97%)
energy	
	body (2.17%)
main (100 %)	demodulate (47.53%)
	estimate_update (50.51%)

Figure 4-6 Total Energy and Cycle for the MUD Algorithm and Kernel Percentage Breakdown

5: High-Level Simulation and Synthesis of Reconfigurable Satellite Co-processors

In most DSP and communication algorithms, the kernel data-flow computations are the most energy consuming part. Therefore, the overhead of the satellite architecture that implements the kernels has to be reduced as much as possible. In this chapter, a quick overview of the tools to assist in building optimized satellite co-processors is first given. A later section, section 5.1, is devoted to a detailed description of the satellite architecture model that is suitable for data-flow computations. In section 5.2 and 5.3, the simulation and synthesis tools to map the algorithm to the satellite architecture are given respectively.

In the proposed methodology, the description of algorithms is in a high-level language (C) so the algorithms can be targeted to different implementations. Targeting algorithms to a micro-processor has a standard mapping process—a compiler associated with the micro-processor is used to translate the high-level language to the object code, and an instruction-set simulator is usually used to collect performance and power data (as shown in Section 4.3.1). On the other hand, there is no straightforward way to rapidly translate algorithms to their register-transfer-level (RTL) representation of satellite co-processor implementations. In this chapter, two approaches, simulation-based approach and synthesis-based approach, are presented to close this algorithm-architecture gap. First of all, a simulation environment specified in the C++ language is described. This simulation approach advocates a manual algorithm-to-architecture process so the designers can model mappings of their own choice. The simulation environment gives rapid feedback of the performance and power of the manual mappings. However, the manual mapping process is often a laborious procedure.

Therefore, an automatic synthesis path with C as an input is created to speed up the design and optimization process. The synthesis tool is described in the second half of the chapter.

5.1 Data-driven Reconfigurable Satellite Co-processor Architecture

The heterogeneous reconfigurable architecture proposed in Section 2.2 consists of control-flow computations performed on the micro-processor and data-flow computations executed on the heterogeneous satellites.

The focus of this sub-section is on giving a detailed description of the satellite architecture. This description includes a generalized model for the reconfigurable data-driven satellites as well as the inter-satellite communications. To ease the implementation effort, the Pleiades architecture template fixes the interface mechanism between the micro-processor and the satellite as well as the communication scheme between each satellite. The interface between the micro-processor and the satellites takes on a common protocol between processors and ASICs—the communications from the processor to the satellites are synchronous and memory mapped; the communications from the satellites to the processor are interrupt driven. On the other hand, the inter-satellite communication is domain specific (data-driven) in order to obtain higher computation and energy efficiency.

In the thesis work of [Teu97], it is shown that for communication algorithms implemented on programmable processors, around 35-48% of the energy is spent on control related computations. These control-related instructions can be eliminated if a data-path mimicking the data-flow of the computation is constructed spatially. Another advantage of such a data-path is that more parallelism and pipelined computation (as required by the algorithm) can be supported in the architecture. There already exist many architectures that exhibit data-driven behavior because this architecture style

matches closely with the algorithm structure [Lip91]. Based on this analysis and the data-flow property of the kernel computations, the satellite architecture of the Pleiades template is data-driven. Figure 5.1 shows the basic features of the data-driven satellite architecture. The important thing to note is that the dedicated link, parallelism and pipelined computation are supported “within” a kernel computation. Timing sharing of links and satellites can happen between different kernels. As already shown in Section 2.2, the energy and performance efficiency of the satellite architecture is more than one order of magnitude better than the state-of-the-art programmable or reconfigurable processors.

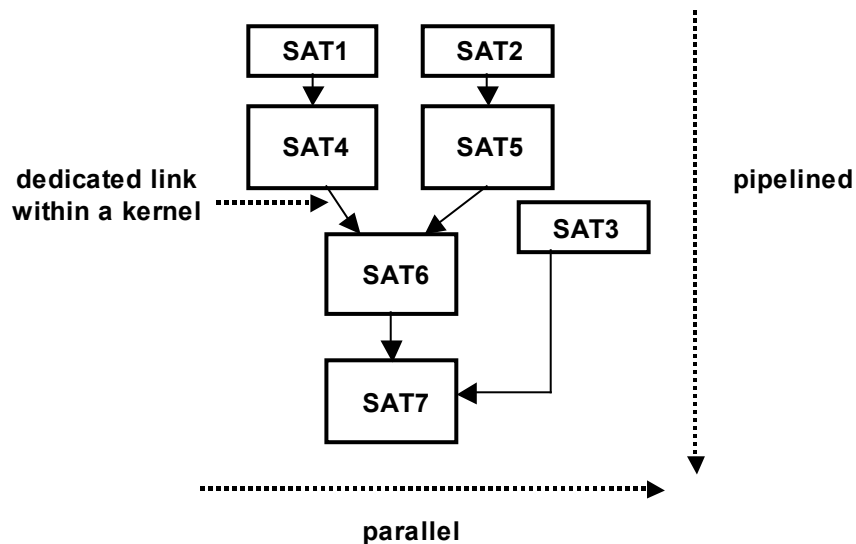


Figure 5-5-1 Features of the satellite architecture

In order to support the above architecture features for each kernel and keeping the satellites and inter-satellite connections re-usable for different kernels, each satellite is equipped with a data-driven protocol wrapper, and special data-driven routers are embedded in the inter-satellite connection. Section 5.1.1 addresses the data-drive protocol and Section 5.1.2 describes various data routing elements.

5.1.1 The Data-driven Protocol

In the realization of the Pleiades architecture template, the data-driven satellites are fine-grained to medium-grained according to the definition of [Kie98], which means that they are capable of executing two to five different types of instructions. Satellites with a large instruction set are avoided so the overhead of instruction decode and sequential control can be minimized. The functionality of the satellites is divided into three basic categories: source (data generation), computation (data computation) and memory (data storage).

One of the important ways to achieve low power implementation is to get rid of global control as much as possible. Wrappers are placed around all satellites in order to eliminate the global energy overhead of satellite computations.

First of all, the wrapper ensures that each satellite follow strict execution (i.e. operation starts only when all input data are ready) so there is no spurious computation. Secondly, to support adaptive computations without reconfiguration such as changing the vector length or number of taps for each satellite, a minimum-overhead mechanism is developed for passing the data from one satellite to another. This kind of distributed local control requires each satellite to be equipped with some amount of intelligence. The most fundamental intelligence is for the satellites to be aware of the basic data types frequently used in DSP computations. Three types of data structure are currently defined (scalar, vector and matrix) to be exchanged in-between satellites. The information on data structure of a data stream is indicated by tokens sent in parallel with the corresponding data [Ben99].

Each computation satellite needs to be configured for the data structures it consumes and produces. The source and memory satellites are in charge of generating tokens indicating the end of the data structure. Three examples of the computation satellites are shown. Figure 5.2c is the multiply in scalar mode (takes in two scalar data

and produces one scalar); Figure 5.2d is the multiply in scalar-vector mode; Figure 5.2e is a MAC (takes in two vectors and produces a scalar). The important thing to note is that only the “end” of a data structure is recognized by each satellite and automatic rate matching happens when a data structure of higher dimension is connected to one with smaller dimension. For example, when the embedded processor (Figure 5.2b) is connected to the MAC (Figure 5.2e), a scalar is produced at the output of the MAC after the first end-of-vector token. An end-of-vector token is generated by the MAC satellite after the end-of-matrix token from the embedded processor.

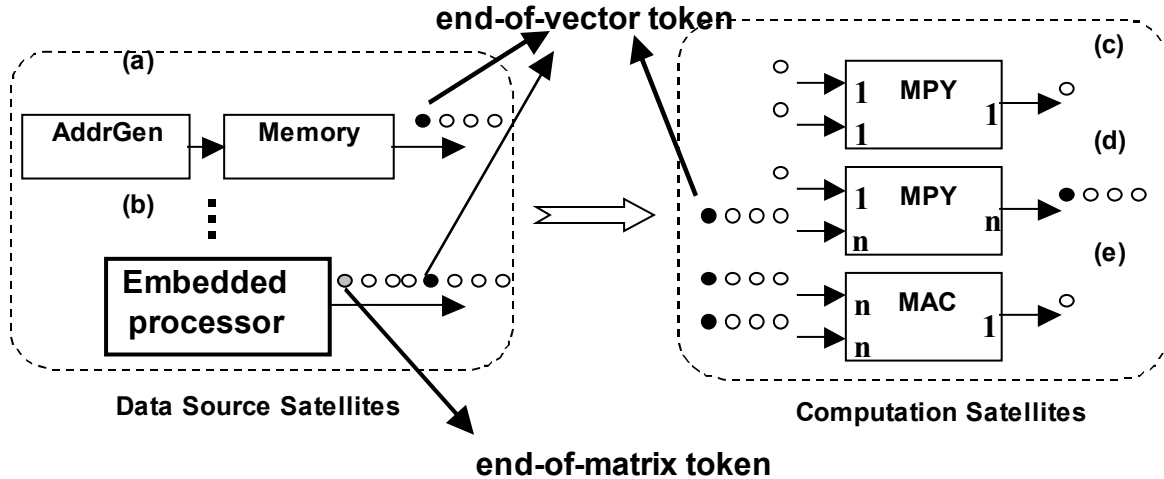


Figure 5-5-2 The dataflow satellite architecture (a) vector mode of AddrGen and Memory (b) matrix mode of the micro-processor (c) scalar mode of the Multiply satellite (d) scalar-vector mode of the multiply satellite (e) the MAC satellite

5.1.2 Inter-Satellite Dedicated Connections

Reconfigurable interconnection network is used to establish dedicated links between satellites (within a kernel execution) to preserve data correlation in signals, thus reducing energy consumption. In order to support dedicated links between satellites without reconfiguration overhead and global control, data steering elements

are embedded in the reconfigurable network. In general, data steering elements are divided into three categories: static (data goes in a fixed direction in between reconfiguration periods); statically scheduled (data goes in directions instructed by programs configured at reconfiguration times); dynamically determined (data is annotated with the direction).

Figure 5.3 shows two loops that have two different kinds of data steering elements in the satellite mappings.

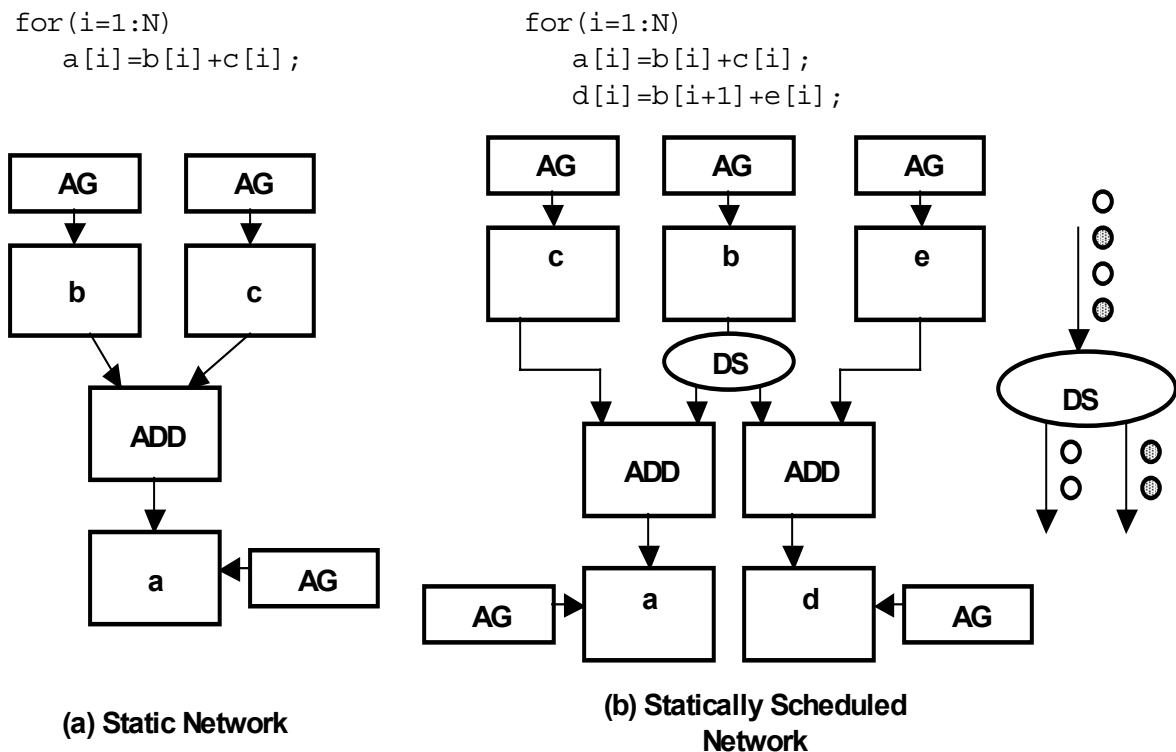


Figure 5-5-3 Different categories of data steering elements

Only the first two categories of data steering elements are supported by our realization of the architecture template because they are sufficient in supporting a

variety of data-flow intensive computational loops. Dynamic data steering imposes a large energy overhead in the data steering elements for the granularity of the computational satellites. Therefore, it is currently not supported in the proposed architecture template. Lack of time-multiplexing within a kernel is compensated for by using higher-level time multiplexing. Data correlation is preserved on each node-to-node connections, thus saving power.

5.1.3 Libraries of Basic Satellite Modules

To obtain reasonably accurate performance profiling for kernel mappings, it is important to have good models for the satellite modules. The data-flow driven computation is currently implemented using global asynchronous and local synchronous clocking circuitry. A general handshaking scheme has been developed and a library of satellites has been designed using the scheme and their power and timing data collected [Ben99]. Appendix A contains the list of satellites and their performance data. Among the library modules, address generators, input ports (with data from microprocessor) and FPGAs serve as sources and are in charge of generating end of data structure tokens. Implementation issues for low-power reconfigurable interconnection networks are addressed in detail in Chapter 7. The power and delay data of these implemented satellite modules serve as the basis of the simulation and synthesis tool.

5.2 High-level Simulation Strategies

The goal is to devise a simulator to verify the functionality of a mapped kernel with the rest of the system as well as to get a high level estimate of the cost in terms of latency and energy. Therefore, it is necessary for the simulator to be written in a high-level language so co-verification with the rest of the system is possible. In addition, there is a trend of moving the system modeling and simulation technique away from the

pure RTL model in order to increase simulation speed. In [Gue99], it is shown that a C/C++ model of DSP processors can achieve two to three order of magnitude speed improvement for core only simulation, and about one order of magnitude of improvement when bus and interrupt behaviors are modeled in C/C++. The improvement in simulation speed provides faster design turn-around time, and therefore allows the designers to explore more architecture options.

In this section, a simulation environment used to model the data-flow driven architecture is given. The simulator is written in a high level language (C++) to facilitate co-verification and exploration of hardware-software systems, which requires faster simulation speed for faster performance collection.

5.2.1 The C++ Intermediate Form (C++IF)

In this design methodology, an intermediate form is developed to represent a net-list of interconnected satellites. Since the intermediate form specifies kernel mappings. It is desirable for it to contain or generate performance information. Therefore, the intermediate form also has the ability to provide behavioral and cycle true simulation to collect energy and timing data. As will be seen in later chapters, this intermediate form is used for configuration code generation as well as reconfigurable interconnect architecture optimization.

The intermediate form is based on the concept of object-oriented specification. Since the computation dataflow is mapped to clusters of satellites, an object-oriented intermediate form with satellite modules (heterogeneous satellites) and queues (links between satellites) as base classes is created.

A mapped kernel is constructed by building a net-list using instances of satellite module and queue as shown in Figure 5.3. In the beginning part of this program example, 2 AGP (address generator processor) modules, 2 memory modules, one MAC

module and 5 queues are instantiated (declaration part). Next, the net-list of the kernel is specified by connecting each satellite to a queue. At the final stage, calling the *execute()* function starts the overall simulation for the kernel.

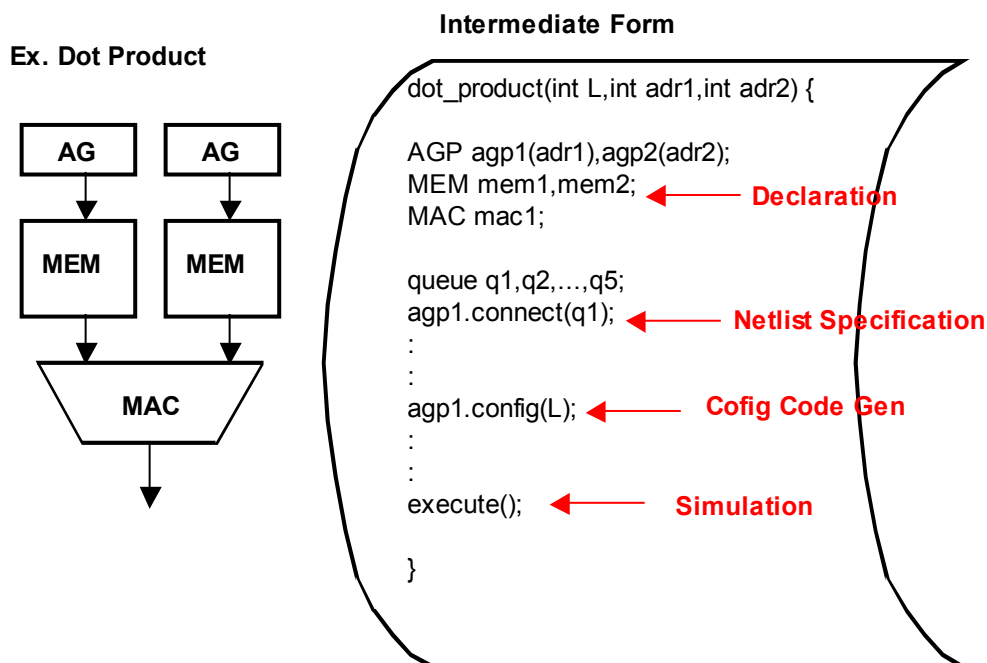


Figure 5-5-4 The C++ Intermediate Form

5.2.2 Simulation Based on the C++ Intermediate Form

As shown in Figure 5.3, an application specific simulator is automatically instantiated once a net-list is specified in the C++ Intermediate Form. The simulator is intended for functionality verification as well as collection of cycle and energy information. The data-driven architecture makes it easier to build a simulation environment since many dataflow simulators already exist for algorithm verification. Many dataflow models of computation exist and have its own associated simulation environment [for example, the domains in the Ptolemy simulator]. Among the well-

established dataflow models are Static Dataflow (SDF), Binary Dataflow (BDF), Dynamic Dataflow (DDF), and Process Networks, listed in order of run-time scheduling complexity from the least to the most time consuming. In order to simulate the dataflow, the simulation framework in general needs a global scheduler. The job of the scheduler is to take in the graph (representing the dependency of the dataflow design) and the consumption and production rate of each node to figure out the number of firings per node and the order of firing for each node. The only dataflow simulation that does not require a global scheduler is the Process Network model, where each node is replaced by a process and the overall scheduler is done by the explicit synchronization of each process. The SDF has an elegant formulation and sophisticated optimization techniques can be applied to it, however, the application domain of SDF is limited. However, all these models and their associated simulation techniques are limited to behavioral simulation since there is no concept of time in the computational model or in the simulator.

Based on the realization of the satellite architecture template, a simulation environment is developed to provide an application-specific simulator in order to facilitate rapid verification and performance feedback. It is decided to model the computation as process networks even though the data-path modules can be modeled as SDF or cyclo-static SDF (memory modules). The reason is that as the future satellite modules evolve into bigger sub-blocks or sub-systems, simple SDF or BDF will not be able to encapsulate the computation.

The simulator is built in the following manner: All satellite modules are modeled as concurrent processes, and queues between satellites modeled as synchronized objects. Based on the internal state of the module, certain number of inputs on each port is required. Once all inputs are present at the input ports and a module is fired, certain number of outputs is produced, and the internal state might change. Blocking reads

and blocking writes to the queue are used to enforce the buffer size between modules. Our simulator supports computations as concurrent processes with buffer size of one on each communication link. This model is similar to the actual architecture so timing and energy is easier to model as well. Figure 5.4 shows an example of such a network of processes being executed one time each. All four processes are started after the initialization. P0, P1 and P3 are stopped once they try to read from queue C, A and B respectively. P2 writing to queue A clears the blocking on process P1 and P1 writing to B and C clears the blocking on P0 and P3.

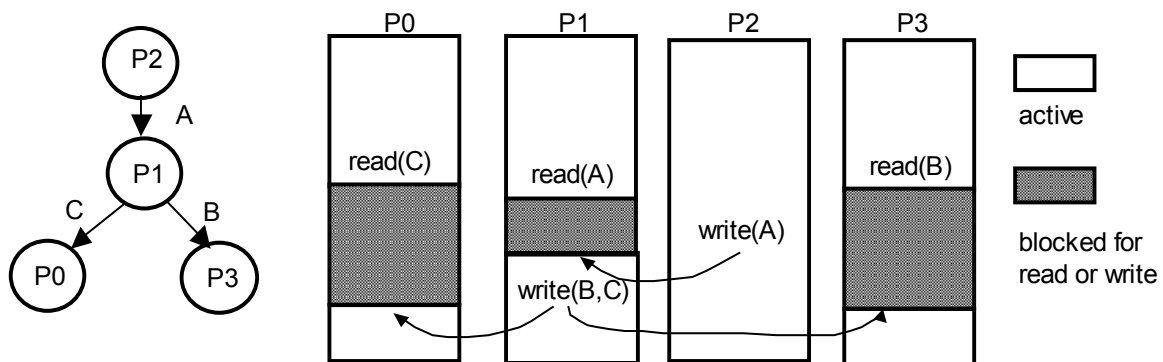


Figure 5-5-5 Modeling of the Data-driven Architecture Using the Process Network Model

To gather performance data, energy and time stamps are associated with tokens flowing through the modules and queues. The total energy for the kernel is the sum of all energy of modules and interconnects being accessed. Therefore, every time a module, a data-steering element or a queue is activated by an incoming token, a method is called to log the energy of the component into an overall database. The timing of the kernel is somewhat difficult because of pipeline effects.

Recall that each one of the queues between the modules has blocking reads and blocking writes. This means that when one port of a module tries to read from a queue and the data is not present, the module's process is blocked. The same rule applies when a module tried to write to a queue whose buffer size is full. We first introduce the psuedo-code to calculate the timing stamps: locally distributed algorithm.

First of all, recall (from Section 5.?) that the execution of a module only happens when all input ports have enough input data. Let $\text{Input}_{A,i}$ be the data on the i th input port of module A ; $T(d,n)$ be the time stamp on the n th occurrence of data d ; N_A is the number of input ports belonging to A ; $D_{A,j}$ be the latency through module A to produce output j once all data is ready. The time stamp at the output j of module A is calculated as follows:

$$T(\text{output}_{A,j}, n) = \max(\max(\{T(\text{input}_{A,i}, n) \mid i \in \text{inports of } A\}) + D_{A,j}, T(\text{output}_{A,j}, n-1))$$

Once all data at the inputs are ready, the maximum of the input time stamps is added to the latency to produce a temporary time stamp of the output. This time stamp is compared with the actual consumption time stamp of the previous output, and the larger of the two is selected. For the purpose of calculating time stamps, a queue with delay (a connection delay) is treated like an one-input-one-output module.

The above time stamp calculation algorithm makes sure that pipeline effects are modeled correctly. Figure 1-6 shows an example of a pipelined data-path with one pipeline register on each connection. For $P3$, the time stamp on the output is always the time stamp of each input plus the latency of $P3$. However, for $P0$ and $P2$, since pipeline stalls occur, the time stamp of the output depends on when the previous output is consumed.

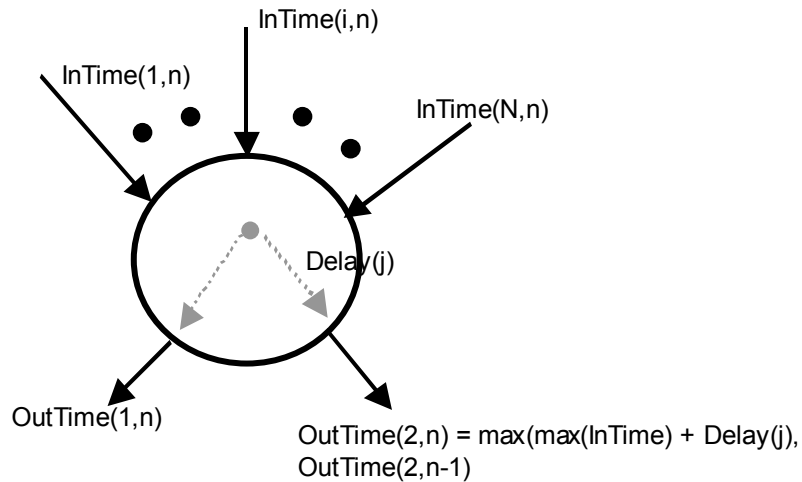


Figure 5-5-6 Computation of Finish Time of the Satellite

Currently, the simulator is implemented in the C++ language and the Solaris thread library [9] (other common thread libraries can be switched in easily). Common satellite processors (such as MAC and multiply processor, ALU processor, memory, and address generator) and data-steering-interconnect modules have been incorporated in our satellite module library.

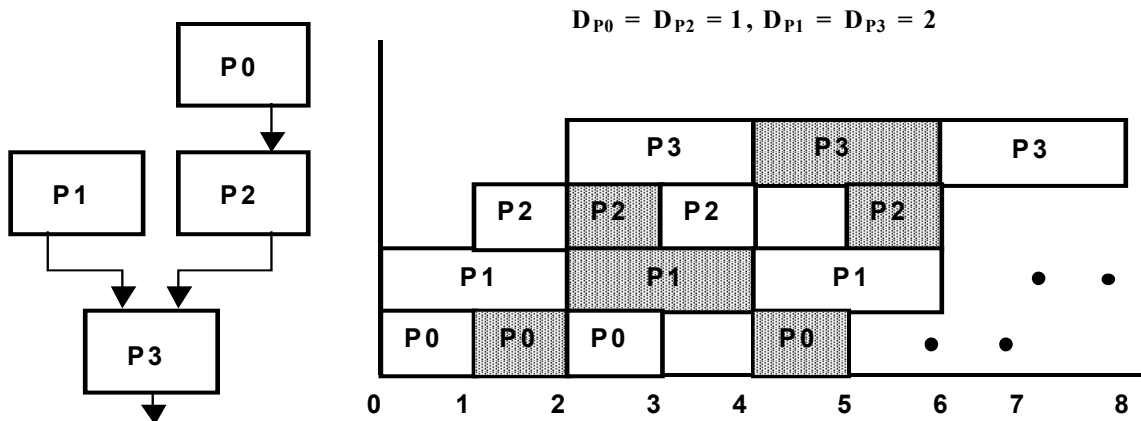


Figure 5-5-7 Modeling of Pipeline Timing

5.2.3 High-level Simulation Case Study

In this section, a kernel mapping process is used to demonstrate the effectiveness of the simulator. The simulation speed of the tool is first quantified and the performance data generated by the tools is presented.

The design example is the least mean square (LMS) multi-user detection algorithm, which is profiled at the end of Chapter 4. This algorithm is composed of two adjacent loops (one that demodulate the symbol, the other updates the energy matrix of the user being demodulated), and some interim scalar computations in-between the loops. In this case study, the kernels extracted in Section 4.4.1 are used and the following five mappings are this investigated:

Mapping 1: Entire algorithm in software

Mapping 2: The first loop, demodulate, running on the satellites, the rest in the software

Mapping 3: The second loop, `estimate_update`, running on the satellites, the rest in the software

Mapping 4: Both of the loops running on the satellites, with reconfiguration in between.

Mapping 5: Both of the loops running on the satellites, with no reconfiguration in between.

Simulation Speed Comparison

The original algorithm is specified in C and the energy and latency of mapping 1 is first collected using the instruction set profiling tool described in Section 4.3 (for the

embedded ARM8 processor). As shown in Figure 4.6, the energy and latency of the demodulator and estimate_update kernels are also the output of the profiler. To obtain the latency of each one of the kernels, two simulation paths are taken. In the first path, the architecture simulation is done in the Synopsys VCS simulator [Syn01]. Kernels are mapped, and the VHDL blocks of the satellites are connected according to the mappings. In the second path, the satellites in C++IF are connected to simulate the kernels. The speed of the two simulators is listed in Table 5.1. The data is collected by feeding one second of antenna input to the mapped kernels.

Table 5-5-1 Speed Comparison between Synopsys VCS and C++IF-Based Simulator

Kernels	Simulation Time for Synopsys VCS (sec)	Simulation Time for C++IF Simulator (sec)
demodulate	18.02	777.92
estimate	16.70	689.38

As shown in Table 5-1, the simulation speed of the VHDL simulator is on the order of 40 times slower than the C++IF based simulator. The slow simulation speed of the VHDL simulator prevents it from being used at the design exploration stage. Another drawback of the VHDL simulator is that the designers cannot verify the correctness of the kernel mappings in the same simulation environment unless co-simulation or manual vector transfer is used. On the other hand, the C++IF based simulation offers not only fast simulation speed but also functional verifications of systems consisting of hardware and software. Therefore, for the LMS design example, the C++IF based simulator is used to explore different mappings.

Results of Exploration

The first loop, demodulate, is extracted to a function call, mapped to the C++ intermediate form, and simulated with the rest of the application for functional verification. The energy and latency of the newly mapped loop is collected from the simulation as well. We perform similar tasks for the rest of the three mappings. Figure 5-7 shows the energy and latency data for each one of the 5 mappings (all satellites and the microprocessor are running at 1V). As shown in the figure, significant improvements in both energy and time are achieved as we move more computations to the satellites (as predicted). One thing to observe (difference between mapping 4 and mapping 5) is that it is important to try to pack in as much computation into the satellite as possible since re-configuration is very costly with current implementation (through the RISC micro-processor). Later in this thesis, we will address the issue of reconfiguration and optimizations done in software. In the future, it is also important to provide better hardware architecture for reconfiguration.

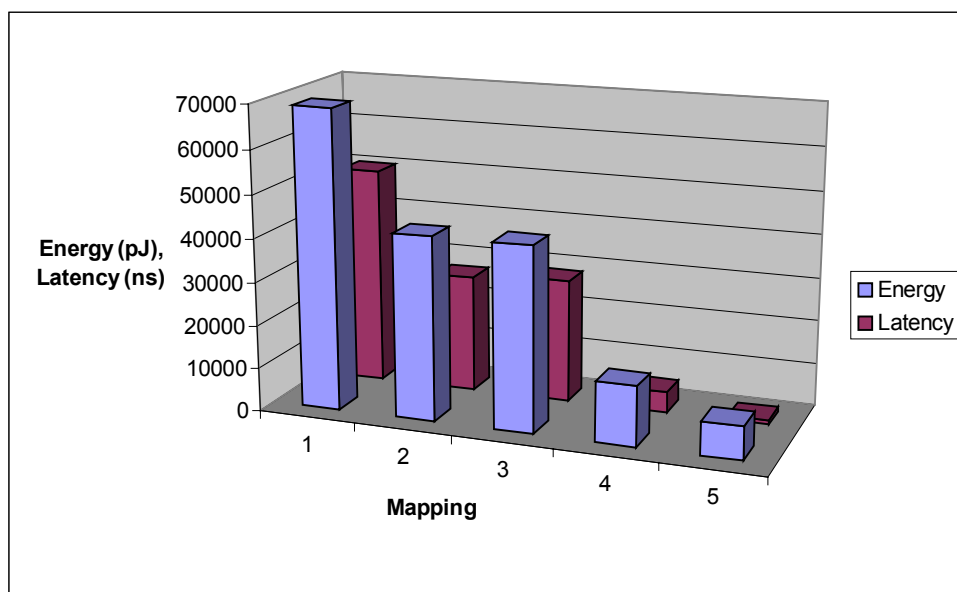


Figure 5-5-8 Energy and latency for different HW-SW mappings for the LMS MUD algorithm

5.3 High-Level Synthesis Strategies

While the simulation environment provides an efficient way for designers who can manually map kernels to obtain performance profiling information, mapping algorithms to the dataflow architecture is time consuming, and requires a detailed understanding of the underlying architecture. It is therefore advantageous to provide a synthesis route from algorithms to the data-driven architecture. The synthesis tool can free the designers from the tedious mapping process, thus speed up the design process even more. This section contains the description of a synthesis environment to compile a kernel described in a subset of C to satellites in the C++ intermediate form.

5.3.1 Basic Synthesis Flow

Many efforts from the high-level synthesis research have been devoted to translating algorithm descriptions to efficient hardware implementations. Many high-level synthesis tools start with application specific languages or hardware description languages. Recently, many high-level synthesis projects concentrate on synthesizing hardware from general-purpose high-level languages (such as C/C++ or Matlab) [She01][Cal98][Wan98], because these languages are used by a large number of system designers to implement algorithm. Taking a similar approach, C is used as the starting point in our methodology. Since the C language is originally intended for software programming, not all programs specified in C can be mapped into hardware. The subset of C that is supported in our synthesis environment requires the kernel to reside in nested loops or adjacent nested loops. In addition, no function calls are allowed in the loop body except for function calls that are specified to the synthesis tool as a basic computation satellite.

Figure 5.7 captures the steps of the synthesis path. The synthesis environment uses the same front end as the SUIF compilation environment (first introduced in Chapter 4). The input to the entire environment is an algorithm specified in C. The

path on the left is the software generation path in the SUIF. After the source optimization stage, the flow is forked off to a hardware synthesis path. The input to the hardware synthesis path is the abstract syntax tree, control dataflow graph (CDG) is the intermediate representation, and the output is a mapped kernel specified in the C++ Intermediate Form and the latency and energy consumption of the mapping. The synthesis strategy is to provide a straightforward synthesis path from an algorithm representation to a mapped kernel by using direct mapping. The cost of the mapping is fed back to the designer who uses the synthesis tool, and transformations are done at the algorithmic level. The most appropriate place to carry out these transformations is in SUIF or manually by the designer. The remainder of this section is devoted to describe each step in the hardware synthesis path. A design example using the synthesis tool is presented at the end of this section.

5.3.1.1 From SUIF to Control Dataflow Graph

In the first two stages of the synthesis path, the algorithm is read in by the SUIF environment, analyzed and represented as the Abstract Syntax Tree (AST). The Abstract Syntax Tree representation favors a compiler backend that generates RISC-like software since the dataflow of each variable is absent in the representation. To synthesize AST to hardware, it is important to explicitly represent the dataflow (i.e. each operation as a node and data feeding the operation as edges into the node). In the synthesis path, three steps, SSA transformation, loop delay insertion and array dependency analysis, are taken to detect the data and control dependencies of scalar and array variables in the algorithm.

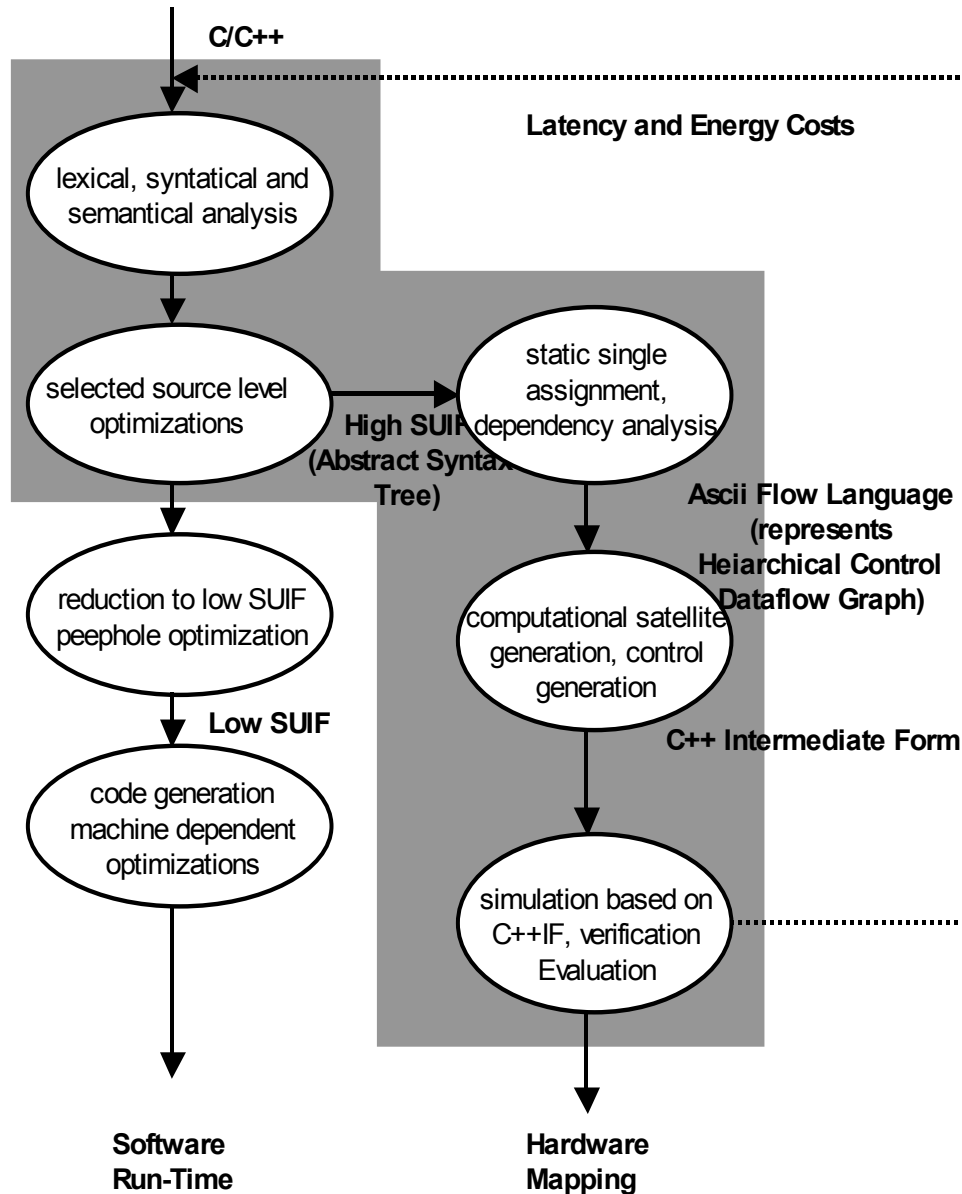


Figure 5-9 Steps in the software and hardware synthesis flow

First, static single assignment (SSA) transformation [AyH00] and delay insertion are applied to the AST representation of the algorithm. The necessity of these two steps is illustrated in the following example. Figure 5.10a shows a C code example. The value of the variable `a` is changed in the first assignment of the loop body and all subsequent usage of `a` should refer to the new value; variable `a` is changed again in the second

assignment and the next loop iteration should use the new value for a. Similarly, variable b is changed in the third assignment, all subsequent references to b (in the next iteration of the loop) should refer to the value.

Figure 5.10b shows the code of its loop body for a RISC-like programmable processor. The mapping is almost one-to-one since the dependency between the production and consumption of each symbolic variable is taken care of by using the right register. However, to synthesize correct hardware structure, it is important to realize that a new data is generated for each assignment statement. Figure 5.10c shows the SSA form of the loop body of the same code. A fast algorithm to perform SSA transformation is described in [AyH00]. Figure 5.10d shows the control dataflow graph of the loop body. It is easy to see that the CDG is generated by replacing all operators as nodes and all variables as data edges.

The SSA step takes care of transforming scalar variables in C to a representation for hardware mapping. The following transformations are done on array variables, and they are based on some assumptions on how array variables are mapped to the architecture. Each array is assumed to be mapped to a memory with its own address generator. Therefore, each array access is represented as either a read or write node. Control edges are then added to the CDG to preserve the read-after-write (RAW), write-after-write (WAW), write-after-read (WAR) dependencies for a single array.

Delay insertion step follows the SSA step and array dependency analysis step to take care of variable dependencies across the loop boundary. A delay node is inserted between the last data edge and the first data edge of all scalar variables that are on both sides of the assignment statement in the loop body, or the last array write and first array read nodes. Figure 5.10e is the CDG after delay nodes are inserted to the graph in 5.10d.

A dataflow intermediate representation with control information is needed, and the ASCII Flow Language (AFL) used by the High-level Synthesis Tool Hyper [Chu89] is chosen. The CDG used in the synthesis supports control paths uses hierarchical CDGs from the Hyper synthesis environment. The control types supported in the C language are perfectly nested “for” loops and multiplexer-like “if and else” statements.

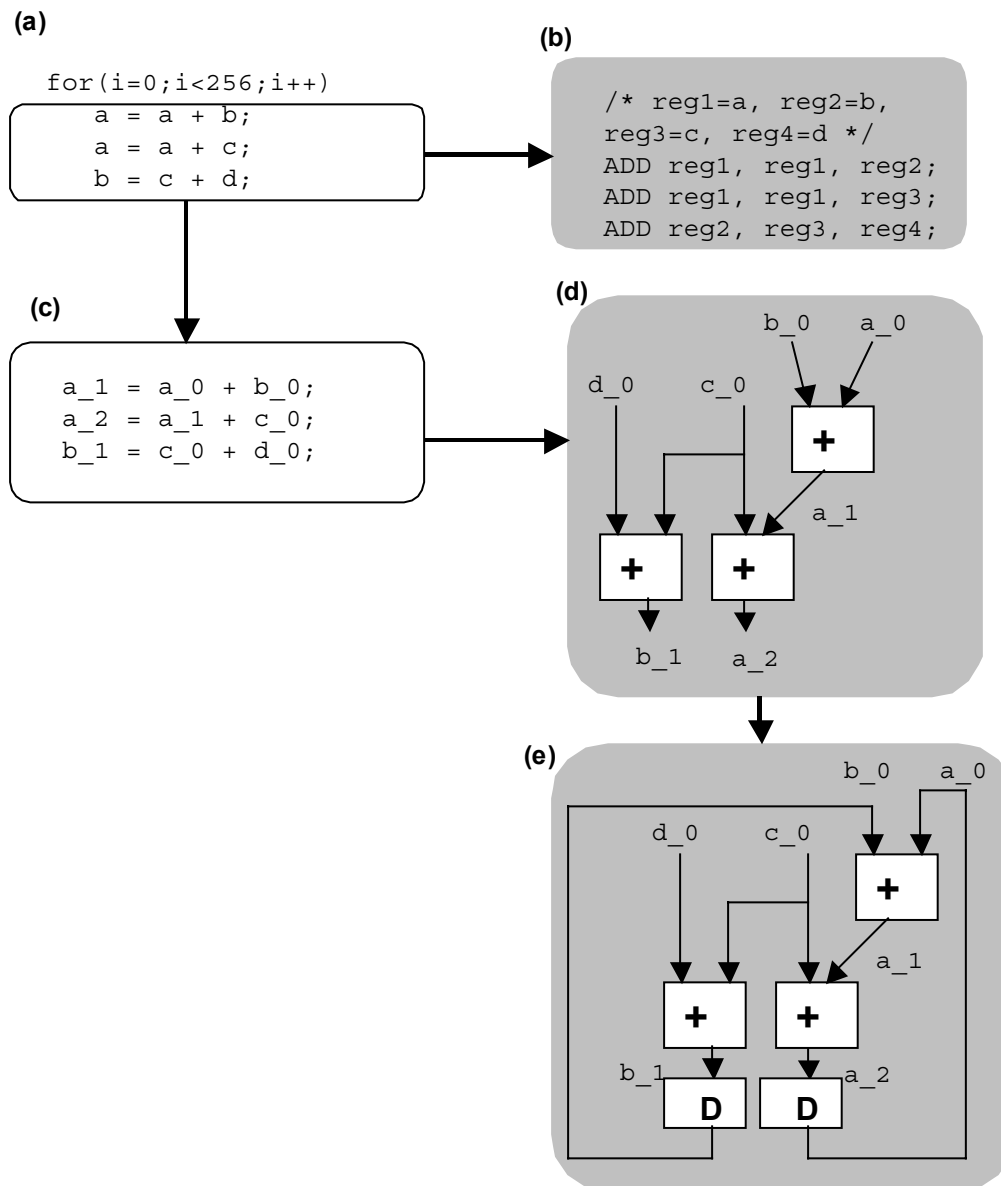


Figure 5-10 (a) the original c code (b) software generation for a RISC-like processor (c) the loop body after static single assignment (SSA) transformation (d) control data-flow graph for the loop body after SSA transformation (e) control data-flow graph after delay node insertion

5.3.1.2 From Control Dataflow Graph to Dataflow-driven Architecture

Once the algorithm is translated into a hierarchical control-dataflow graph, the mapping to the dataflow-driven architecture is straightforward. The direct mapping with rate optimal [Rei76] scheduling is employed. In direct mapping, there is no hardware multiplexing for the computational nodes. A computational node in a CDG is assigned to a satellite. Rate optimal scheduling usually incurs large area overhead. However, the area penalty is avoided in our system since higher utilization of each hardware satellite module is achieved at the system level, where different kernels are reconfigured from the same set of satellites.

The synthesis algorithm is broken down into three distinct passes: The first pass deals with computational nodes that can be mapped into computational satellites; the second pass translates control edges and loop nodes into Address Generation satellites and data steering elements and annotate read/write nodes of the same array with the same memory satellite; and a final pass generates the net-list. These first two parts annotate the control-dataflow graph with the satellite modules and data steering elements, and a final graph traversal translates the graph into a net-list representing the dataflow-driven architecture (in C++ intermediate form).

Mapping Computational Nodes

For the mapping of computational nodes, a library of satellite modules is provided. Some of the satellites map to several chained operation nodes since such implementation gives better area, timing and energy performance. For example, a MAC satellite is mapped to a multiply and an accumulation node in the control-data flow

graph. The template matching algorithm from Hyper [Gue97] is used to cover the graph with library modules so that the optimal overall latency is achieved.

Performance and Energy Estimates

Estimates of the latency and energy of the mapping is obtained at this point. The latency (described in Eq. 5.1) is characterized by the iteration period bound of the loop and the number of iterations of the loop. The restrictions on the “for” loop indices are that the start and end has to be an affine function of the enclosing loop indices, and the increment has to be integer. Therefore, loop iteration count for the perfectly nested loop depends on all the loop indices, and cannot be easily (or very efficiently) calculated in the synthesis tool. A function calculating the loop iteration based on the loop indices is also output by the estimator. A software package, MathCad, which is capable of evaluating functions symbolically uses the loop iteration function, Eq. 5.1 and Eq.5.2 below to get the latency and energy estimates.

$$Latency = LongestPath + (loop_iter - 1) \times IPB \quad (Eq. 5.1)$$

$$Energy = (loop_iter - 1) \times \left(\sum_{nodes} energy(satellites) + \sum_{dataedge} energy(int erconnect) \right) \quad (Eq. 5.2)$$

Control Information Generation

The control generation of the synthesis process is different from other synthesis environments that target other AISC-like architectures. As described in 5.1.1, the control of the dataflow-driven architecture is taken care of by the address generation nodes and the data steering elements. Currently, an address generation and a data steering element are associated with each array since it is assumed that all the data in an array resides in one memory satellite. Figure 5-11 shows the three types of control

elements to be generated. Figure 5.11a is the address generator; Figure 5.11b is the data alternator and Figure 5.11c shows the data merger.

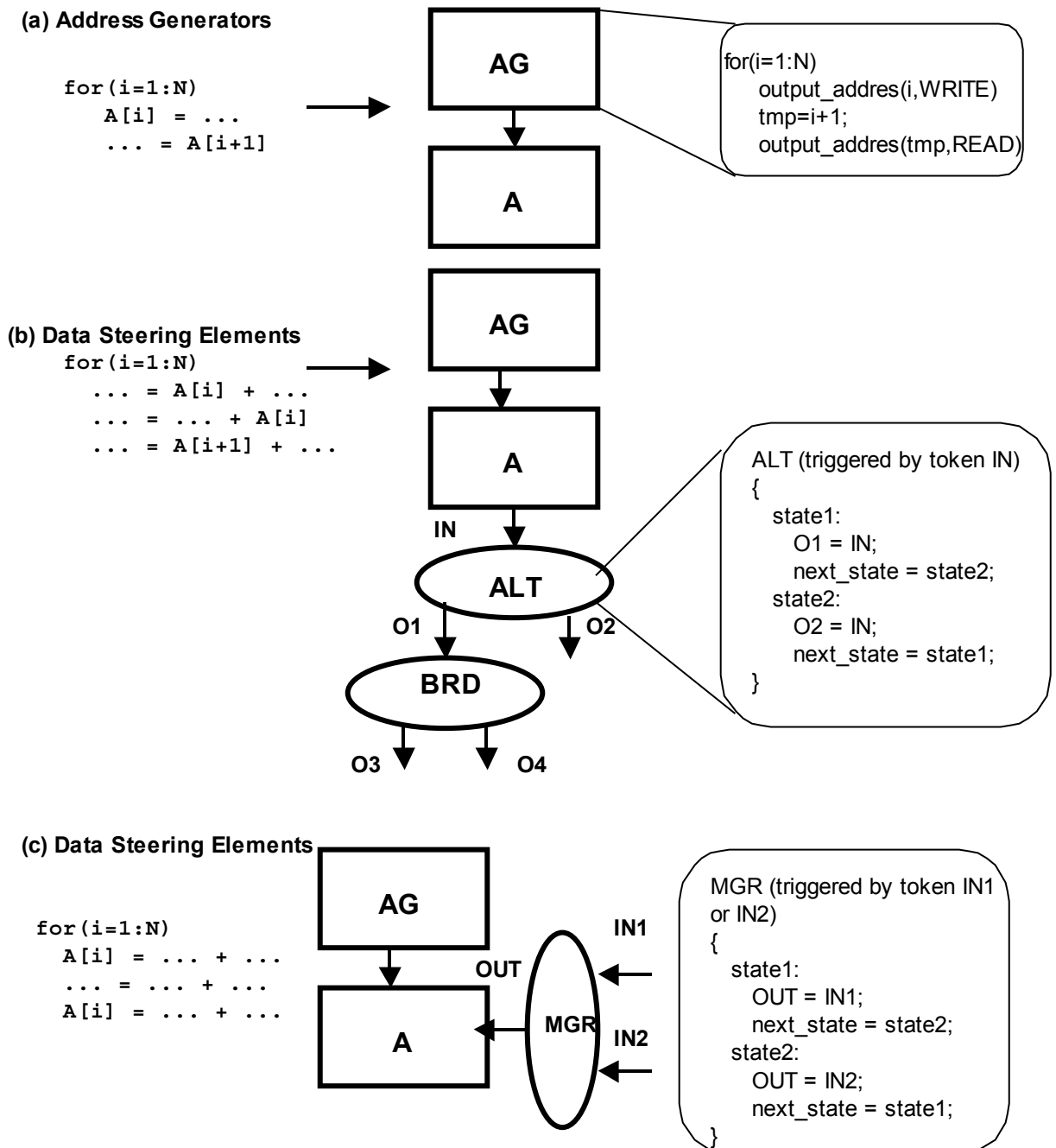


Figure 5-11 Examples of Mapping Array Accesses to Address Generators and Data Steering Elements

The control elements are generated from the CDG in a single traversal of the graph. During the traversal, two data structures are maintained to keep track of the address generator and data steering elements for each array. The information contained in the address generator is the “for” loop enclosing the array, the array indices and the read/write mode for each access. The information recorded for the data steering element is all the read/write access and the data edge where the data goes to or come from. The post-processing pass follows the graph traversal to generate programs for each address generator and data steering nodes. As shown in Figure 5.11a, the address generator program can be recreated from the information recorded. There are three kinds of statically scheduled data steering elements: the alternator, the broadcaster and the merger. The alternator and the broadcaster are used at the output of a memory bank (when there are multiple reads from an array). The merger is used at the input of the memory bank (to select multiple data needed to be written in). As shown in Figure 5.11b and Figure 5.12c, two data steering finite state machine are created for each array: one for output, the other for the input.

5.3.1.3 Useful Transformations

In our synthesis environment, many transformations are used at the algorithm level. The transformations used are from two categories: one set of transformations is used to enhance the final performance of the kernel mapping (dead-code elimination, constant propagation and loop invariant detection and extraction); the other set is provided to help explore different final kernel mappings (such as unrolling to expose parallelism at the implementation level). The first set of transformations is always enabled since the final implementation can only improve after the transformations. The second set of transformation has to be enabled by the designer and sometimes manually transformed (at the algorithm level) to obtain a different kernel mapping. The SUIF transformation library is used for both categories.

5.4 Case Study for the Synthesis Flow

In this section, two synthesis examples are shown to illustrate the design exploration space that the fast synthesis environment enables. The first example shows how the combination of algorithmic transformation and the synthesis tool can be used to evaluate the effects on parallelism. The second example shows how different algorithms can be rapidly mapped to our architecture and compared with other algorithm and architecture.

5.4.1 Evaluation of Data Parallelism

In this section, we show an example of exploring different satellite mappings for a kernel. The same kernel used (LMS MUD) in section 5.2.3 is investigated again. The starting point is the algorithm specified in C. The synthesis tool is used to map this kernel entirely to the satellites. The energy, latency and area of the first mapping is shown as the left bars in the three charts shown in Figure 5.12 (the cost is for demodulating one symbol for all users in the system). The loops are then unrolled once at the algorithm level (in SUIF). Since there are no dependencies between the inter-loop array accesses, the computations can be parallelized. The costs of the new mapping are shown as the right bars in the same three charts in Figure 5.13. As shown in the cost charts, the parallelism only improves the latency of the implementation. The reason is that providing parallelism increases the number of satellites as well as interconnection. Many DSP and communications, such as the LMS MUD studied here, has a real time constraint. As long as the real-time constraint (the number of symbols per second) is met, there is no need to exploit more data parallelism to increase implementation latency.

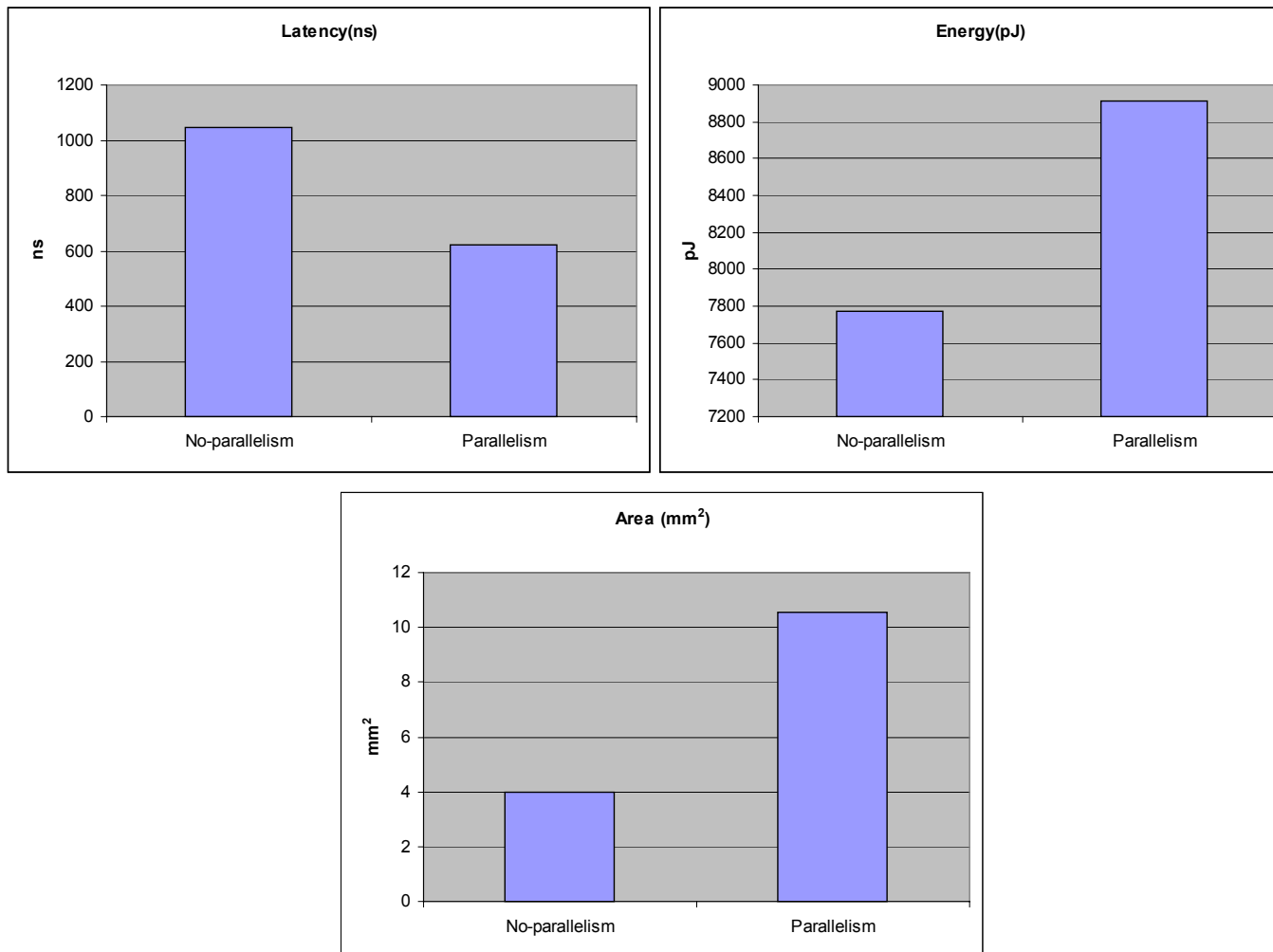


Figure 5-12 Latency, energy and area comparisons for kernel mappings supporting different parallelism

5.4.2 Architecture and Algorithm Selection

This design example is aimed to synthesize three different multi-user detection algorithms to different architectures (programmable architecture, reconfiguration data-flow driven architecture activated in this thesis and ASIC architecture). The real-time constraint for these algorithms is that it has to support demodulation of 0.8M symbols

for 28 users in one sec. The three algorithms are matched filter, MMSE trained detector, and MMSE blind detector (with increasing complexity and increasing capability to deal with interference in the communication channel). The ideal system design will be to have an implementation that can be reconfigured between these algorithms based on the channel condition. The estimates for the TMS320C54 and ASIC implementation is taken from [Zha99]. The starting point for the Pleiades implementation is the C description of these three algorithms (in Appendix II), and the mapping is synthesized from the description. Table 5.2 and Table 5.3 show the power and area costs of each algorithm on the architecture. IMS320C54 and Pleiades, once the architecture is designed for the most complex algorithm (MMSE blind), the same architecture can be re-used for all other algorithms (shown in the area table below). Three separate ASIC implementations have to be designed to support reconfigurability, so the area is the sum of all three ASICs.

Table 5-2 Power/energy consumption (mW/mJ) of different algorithms on various architectures

	Matched Filter	MMSE Trained	MMSE Blind
TMS320C54	68.00	152.00	303.00
Pleiades	4.46	12.96	18.10
ASIC	0.40	1.60	3.10

Table 5-3 Area (mm²) of different algorithms on various architectures

	Matched Filter	MMSE Trained	MMSE Blind
TMS320C54	115.00/530.00	253.00/530.00	530.00/530.00
Pleiades	1.43/4.00	3.10/4.00	4.00/4.00
ASIC	0.60/5.60	2.00/5.60	3.00/5.60

Figure 5.13 to Figure 5.15 show the mapping of each the algorithms.

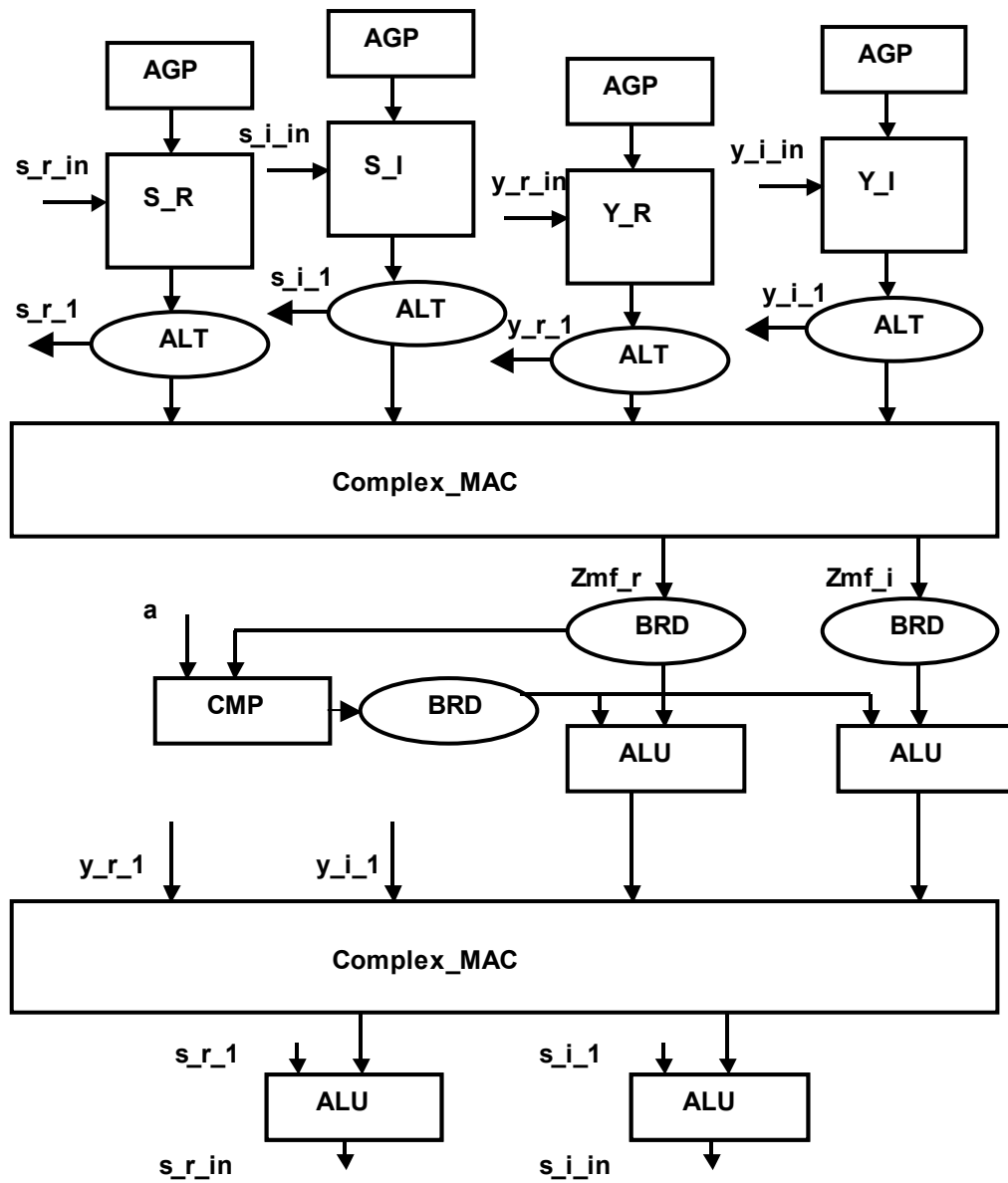


Figure 5-13 Mapping of the MMSE trained detector

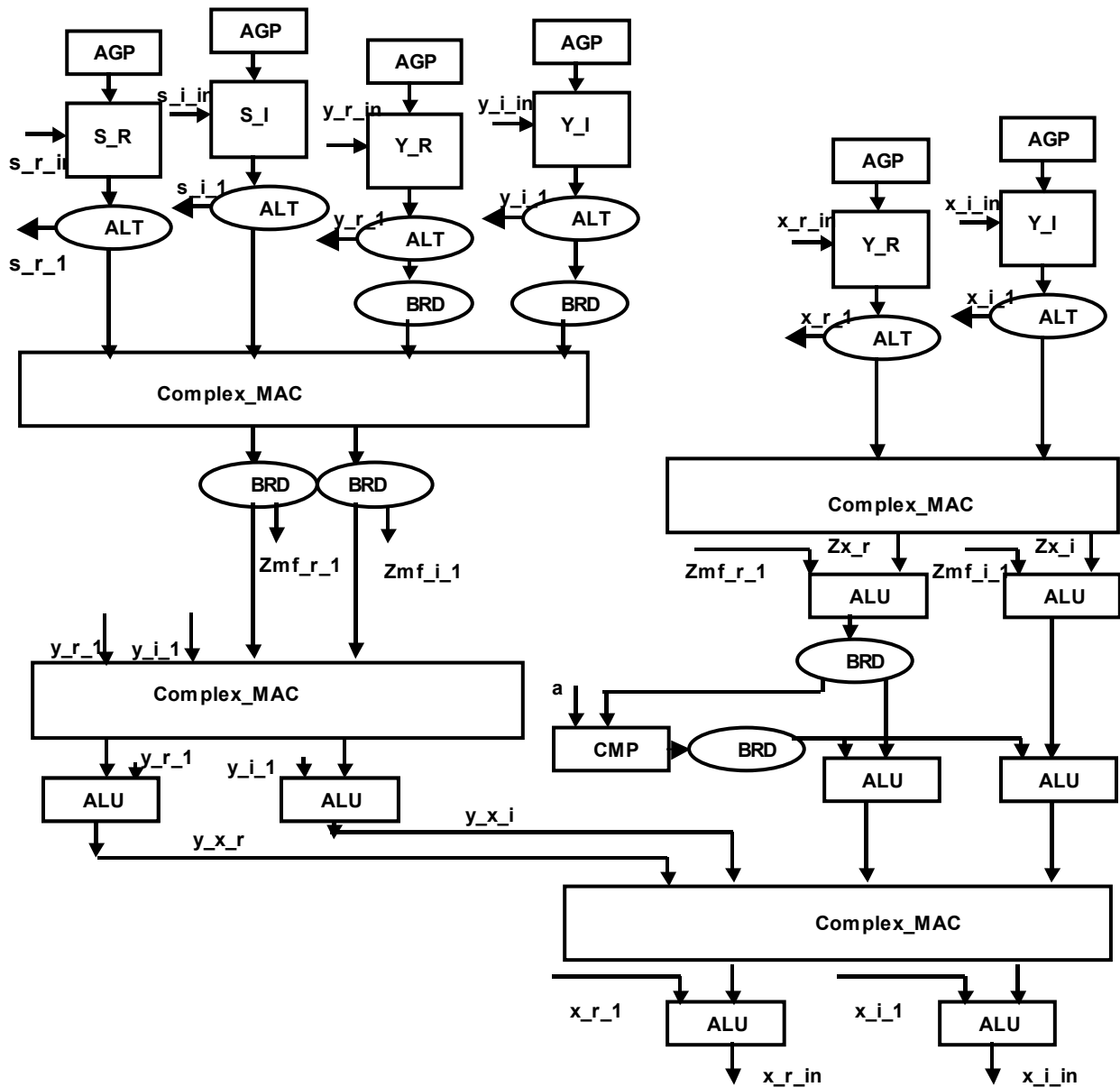


Figure 5-14 Mapping of the MMSE Blind detector

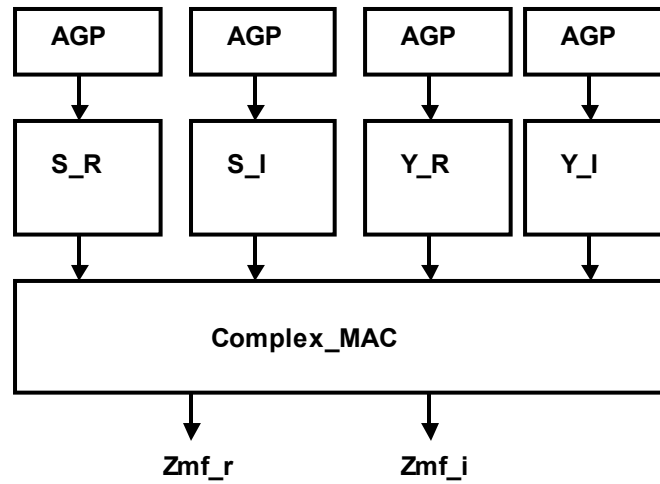


Figure 5-15 Mapping of the matched filter detector

In the next chapter, the kernel mappings synthesized and simulated using the tools described in this chapter are used to instantiate a final architecture that includes a control processor and satellite processors.

6 Hardware/Software Partitioning and Architecture Instantiation

The co-processors synthesis and simulation environment presented in Chapter 5 provides a set of hardware mappings for each kernel. This chapter addresses how to choose between different software and hardware mappings for each kernel, so that the overall system performance can be optimized.

In the first part of the chapter, the control flow of computation used by the (Pleiades) architecture advocated in this thesis is briefly presented again. Based on this generalized architecture style, a formulation for partitioning an application across hardware, software, and reconfigware (deciding a hardware mapping or software mode for each kernel) to minimizing energy while subjecting to timing constraints is presented. Solutions to solve the partitioning problem are proposed.

The partitioning step decides the architecture (satellite types and satellite numbers) for kernels, but the architecture for the entire application is not yet defined, because the resource of each kernel has to be time-shared with other kernels. In the later sections of this chapter, an architecture instantiation stage is described. The architecture instantiation stage follows the partitioning stage to decide on an architecture instance for the entire application.

6.1 The State-of-the-Art Hardware-Software Partitioning Approaches

Performing hardware/software partitioning subject to area and timing constraints is a design methodology studied by many researchers to increase the performance of a system [VGG94][Kal95b][HeE97]. However, minimizing the power consumption of a system through hardware/software partitioning has not been addressed until recently

[Dav97] [Hen99]. A task-based partitioning/scheduling algorithm to minimize the overall power consumption is introduced in [Dav97], but the scheduling of tasks in the methodology assumes different models of computation than the one required by the Pleiades architecture (for example, no time-sharing of hardware is allowed in [Dav97]). In [Hen99b], the solution is obtained through a greedy partitioning algorithm. However, sometimes it is worthwhile to consider finding the global optimal for some application specific instances. Therefore, none of these works can be used directly to solve our partitioning problem. In the following sections, a formulation of the partitioning problem for the Pleiades architecture is first given. To help solve the problem optimally in certain cases, some valid assumptions about the problem based on the control-flow of architecture and real-life experience are described.

6.1.1 Review of the Computation Control-Flow

To understand the hardware/software partitioning algorithm needed by the design methodology, it is necessary to have a quick review of the control flow of the Pleiades system. As shown in Figure 6.1, the main computation is done on a core processor while kernels can be potentially spawned off to accelerators (satellites). The entire computation flow is called each time a set of periodic inputs comes in. The timing and energy estimates of the processor core are obtained using methods discussed in Chapter 4, and the estimates of the kernel mappings are obtained by the tools outlined in Chapter 5.

6.1.2 Formulation of the General Partitioning Problem

Based on control flow of the system described in Section 6.1.1, the formulation of the partitioning problem is defined as follows:

Architecture Assumption: The architecture includes a voltage scalable core processor with satellites running at a fixed voltage. However, the voltage of the core can be set only at the initialization time, not at run-time.

Goal: Minimize total energy while meeting timing constraints.

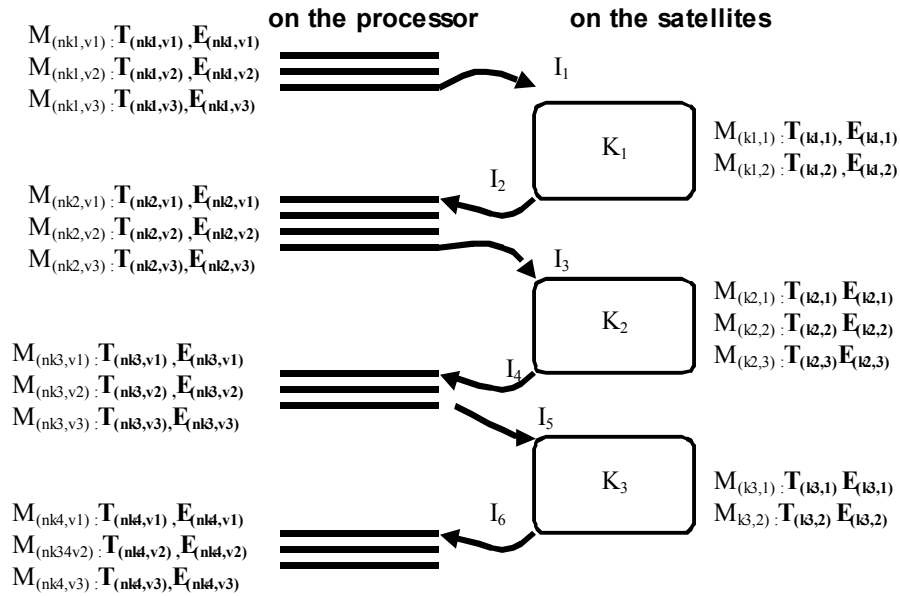


Figure 6-6-1 Pictorial Representation of the Partitioning Problem

The problem is formulated as an 0-1 integer linear problem (ILP) and the variables are defined below (the variables are also shown in Figure 6-1)

Variable descriptions:

K_i	where $i = 1..n$ represents the i th kernel in the application, there are n kernels
$T_{(ki,j)}$	represents the performance for the j th mapping for the i th kernel (HW or SW)
$E_{(ki,j)}$	same as $T_{(ki,j)}$, but for energy
$T_{(nki,vj)}$	represents the performance for the i th non-kernel computation, for each voltage level, v_j , where $j=1..m$
$E_{(nki,vj)}$	same as T_{v_i} , but for energy

Decision variables (0-1 variables):

$X_{(ki,j)}$	decision variable for each kernel mapping (1 if chosen, 0 if not)
Y_{v_i}	decision variable for each voltage level (1 if chosen, 0 if not)

Formulation based on the goal and variables:

$$\text{Min } \sum_{\forall i} \sum_{\forall j} X_{(ki,j)} E_{(ki,j)} + \sum_{\forall i} Y_{v_i} E_{v_i}$$

Subject to:

$$\text{For each } i, \sum_{\forall j} X_{(ki,j)} = 1$$

$$\sum_{\forall i} Y_{v_i} = 1$$

$$\sum_{\forall i} \sum_{\forall j} X_{(ki,j)} T_{(ki,j)} + \sum_{\forall i} Y_{v_i} T_{v_i} \leq RT$$

6.1.3 Solution to the Partitioning Problem

The formulation given in section 6.1.2 indicates that the problem is an integer linear programming (ILP) problem, which is NP-complete. In order to obtain a solution more efficiently, we use several characteristics of our system to obtain to a solution. In certain cases, optimal solution can be achieved.

First, the problem is reformulated into a flow problem on a directed graph $G = (V,E)$ as follows:

Variable descriptions:

I_i	where $i = 1..2n$ represents the i th interface between a kernel and a non-kernel, there are n kernels.
$M_{(ki,j)}$	represents the j th mapping for the i th kernel (HW or SW), with two costs - $T_{(ki,j)}$ and $E_{(ki,j)}$.
$M_{(nki,vj)}$	represents the i th non-kernel mapping, for each voltage level, v_j , where $j=1..m$. Corresponding costs are $T_{(nki,vj)}$ and $E_{(nki,vj)}$.

Steps to transform the ILP problem to the flow problem:

1. Convert all interface points (I_1, \dots, I_n) to nodes and add one source node to represent the beginning of the computation flow, and a sink node as the end where

$$V = src \cup snk \cup \{I_1, \dots, I_n\}$$

2. Convert each mapping of the kernels and the non-kernels as an multi-cost edge, with the corresponding energy and delay as the costs of the edge where

$$E = M_{(ki,j)} \cup M_{(nki,vj)}$$

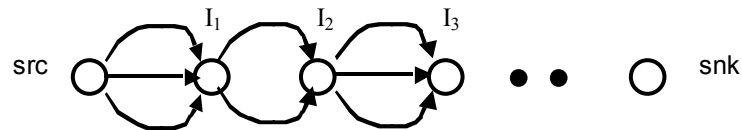


Figure 6-6-2 Graph Representation of the Partitioning Problem

Figure 6.2 shows part of the graph that represents the partitioning problem presented in figure 6.1. The most important observation that can be made from the graph is that a path from source to sink with the minimum energy cost and timing cost less than RT (Time constraint of the application) is the optimal solution to our partitioning problem.

Three characteristics of the cost metrics are explored which leads to three steps to solve the partitioning problem. In the solver, this three steps are attempted sequentially. If the problem fits the criteria of the step, the solver for that particular step is invoked and the remaining steps of the solver is not attempted.

Step 1:

Since the voltage of the coprocessors is fixed, if $t_1 \leq t_2 \rightarrow E_1 \leq E_2$ for all the mappings of all kernels, the optimal solution is straightforward. The solution is obtained by choosing the path with minimum delay, and it is also the path with minimum energy. The reason is that the problem space is convex, and any local minimum is also a global minimum.

Step 2:

If the criteria required by step 1 cannot be met, choose the path with minimum energy. If the corresponding time is less than RT, it is the optimal solution.

Step 3:

Lastly, A branch and bound algorithm is used to solve the partitioning problem if neither step 1 nor step 2 can be used to obtain a solution. In contrary to common belief, the number of kernels in an application is actually very few. In general, 10-20 kernels are the maximum number found in a realistic application according to literature [Cal98], and our experience with voice CODEC and communication algorithms.

6.1.4 Examples for the Partitioning Solver

In Table 6.3, several examples are presented. The number of kernels and the maximum number of mappings per kernel is presented for each application. The step used to solve each application is presented and the time it takes for the solver to get a

solution is presented. The table also shows if the solution obtained is optimal or not. The first 4 applications are derived from known DSP or communication algorithms. The last example is a randomly generated case.

Table 6-1 Examples and results for the partitioning solver

	Number of Kernels	Max Mappings per Kernel	Step from the Solver	Time (sec)	Optimal?
Adaptive MUD	3	4	1	0.06	Optimal
Viterbi Decoder	3	3	1	0.06	Optimal
JPEG decoder	5	2	1	0.06	Optimal
VSELP decoder	8	3	2	0.27	Optimal
Random	20	4	3	2.14	Heuristic

6.2 Architecture Instantiation

The output of the partitioning step described in section 6.1 is a mapping (represented as net-list) for each kernel in the algorithm. For most of the hardware/software partitioning problems, the union of all net-lists is taken to be the total hardware required and thus decides the final architecture. However, the reconfigurable architecture that the methodology is dealing with requires one extra step in order to deliver a complete architecture description. The reason is that many of the satellites can be shared from one reconfiguration to another. Based on the list of all mappings and the model of computation and reconfiguration, the type and number of satellites are determined in the following way in a stage called architecture instantiation:

First, during this architecture instantiation phase, satellites are divided into two categories: computational satellites (MAC, ALU, FPGA etc, defined as *Comp_Satellite*) and storage satellites (memory banks, defined as *Mem_Satellite*). The instantiation of *Comp_Satellite* is discussed in this section, and the design of the *Mem_Satellite* is presented in the next section.

To determine an architecture instance for the *Comp_Satellites*, two things have to be decided. The first decision to make is to determine the types of the *Comp_Satellite*, and the second is the number of instances for each type. Therefore, given all the kernel mappings to be run on the final architecture, the type of *Comp_Satellite* is the union of the types in each chosen mapping. The number of each *Comp_Satellite* type is set to be the maximum number of that particular satellite type among all kernel mappings. Figure 6-3 shows a simple example of the architecture instantiation, where there are only two kernels and all satellites shown are *Comp_Satellite*. The types of *Comp_Satellite* are A, B, C, and D in the example.

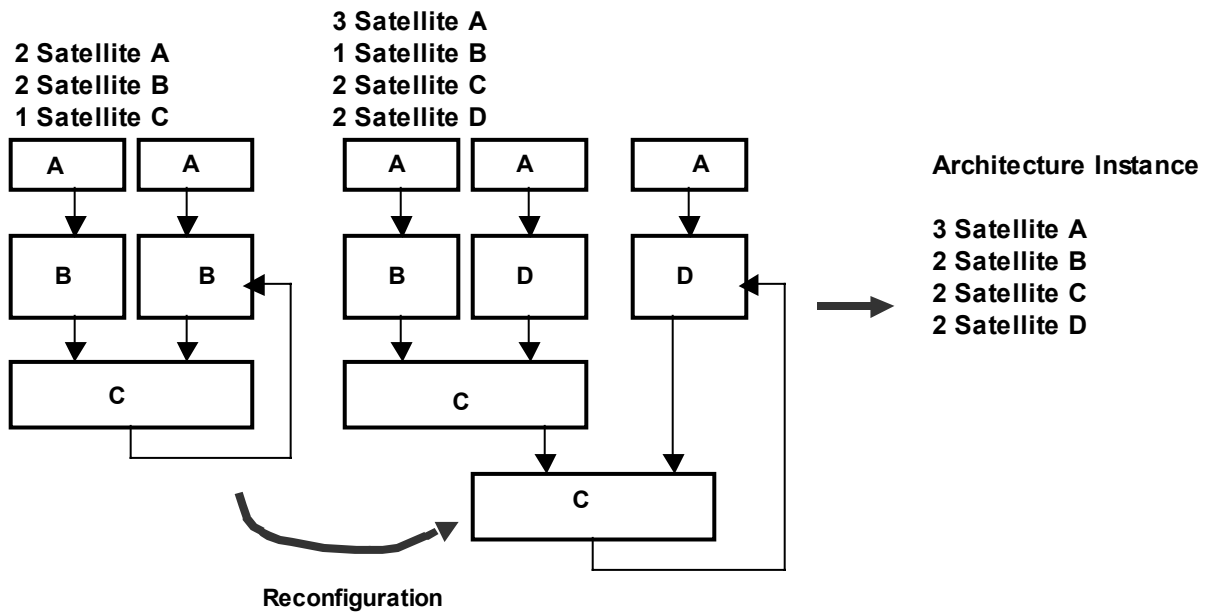


Figure 6-6-3 An Example of Architecture Instantiation

6.2.1 Data Allocation for Memory Satellite Instantiation

To instantiate the Mem_Satellite architecture, the following two decisions have to be made based on a set of kernel mappings: first, the different sizes of memory satellites needed; second, the number of satellites needed for each size. While the Pleiades architecture template promotes distributed memory banks to support data access parallelism *within* a kernel computation, it is undesirable to have one memory satellite allocated for each array of data in real designs. Consider the example where kernel 1 requires concurrent data access from array A and B, kernel 2 from B and C, kernel 3 from C and A. The maximum number of memories among all kernels is two if the Mem_Satellite architecture is instantiated like the Comp_Satellite in Section 6.2 (Figure 6.4a). However, in order to satisfy the required concurrency, the optimal number of memory is 3 (Figure 6.4b). On the other hand, it is often not realistic to assign a memory satellite for each array in the algorithm (because it will result in large number of small memories of variable sizes). Therefore, the sizes of Mem_Satellite and the corresponding number of satellites needed are more difficult to determine than what has been described for Comp_Satellite.

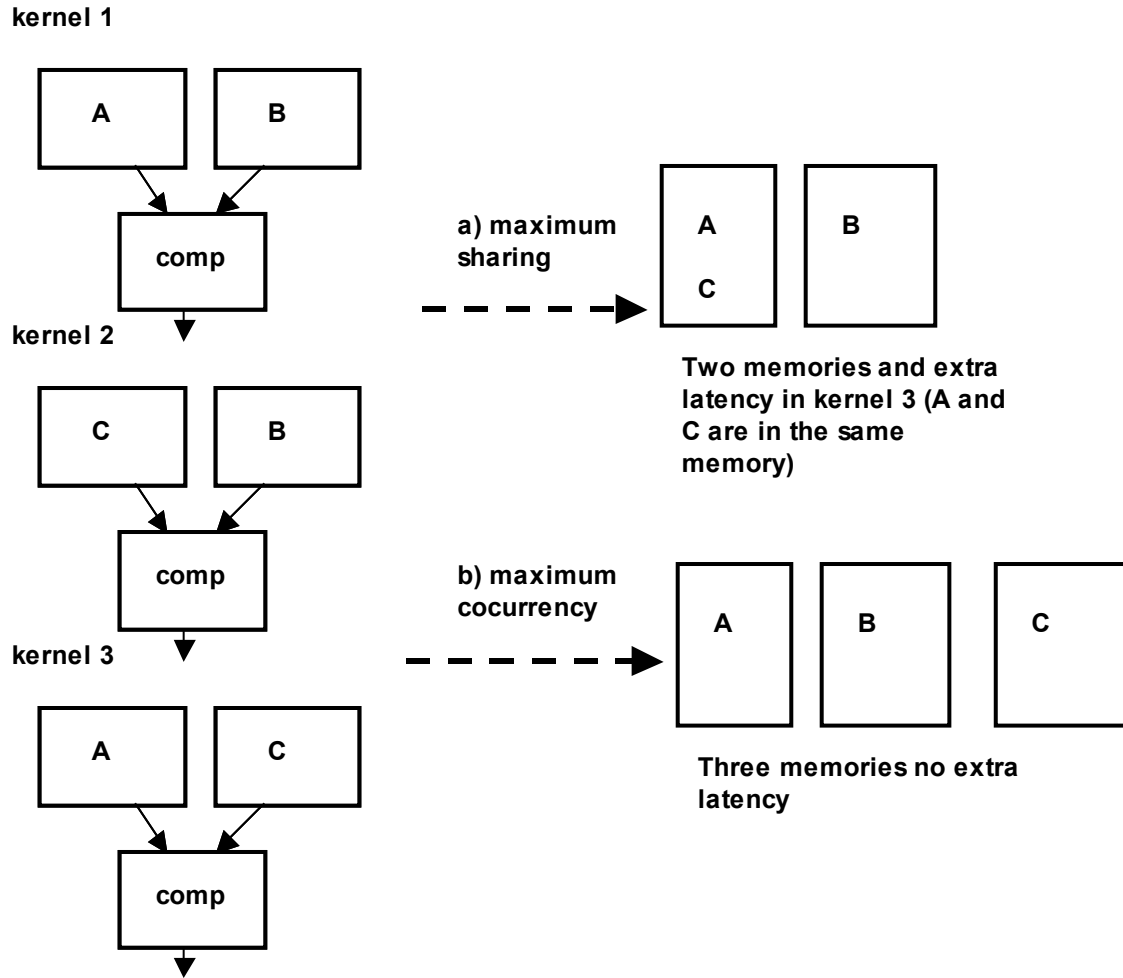


Figure 6-6-4 Example to show the difficulty in determining the memory satellite architecture

A methodology is devised to allow the designers to evaluate several memory configurations of their choice (a memory configuration provides the number of memories and the size of each memory) and decide on a configuration that provides the required concurrency with minimum number of memories and minimum energy consumed. In order to find out the best memory allocation given a memory configuration, a graph $G = (V, E)$ is constructed from the algorithm as follows:

$v_{\text{array}} \in V$ for each array in the kernel netlists, $\text{size}(v_{\text{array}})$ is the size of the array.

$e_{array_i,array_j} \in E$ if *array_i* and *array_j* need to be accessed concurrently in any kernel

The assumption is that N memories of size $\{S_1, S_2, \dots, S_N\}$ are given in the memory configuration. In addition, $\{E_1, E_2, \dots, E_N\}$ and $\{D_1, D_2, \dots, D_N\}$ are the energy and latency consumed to read or write one data from the memory. Another required input to the data allocation algorithm is the size and access frequency of each data array in all kernels (all information can be obtained using the profiling technique introduced in Chapter 4). The data allocation algorithm finds an N way partition for V with minimum connections between nodes in the same partition and minimum overall energy consumption gives the best memory allocation for the concurrency required. This NP-Complete problem is solved heuristically using simulated annealing. The benefit of simulated annealing is that the cost function can incorporate many different design metrics. In this case, the cost function includes the energy consumption for the given memory allocation (the access frequency of an array, A , multiplied by E_N where N is the memory A is allocated to) and the latency overhead (the access frequency of an array, A , multiplied by L_N where N is the memory A is allocated to, where A is one of the two end nodes of an edge that is not in the cut, and L_N is the larger of the memory latency of the two end nodes).

6.2.1.1 Data Allocation Case Study

An example on how to decide a good memory allocation for a voice CODEC application (VSELP) is described in this section. The inputs to the memory allocation tool are arrays used in the algorithm kernels (along with their size and access frequency in the kernel) and the energy consumptions to read from and write to the memory banks to be used in the final design.

In this design, 8 kernels have been extracted from the VSELP algorithm. There are 29 data arrays used in these kernels, and they have size ranging from 5 words to 350

words (each word is 16-bit). The total size of the data arrays sums up to 4796 words. A graph is constructed from the kernel mappings. The graph has 29 nodes (data arrays) and 21 edges (number of times an array needs to be accessed concurrently the same kernel with another array).

The three types of memory banks that can be used in this design are the 512-word bank, the 1024-word bank and the 2048 word bank. Their statistics are summarized in Table 6.2.

Table 6-2 Statistics of memory modules used in the architecture instantiation stage

Memory Size	Energy (Read & Write)	Latency in ns	Area in mm²
512 words	7.0 pJ	11 ns	0.16
1024 words	8.0 pJ	14 ns	0.32
2048 words	8.7 pJ	16 ns	0.64

Many memory configurations are first attempted. Some memory configurations, such as the configuration with ten 510-word banks, are not legitimate configurations even though the total memory size is larger than the total memory requirement. The reason is that there are many large arrays of size (350, 256 and 146, for example). Once the array is allocated to a memory bank, another large array cannot fit into the free memory space. Four final configurations are presented below for analysis.

1. Four 2K memory banks
2. Six 1K memory banks
3. Four 1K memory banks and four 512 memory banks
4. Fourteen 512 memory banks

Based on the data allocation output from the memory allocation tool, the energy consumption and latency overhead can be calculated. Table 6.2 summarizes the result for each configuration. It is clear that configuration 1 and configuration 2 are not good candidates for final implementation (they incur too much latency overhead since not enough concurrency can be supported by these two configuration). Configuration 3 and 4 are good choices since the latency overhead is smaller, and the energy and area costs are smaller than the first two configurations.

Table 6-3 Energy, latency overhead and total area results of data allocation for different memory configurations

	Energy	Latency Overhead	Total Area
Configuration	0.23 μ J	13.90 ms	2.56
Configuration	0.22 μ J	9.26 ms	1.92
Configuration	0.20 μ J	2.07 ms	1.92
Configuration	0.19 μ J	0 ms	2.24

In Chapter 8, the entire design flow for the voice CODEC chip is presented, the final design has memory configuration 3 since the design has a more stringent requirement on final area.

7 Reconfigurable Architecture Specific Explorations and Optimizations

Most of the previous chapters are devoted to presenting methods of identifying computational kernels and deriving a good satellite architecture for kernel implementation. In terms of implementations, the kernels are composed of not only the satellites, but also the interconnection between satellites and the software that configure the kernels. This chapter addresses another important aspect of the heterogeneous reconfigurable architecture design environment—providing methodologies to aide the optimizations of the reconfigurable interconnection and reconfiguration software.

7.1 Two Design Challenges of Reconfigurable Architectures

Reconfigurable architectures such as FPGAs have long been treated solely as prototyping platforms, not as final implementations for application or domain specific products. Since system-on-a-chip (SOC) designs have more stringent Power-Delay-Area constraints than most board-level systems, reconfigurable fabrics have not been widely used in SOC designs either. The reason is due to two inherent drawbacks of many previous incarnations of the reconfigurable architectures.

First of all, traditional reconfigurable architectures exhibit large energy overheads because of their reconfigurable interconnection network. Studies have shown that about 90% of the energy that an FPGA consumes goes into the interconnection network [Var00]. Even for some of the application specific reconfigurable processors, interconnection network and control logic take up to 75% of total energy used [Vis95].

Second, the time and energy required to reconfigure the hardware are usually extensive [Haw98] also. Recently, many research projects have made major inroads to improve reconfiguration efficiency using either software [Haw98] or hardware [Hau97] techniques. However, much improvement is still needed to make reconfigurable architectures a more practical approach for SOC designs. The recent search for a flexible, yet efficient, implementation platform has made the embedded reconfigurable architecture a very likely candidate to be included in the SOC design.

The main idea behind reconfigurable computing is to build a computational engine through a spatially programmed connection of processing elements. The interconnect model to be considered by this thesis is depicted in Figure 7.1. The graph in Figure 7.1a shows the flow of control within an application. In this example, two kernels are spawned off from the main program. In Figure 7.1b, the configurations of the two kernels are shown. On the time versus configuration graph (Figure 7.1c), the boxes in-between reconfiguration on the time-axis represent a set of inter-module connections that has to be realized simultaneously. Note that a configuration period can vary from one single clock cycle, over a single duration-limited function, to the entire duration of an application in a multi-function system [Kal95].

This reconfiguration concept requires the underlying interconnect architecture to support sufficient concurrency within a configuration, while allowing for time-sharing of resources across configurations. An efficient interconnection network hence must have sufficient flexibility to support the required connection patterns, while still maintaining good performance and efficiency for each configuration. In terms of the configuration time and power, the coarser granularity of the Pleiades architecture guarantees significantly less overhead than the fine-grained FPGA. However, better configuration timing and energy can still be achieved by taking advantage of features of the architecture and applications.

For the rest of the chapter, basic concepts and further optimizations that can be used to improve the interconnection architecture (Section 7.2) and reconfiguration software (Section 7.3) are discussed. The key contribution of this research is the provision of appropriate models and environments to quantitatively evaluate these hardware and software choices.

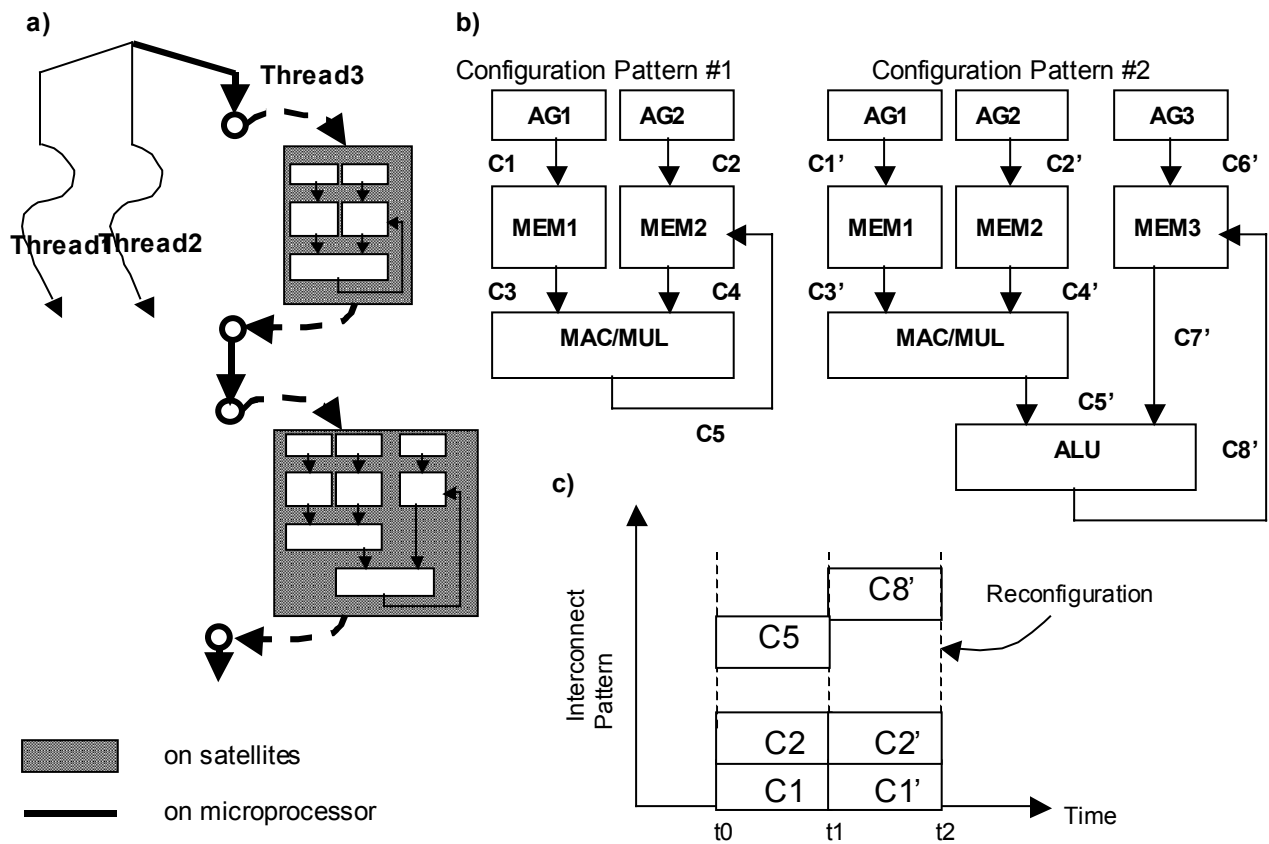


Figure 7-1 The Flow of Computation and Concept of Reconfiguration

7.2 The Reconfigurable Interconnect Architecture Exploration Environment

When designing an SOC, the suitability of different reconfigurable interconnection architectures has to be quantified and the parameters associated with each architecture have to be determined. This section describes a methodology (shown in Figure 7.2) to

evaluate the architectural effects on the final system. The methodology first compares architectures with each other in an application independent way. A retargettable interconnection evaluator is introduced for this purpose. Second, applications are mapped to all the architectures, and the cost of each implementation is evaluated. The mapping process is usually considered a laborious procedure, and is not employed in the traditional architecture design cycle. By using the methodology in this thesis, the kernel net-lists and the frequency of each kernel in an application are easily transferred to the architecture designer. When the net-list and frequency information of kernels are combined with the retargettable interconnection evaluator, the final impact of choosing a particular architecture is realized rapidly.

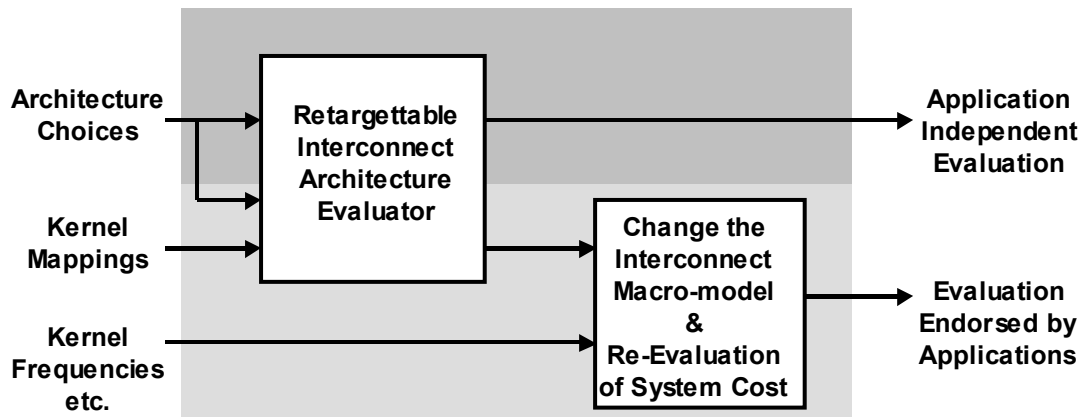


Figure 7-2 The Evaluation Environment for Reconfigurable Interconnection Architectures

There are many performance metrics that can be used to evaluate the final implementation. Since keeping the energy of the reconfigurable interconnection network as low as possible while still meeting the flexibility requirement is crucial to the success of the approach of heterogeneous reconfigurable architecture, energy serves as a good metric to evaluate various architectures. In the next sub-section, an overview of reconfigurable interconnect architectures is first given. Next, a model used to quickly provide quantitative feedback of energy efficiency of all architectures is presented next.

Results from using the methodology to design reconfigurable interconnection architecture for sets of kernels are detailed at the end of section 7.2.

7.2.1 Overview of Reconfigurable Interconnect Architectures

In this sub-section, several reconfigurable interconnection architectures suitable for SOC implementation are discussed. Taking knowledge from both the multi-processor network topology and reconfigurable interconnect for fine-grained FPGAs [Var94][Zha99], many reconfigurable interconnect architectures are introduced in this section. These architectures are introduced in terms of increasing design complexity and decreasing energy consumption.

Crossbar and Multi-Bus Network

A crossbar interconnect network allows simultaneous connections from any input port to any output port. Figure 7.3a shows a conceptual view of such a network. One possible implementation of the crossbar network assigns each input to a global bus, and each of these busses can be connected to any outputs through a switch. Observe that such a network requires only one switching stage, which means that every input and output pair is connected through a single switching element. This architecture provides full connection flexibility, but suffers from a large area overhead (due to wasted wire resources) and a high energy consumption (due to the long global buses and the large number of switches). A multi-bus interconnection (Figure 7.3b) is an optimized cross-bar where the number of busses is reduced to that required by the application.

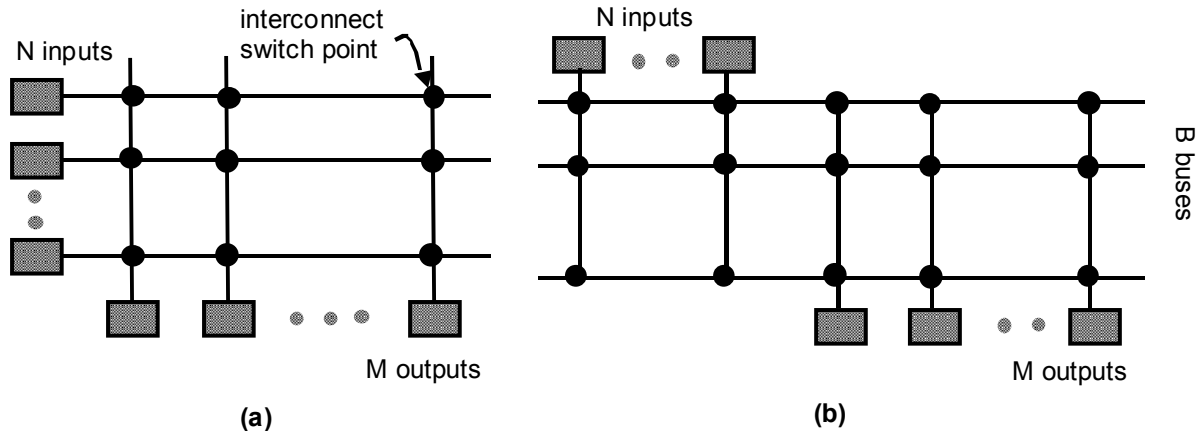


Figure 7-3 Global Interconnect Architectures (a) Crossbar Network (b) Multi-Bus Network

Generalized Mesh

More energy-efficient architectures can be conceived by taking advantage of the locality of most of the connections. In the multi-processor world, many network topologies [Xil][Var94] have been proposed that provide point-to-point links between neighboring network nodes at the expense of connections between distant nodes. Examples are the ring, the 2-D mesh, the generalized hypercube, and the torus. The mesh interconnection network, which is utilized in most contemporary FPGA's, is an example of how these network concepts can be applied to the single-chip reconfigurable interconnect problem. Figure 7-4a shows a simple FPGA interconnect architecture. It consists of switch matrices with limited connectivity (also called S-boxes or switch-boxes) connected by wiring channels. Connection-boxes (or C-boxes) provide the connections to the ports of the computational elements. The mesh network shown in Figure 7.4a has the advantage that local interconnections are implemented efficiently due to the segmentation of the network: only the segments really necessary to provide the connection are activated.

While attractive due to its simplicity, the FPGA mesh network, like all the interconnect structures introduced in this section so far, is not directly applicable to reconfigurable systems in the style of the Pleiades architecture. Since the system is

composed of heterogeneous modules of different shapes and sizes, no regular 2-D grid in the style of homogeneous FPGA can be found. Therefore, a “generalized mesh” structure is proposed, shown conceptually in Fig. 7.4b. A generalized mesh structure is constructed in the following manner: given a placement of the modules, wiring channels are created along the sides of each module. Wherever channels meet or cross, a switchbox is provided. The design parameters of such a network are the number of buses in each channel, which can be optimized individually according to the connectivity requirements, and the connectivity provided by the S-boxes.

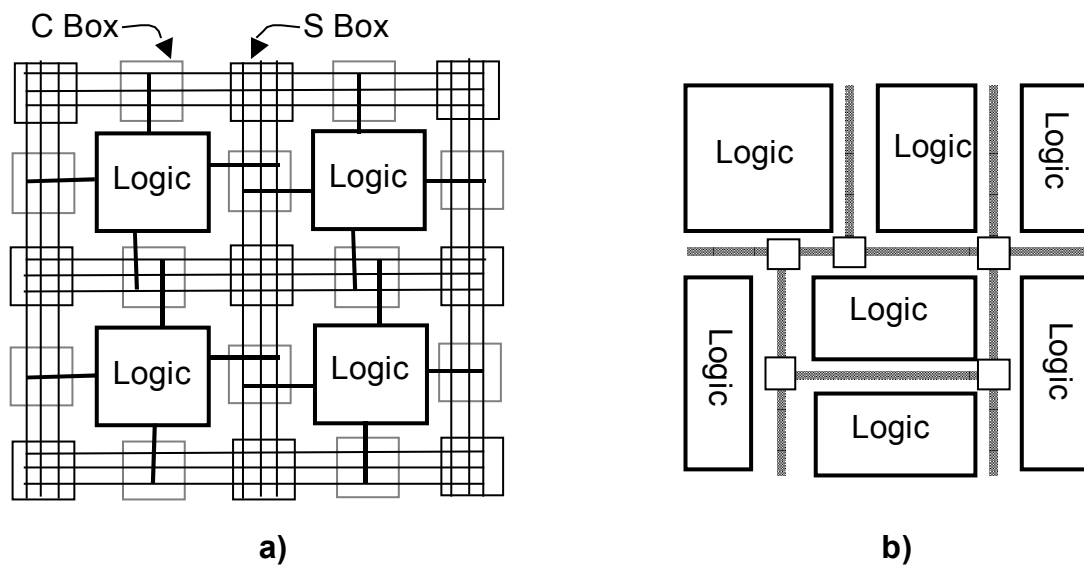


Figure 7-4 a) Regular Mesh Interconnect Structure for Homogeneous Modules
b) Generalized Mesh Structure for Heterogeneous Modules

Hierarchical Interconnect Network Architecture

Energy-efficient architectures must take advantages of the locality and regularity of computation. Exploiting locality by identifying naturally isolated clusters of computational operations and partitioning the hardware accordingly result in the minimization of global busses, thus reducing the interconnect power [Meh95]. Once

natural clusters are formed, mesh, multi-bus or cross-bar networks can be used to connect intra-cluster satellites. Another hierarchy of interconnection network can be placed on top of the clusters (Figure 7.5 shows a hierarchical mesh network). Although the underlying system is heterogeneous, the DSP algorithms usually have inherently regular computation patterns. Partitioning the hardware by preserving such regularity and building such hierarchical interconnection network lead to simpler interconnection structure with reduced fan-ins and fan-outs. Especially for reconfigurable architectures, more regular interconnection architecture achieve better routability and less reconfiguration overhead.

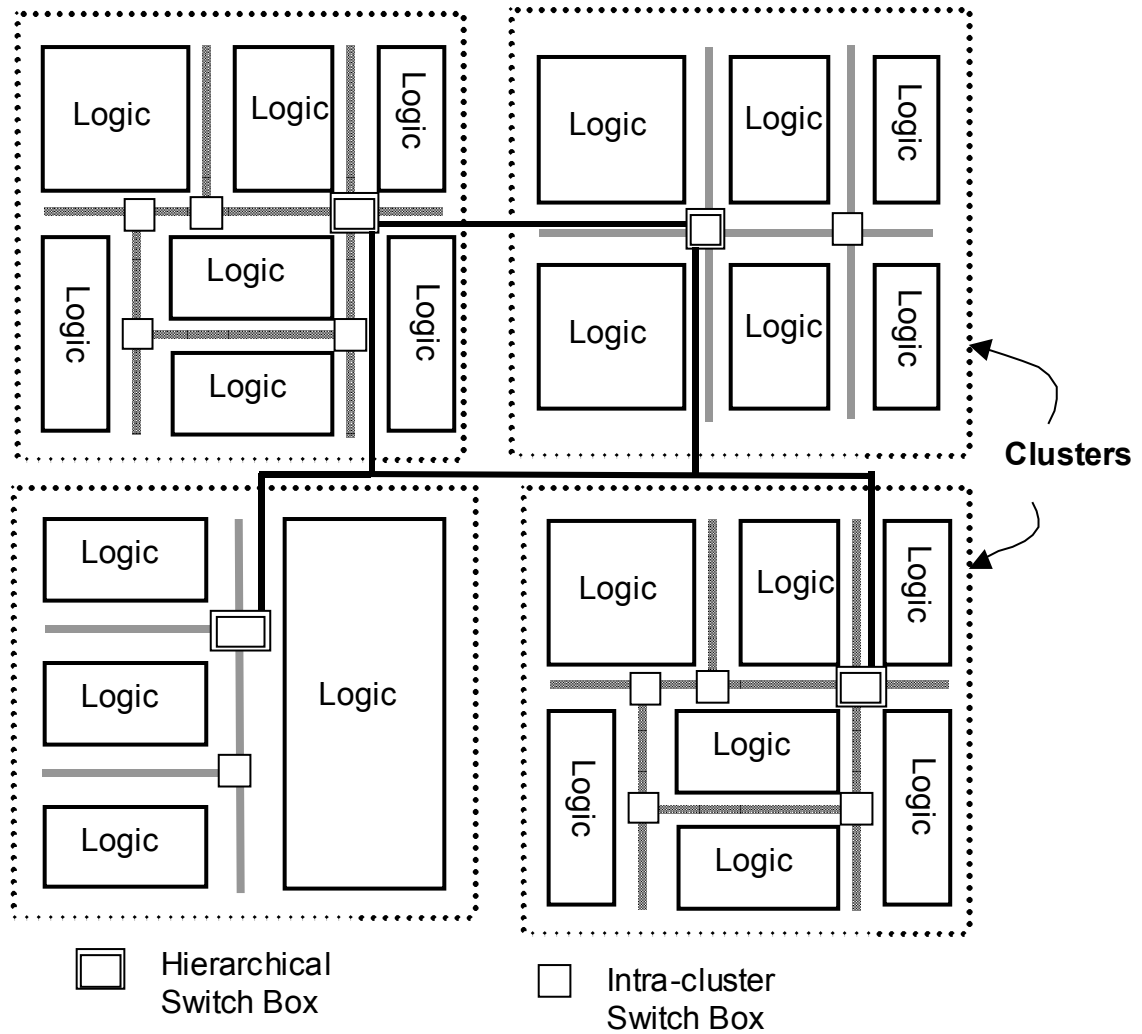


Figure 7-5 Hierarchical Generalized Mesh

7.2.2 The Evaluation Environment for Reconfigurable Interconnect Architectures

The three categories of architectures (cross-bar/multi-bus, mesh and hierarchical mesh) need to be quantitatively compared against each other. Global interconnect structures (cross-bar/multi-bus) are attractive because of their simplicity and ease of programming. Domain-specific architectures (mesh and hierarchical mesh) exploiting locality have to yield dramatic improvements in performance with respect to these

generic structures to be of interest. Incremental gains in either energy or speed are not worth the loss in flexibility and the more complex design process.

In order to compare these three architectures, a retargettable interconnect architecture evaluator is used which is based on a combination of accurate physical models with an architecture-independent router [Alex95][Var99]. The detailed description of the evaluation and model is as follows.

The Retargettable Interconnect Architecture Evaluator

Given an instance of the satellite architecture (the output of the tools described in Chapter 6), some manual steps (such as placing modules and clustering) are carried out to generate a floor-plan so that the interconnection architectures can be explored. All the implementation parameters (type of architecture, number of buses in the channel, etc.) are optimized iteratively with the help of a graph-based router that can route each net on the underlying network architecture to predict the energy and delay data. In this router, each reconfigurable interconnection architecture is described as a graph, G , with the vertices representing the ports and switches, and edges presenting feasible connections between ports and switches. Figure 7.6 depicts how a network of two satellites with one switch box (Figure 7.6a) is modeled as a graph (Figure 7.6b). Each edge in the graph has a weight associated with it to represent the energy consumption. Finding an optimal route between two ports is formulated as finding the minimum-weight Steiner tree on the graph G . A heuristic algorithm proposed by [Alex95] is employed to solve this NP-complete problem. Finding routes for all nets in a kernel net-list such that the total cost is minimized is achieved by using simulated annealing (finding out the cost for routing the nets according to a randomized order).

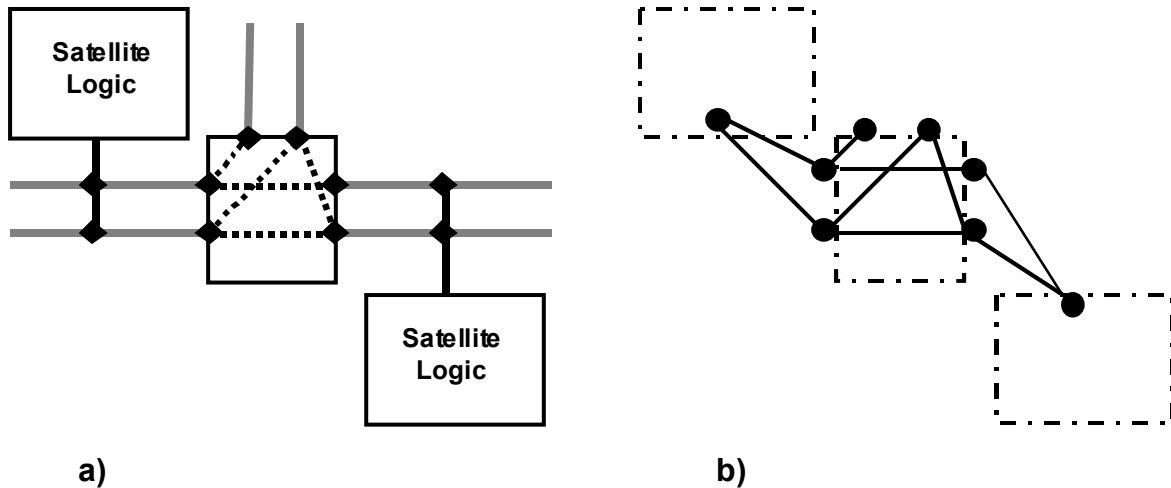


Figure 7-6 Graph-Based Reconfigurable Interconnect Architecture Modeling

An analytical energy model for each net is used to compute the weight for each segment of a route. An universal energy model for the interconnection network is derived (assuming that all the wires are switching) from the following formula:

$$\begin{aligned}
 E &= [C(\text{wire}) + C(\text{switches})]V^2 \\
 &= K_L \cdot L + K_{SW} \cdot W_{SW} \cdot N_{SW}
 \end{aligned}$$

Each variable in the formula is obtained as follows. K_L (in unit pJ/mm) is the energy per unit wire length, and K_{SW} (in pJ/um) is the energy per unit switch-width. W_{SW} represents the switch width, and N_{SW} is the number of switches. K_L , K_{SW} and W_{SW} are characterized coefficients for a given VLSI technology. For each bus of the architecture under evaluation, N_{SW} is derived from the interconnect architecture and the satellite architectures. In this design methodology, it is decided that all channels have the same number of buses. The number of buses is decided by specifying an instance of the interconnect architecture that supports the kernel that has the most

connections (the router specified above is used to determine that). Once the number of buses is decided, the width of each channel is fixed as well. The variable L can then be determined for each bus.

Two Metrics for the Evaluation

The retargettable architecture evaluator is used to provide quantitative comparisons between all architectures in both application independent and dependent ways.

To evaluate the architectures in an application independent way, all the satellites in the architecture instance are ordered as follows: a reference satellite port is chosen and all other satellite ports are ordered according to their manhattan distance (on the floor-plan) from the reference satellite port. The shortest route from the reference port is obtained for each satellite port and the cost recorded. This cost gives the best performance that an architecture can provide between the two ports (since no other routes are present). Therefore, this cost gives a good indication on the energy efficiency of the architecture.

Since any meaningful kernel is constructed by more than one connection, the next thing is to route all the kernel net-lists on each architecture, and record the total interconnection cost for all kernels. Since the nets from one kernel can cost congestion for each other, an architecture that provides the best cost in our first metric might not perform very well when this metric is used.

Results on Reconfigurable Interconnect Architecture Exploration

In this section, the results from exploring reconfigurable interconnect architectures for a CELP-based voice processor are presented. The inputs to this exploration environment are the kernel mappings of the application (VSELP) and a floor-plan of the

architecture instance. How the architecture instance is obtained for this application is detailed in Chapter 8 where a step-by-step example is presented. Only the evaluation of difference interconnection architectures are discussed in this sub-section.

The satellite-port-to-port energy costs are first computed for the interconnection architectures in consideration (multi-bus network, mesh and a 2-level hierarchical mesh). As shown in Figure 7.7, multi-bus network gives the worst energy efficiency and the hierarchical mesh network provides the least energy cost. On average, the hierarchical mesh network is about 2 times better than the mesh network and 3 times better than the multi-bus network.

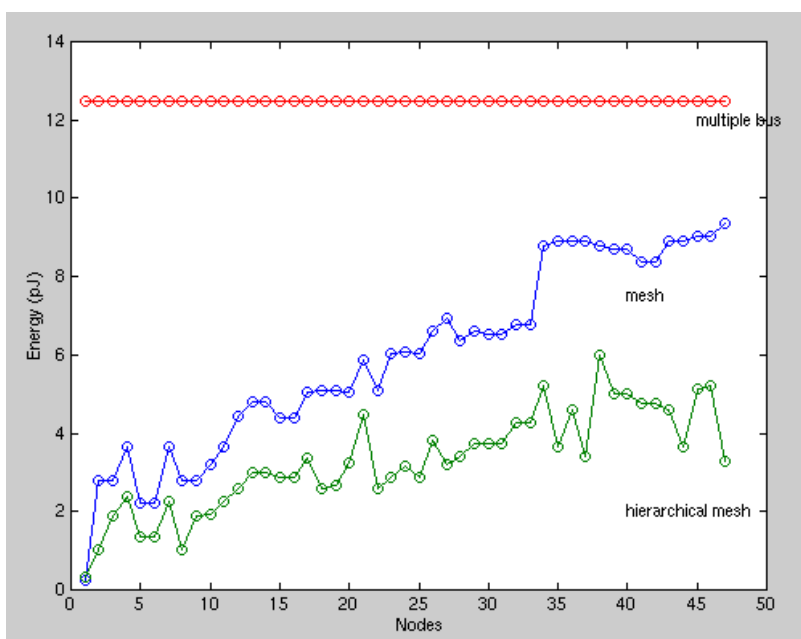


Figure 7-7 Satellite to Satellite Energy Cost for Different Reconfigurable Interconnect Architectures

To understand how applications influence the performance of each interconnection architecture, all kernels of the VSELP algorithm are routed on the architectures. Table

7.1 displays the energy consumption of interconnection for three dominant kernels after they are routed on the interconnection architectures. The results are shown for one iteration of the kernels. It is discovered that the hierarchical mesh network is 1.47 times better than the mesh network in terms of energy efficiency, and 4.6 times better than the multi-bus.

	Dot_product	Vector_sum_with scalar_mul	IIR
Multi-bus	50.0 pJ	50.0 pJ	138.0 pJ
Mesh	17.7 pJ	14.7 pJ	43.4 pJ
Hierarchical Mesh	11.1 pJ	10.2 pJ	31.3 pJ

Table 7-1 Kernel Benchmark Results for Different Reconfigurable Interconnect Architectures

Figure 7.8 shows the total energy consumption of the satellites (modules and interconnection) for the VSELP algorithm. The results are presented for all three architectures under consideration.

Both the mesh and hierarchical mesh networks are able to cut the total power by 20mW when compared to the multi-bus network. In the case of hierarchical mesh network, the interconnection consumes only 2mW, while the interconnection consumes about 6mW in case of the mesh network. The hierarchical network architecture requires only a limited number of buses to achieve sufficient connection flexibility for the target application, and cuts the interconnect energy cost by a factor of 7 compared to a straightforward crossbar network implementation.

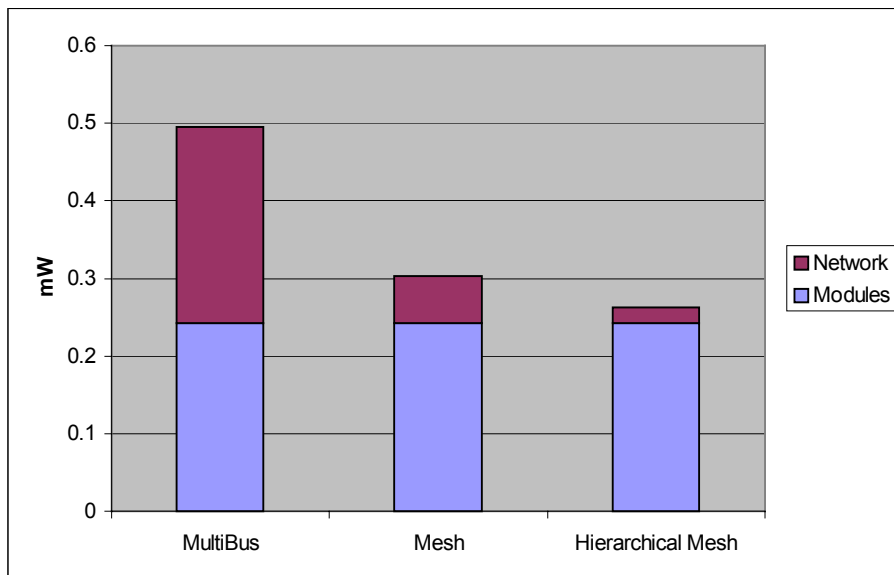


Figure 7-8 Breakdown of Kernel Energy Consumption for Different Interconnection Architectures

7.3 Reconfiguration and Interface Code Generation and Optimization

This section is devoted to another important design challenge of reconfigurable systems—overhead for reconfiguring the reconfigurable. The problem of interface generation between hardware and software has recently gained significant attention in the VLSI CAD community. The problem of integrating processors with reconfigurable elements has added another dimension to the interface generation problem—between software and reconfigurable. Careful reconfiguration and interface code generation is essential [Haw95] to ensure that the overhead of reconfiguring will not offset the speed and energy savings of reconfigurable components. This need is especially pronounced when the reconfiguration frequency is large within an application and when the timing constraints on the application are tight – which is often the case for real-time DSP and communication applications.

In this section, a code generation and optimization process for reconfigurable architectures targeting digital signal processing applications is presented. While the concept of the code generation and optimization process is machine-independent and can be applied to any heterogeneous reconfigurable architecture, the effectiveness of the code generation process is shown on the Pleiades architecture template. Similarly, while the process can be utilized as the backend to any system compilation tool, we use the software methodology proposed in this thesis as the front end to obtain a good software-configware partitioning. A detailed description of the proposed code generation and optimizations is given in section 7.3. At the end of this section, we illustrate the effectiveness of the optimization process by providing results for a base-band voice processing architecture (the example presented in section 7.2).

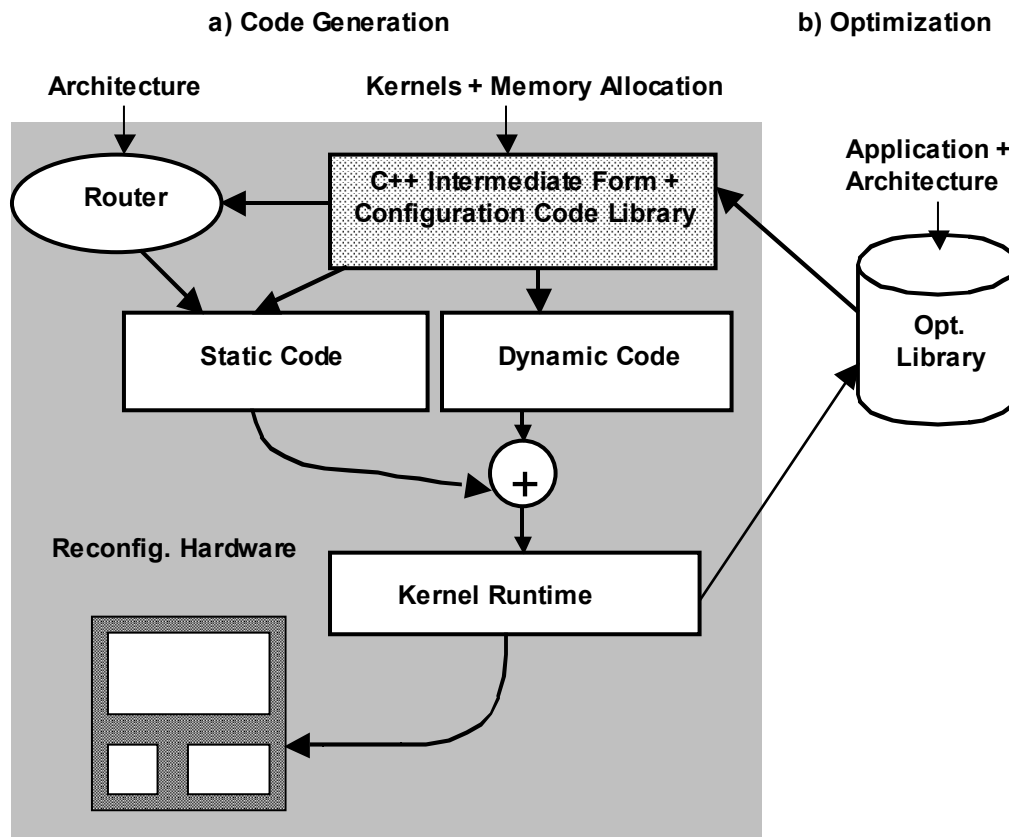


Figure 7-9 Basic Flow for Configuration and Interface Code Generation

7.3.1 Basic Code Generation Techniques

Figure 7.9 shows in detail the components of the code generation step. The inputs to this step are obtained from steps 1 through 3 in the design methodology (Figure 3.3) as well as the interconnection architecture evaluator described in the previous section. At this stage, all the kernels mapped to satellites are encapsulated in procedure calls and the kernel net-lists are specified in the C++ Intermediate Form (C++IF, first described in Chapter 5). A library of reconfiguration codes for each satellite and kernel interface is supplied to the code generator (Figure 7.9a). Since the kernels specified in C++ Intermediate Form has all satellite configuration and connectivity information, reconfiguration and interface code is automatically generated from the kernel procedure calls in C++IF. Once the reconfiguration code is generated for all kernels, bottleneck

analysis is performed on the code generation results and optimizations are performed to reduce the most dominant reconfiguration or interface code type (shown in Figure 7.9b).

To best understand the kind of optimizations that can be performed on the code generation process, reconfiguration code is further categorized as follows:

Based on the time that the complete content of the reconfiguration code is determined, the reconfiguration code is defined as either static or dynamic. If the reconfiguration and interface code can be determined at compile time, it is defined as static, while if it can only be determined at runtime, it is dynamic. It is more efficient to execute static code. Based on the functionality, the different types of reconfiguration code are divided into three categories (shown in Figure 7.10a, Figure 7.10b and Figure 7.10c). Each category has its unique characteristics that help us determine if it favors static or dynamic configuration. The three categories are detailed below.

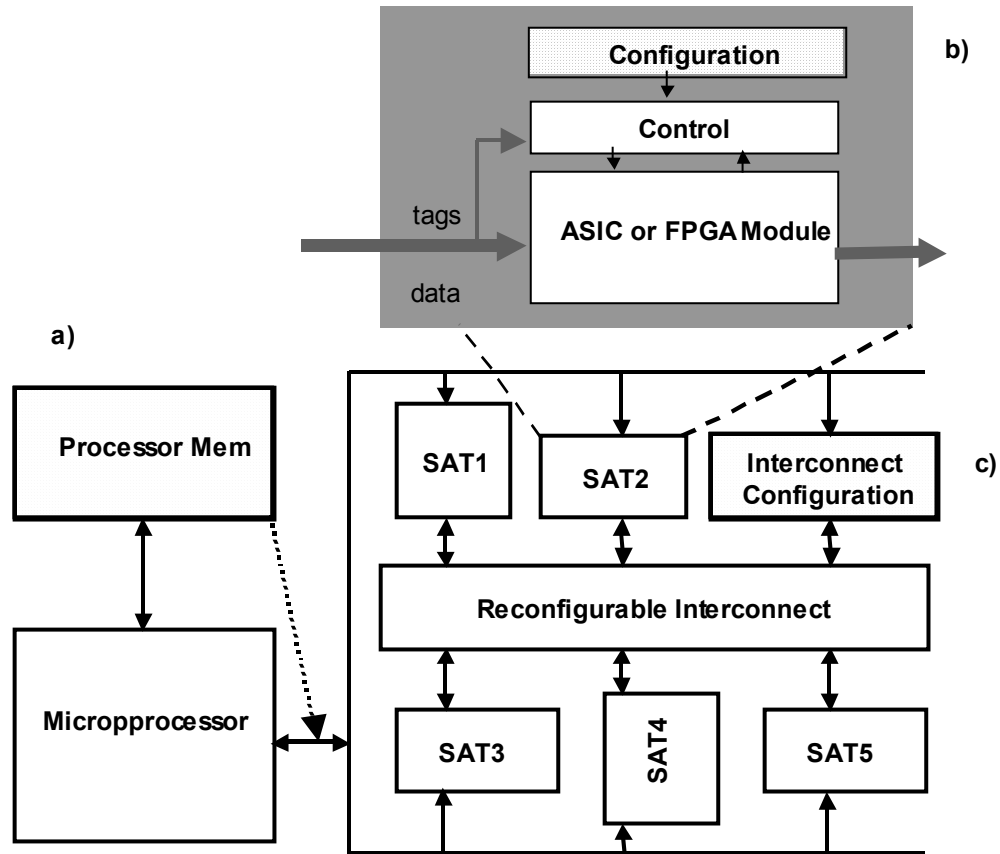


Figure 7-10 Three main categories of configuration code and where they reside (a) Processor-Satellite interface code (b) Satellite configuration code (c) Interconnect configuration code

Processor-Satellite Interface code:

This interface code takes care of the synchronization between the microprocessor and satellites (Figure 7-10a), including resetting of all satellite processors and network before kernel execution. Due to memory and performance constraints and the simplicity of the software/reconfigurable synchronization scheme, an off-the-shelf real-time operating system for the microprocessor is not used. Instead, a more streamlined version of the operating system is generated. These codes are statically determined

based on the processor-satellite interface specification. A generalized interrupt-driven interface code is used as the baseline code.

Configuration code for the satellites:

For a given kernel, most of the operations performed on satellites are known, and therefore the configurations are statically generated. For example, the configurations of storage or computational satellites such as SRAM, ALU and MAC/MUL are fixed once a kernel is mapped. In this case, the configuration code is statically determined. One special satellite is the address generator processor (AGP). It is a micro-programmable satellite and requires a compile time program and runtime configuration. The configurations of AGPs vary from one invocation of kernel to another. Therefore, the parameters have to be passed in from the main application program and reformatted at run-time before they are sent to the satellites for configuration. In this case, the AGP programs are determined statically and the configuration has to be dynamic.

Configuration code for the reconfigurable interconnects:

Configuration code is needed to establish connections for a kernel (Figure 7-10c). It is very time-consuming to perform runtime routing for the reconfigurable interconnection network. Therefore, the net-list is routed on the underlying architecture using the router described in section 7.2.2 at compile time. In this case, the configuration is static.

Using the above basic criteria for dynamic and static codes, and the library of configuration codes, a baseline reconfiguration code sequence is determined for all kernels in an application. However, generating efficient, yet modular, interface and reconfiguration code for kernels embedded within a larger application involves many more tradeoffs and optimizations which we will describe in the later sections. First of

all, an analysis on the baseline configuration code is carried out and analyzed in the next section.

7.3.2 Bottleneck Analysis for Configuration Code Optimization

The total configuration codes for the entire application and for each kernel are shown in Table 7-2. The second column of the table represents the frequency of kernel invocations within 1 second of VSELP decoding. The third column represents the configuration for the specific kernel. Figure 7.11 shows the percentage breakdown of the total configuration code based on the code categories (computational satellites, AGPs, interrupt handler for processor-satellite interface and reconfigurable interconnection). This figure shows that the code categories with the highest cost are the AGP configuration, processor-satellite interface interconnect and AGP program. The reconfiguration of interconnection consumes a lot of energy as well but the overhead is fixed and cannot be optimized. In the rest of this section, a set of local and global optimizations is presented to reduce the cost of reconfiguration code [Li99].

Kernels	Number of Invocation in VSELP	Kernel Reconfiguration Cycles	Total Cycles
FIR	4584	363	1663992
VSMUL	2358	292	688536
IIR	250	454	113500
Dot_Product1 (1 Mac, 1 AGP)	7880	226	1780880
Dot_Product2 (2 Mac, 2 AGP)	83802	300	25140600
Compute_Code	997	363	361911
			29749419

Table 7-2 Total cycle counts per second for configuration code in VSELP

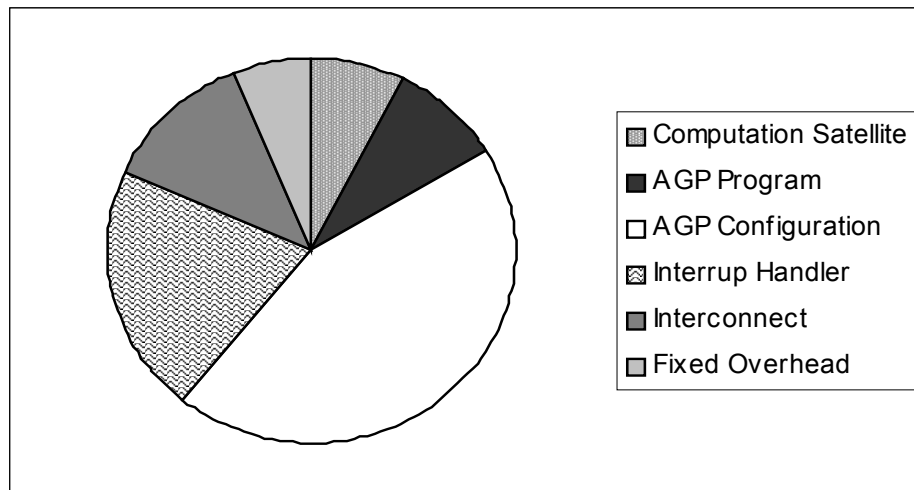


Figure 7-11 Percentage breakdown of configuration code based on code category

7.3.3 Local Optimizations

Local optimizations are the ones that are carried out at the kernel level and global optimizations are performed at the application level. A couple of local optimizations are carefully evaluated to produce a good executable code sequence. To generate the baseline code whose results are presented in section 7.3.2, configuration codes for the satellites and kernels are generalized. No kernel specific optimization is used. However, one must customize configuration code for different kernels and applications to achieve satisfactory performance while balancing the development effort and code complexity.

Kernel Specific Configuration and Interface Code

It is desirable to have customized code routines to take advantage of the special properties of the target application and the kernels involved. System bottleneck analysis (Section 1.3.2) indicates that address generator reconfiguration is the dominating factor during kernel reconfigurations. This can be explained by the fact that its configuration has the most dynamic component. Since not all fields of the AGP configuration can be

determined at compile-time, reformatting of all fields at run-time is performed in the base-line code in order to provide generality. However, for a given kernel, only parts of the AGP configuration register fields are relevant and the rest is treated as don't cares or can be formatted at compile time. Thus it is advantageous to provide customized AGP configuration routines for the different kernels.

The generality vs. performance/power trade-off issue is also present when dealing with the communication from the satellites to the microprocessor. In order to handle applications with multiple parallel threads, the interrupt-driven communication primitive in our interface library is provided. That is, the coprocessors interrupt the embedded microprocessor when they want to communicate. This way communication in a multi-threading environment could be orchestrated in a clean and organized way. Note that sophisticated communication primitives such as interrupt handling and polling do have significant overhead. While for applications that have lots of parallelism to exploit this overhead might be necessary. For applications that are sequential in nature, this expensive overhead cost is not worthwhile to pay. In such case, simpler and cheaper primitive might be more suitable. For example, the VSELP algorithm is mostly sequential, so the sleep and wakeup power saving mode of the processor is sufficient to implement the communication primitive. In this simply primitive, the microprocessor goes to sleep after it finishes configuration and triggers Pleiades to start running, then Pleiades wakes the microprocessor up when it is done. By replacing the interrupt scheme by the sleep-and-wake communication primitive, significant savings are achieved. The behavior of a interrupt service routine is replace by kernel specific codes that copies data from satellites to the microprocessor if required by the kernel.

7.3.4 Global Optimizations

Besides performing optimizations at the kernel level by choosing the right modularity and providing a library of optimized code for each kernel, it is also important

to exploit the special features of the underlying reconfigurable architecture and the application structure to generate better configuration code sequence.

Configuration Program Cache.

Multiple-context FPGA [Kal97] has been proposed as one of the architecture solutions to reduce configuration cycles in between reconfigurations. Our underlying architecture also supports limited multiple-context configurations. In particular, the AGP instruction registers are deeper and can support up to 5 contexts. Since kernels within DSP applications have only limited instruction patterns, all AGP instructions can be stored in the reconfigurable satellites without reconfiguration from the processor. In this case, a kernel can be reconfigured without loading AGP programs from through the microprocessor.

Partial Reconfiguration:

When two identical kernels are called sequentially, only part of the configuration data has to be loaded into the satellites. Specifically, since a particular kernel has a fixed satellite cluster structure and fixed operations, the only configurations that have to be reloaded are the interface code, AGP configurations (for possibly different starting and ending address, vector length etc.) and interconnect. Another module is added to the SUIF [Lam90] compiler front-end to discover if a kernel procedure is in a nested loop (Figure 7.12a) or identical kernels appear consecutively in the same control flow (Figure 7.12a).

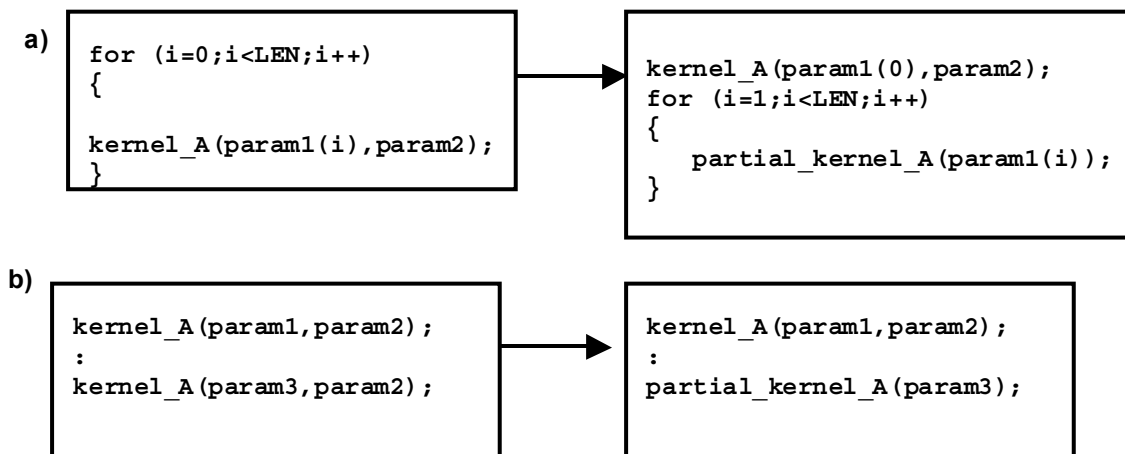


Figure 7-12 Two Kinds of Partial Configuration for Kernels

7.3.5 Test Cases, Results and Discussion

The software reconfiguration optimizations described in this chapter is applied to a voice CODEC application (VSELP) that is mapped to a heterogeneous reconfigurable platform. Note that in this section, the optimization of reconfiguration energy consumption and performance is translated into the minimization of total number of reconfiguration cycles on the microprocessor. The list of kernels mapped to the satellites as shown in Table 7.2 is displayed on the right of Figure 7.13. As presented in Table 7.2, the total number of cycles (on the microprocessor) needed to reconfigure all the kernels in the application is around 30M per second (before optimization). From left to right, Figure 7.8 shows how much improvement each optimization provides (kernel AGP configuration, kernel specific interface code, AGP program cache and partial configuration). The total number of cycles after all optimizations is 11M cycles which is one third of the original baseline code.

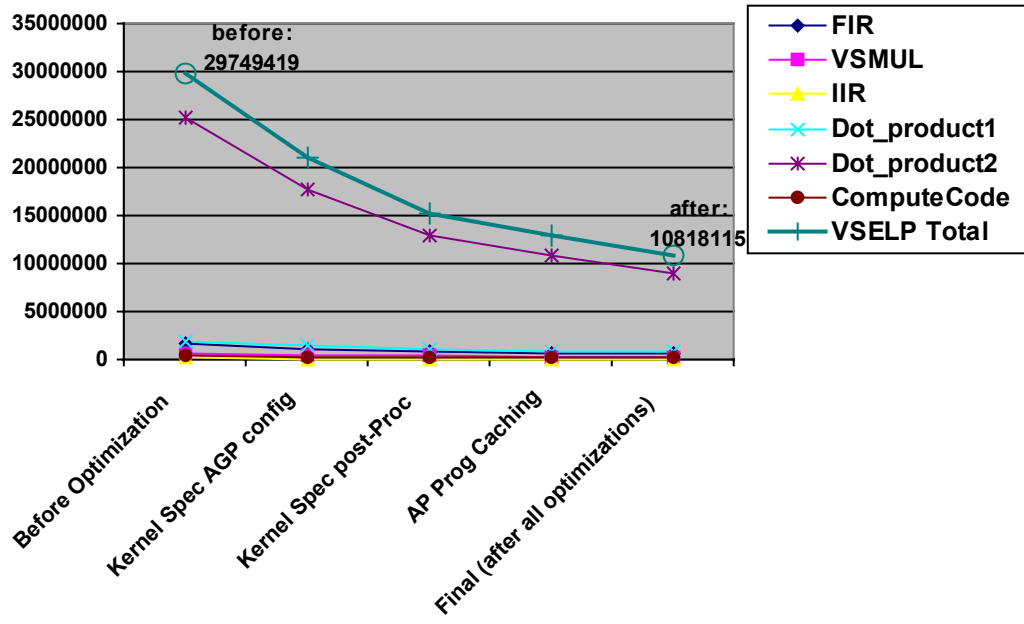


Figure 7-13 Reconfiguration Optimizations and Savings

8 Architecture Design Examples Using the Methodology

This chapter presents the design process of a heterogeneous reconfigurable processor using the methodology described in this thesis. The input is an algorithm specified in C along with the real-time constraint of the application. The methodology is followed step-by-step, and the result is a low-power heterogeneous reconfigurable processor. The effectiveness of the methodology is shown in two ways. First, it is shown through the optimizations that the methodology enables the designers to do during the design process (kernel extraction, hardware and software optimizations). Second, the low energy consumption of the resultant reconfigurable processor is compared with the energy of the application running on a programmable processor.

8.1 Design of a Reconfigurable CELP-based Digital Signal Processor

This case study is a design of a CELP-based (VSELP, LDCELP etc.) digital signal processor (called Maia) [Zha00]. The algorithm used to map to the architecture is a 16-bit fixed-point algorithm of VSELP written in C. The specification requires the coder to process 50 speech frames per second.

Algorithm Characterization and Refinement

The algorithm in C is simulated and the basic-block execution frequency is gathered using the `source_profiler` tool (Chapter 4). The function-level breakdown of algorithmic complexity for the algorithm is obtained, and the information is back-annotated to the source file. It is shown that one of the functions, `Quantize_Gain`, represents considerable complexity (9% of the total algorithm). The annotated source file indicates that a loop (86.46% of the function) exists that can be extracted into

kernel functions. Therefore, the loop is decomposed into several simple (and regular) kernel functions (such as `vector_sum_with_scalar_multiply`).

Software-Centric Approach

After the algorithm refinement stage, the algorithm is compiled and profiled on a specific micro-processor. An embedded ARM8 core is chosen for the Maia design. As a software only implementation, the processor is chosen to run at 2.5V at 120MHz. The total energy, timing and energy percentage breakdown is shown in Figure 8.1. Based on the energy breakdown, some example kernels are identified and circled in Figure 8.1. Table 8.1 shows the most dominant kernels in this algorithm and the total percentage of each kernel.

Table 8-1 Kernel Percentage Breakdown

Functionality		Code Percentage
Kernels	<code>dot_product</code>	34.09 %
	<code>FIR_Filter</code>	19.54 %
	<code>IIR_Filter</code>	1.78 %
	<code>vector_sum_with_scalar_multiply</code>	11.24 %
	<code>Compute_Code</code>	10.83 %
	<code>Covariance_Matrix</code>	1.61 %
Program Control		20.91
Total		100 %

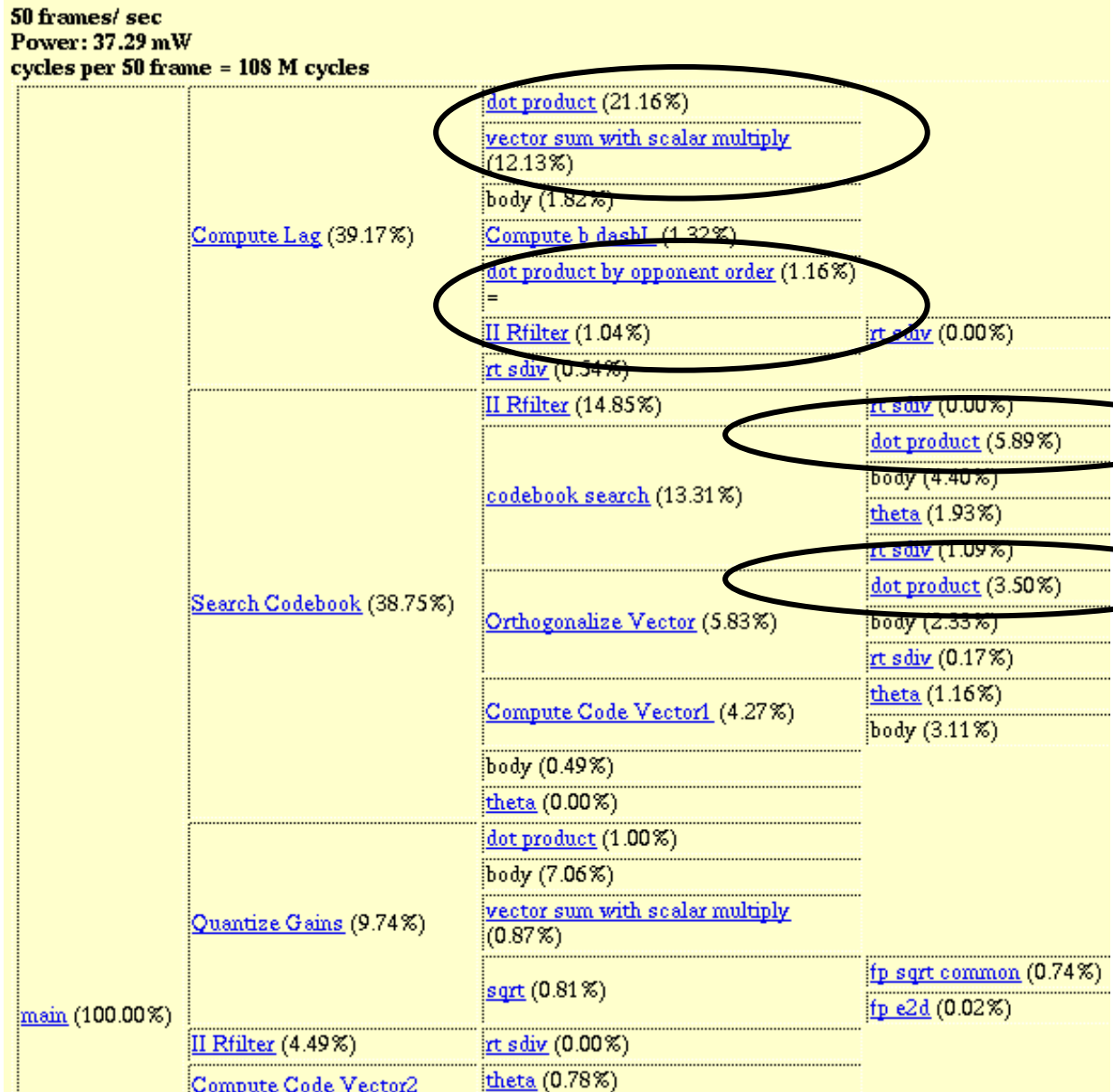


Figure 8-8-1 Total Timing and Energy and Energy Breakdown of the VSELP Algorithm Running on ARM8, Presented in Function Call–Tree Format

Kernel Mapping and Architecture Instantiation

All dominant kernels are mapped to satellite processors. Simple loops such as dot_product, FIR and vector_sum_scalar_multiply kernels have been manually mapped. The FPGA is used as a satellite for infrequent functions (theta function generation in the case of VSELP) that do not justify custom ASIC implementations. Other kernels are mapped using the synthesis tool (direct-mapping). The output of the mapping stage is a set of kernel mapping in C++IF. In addition, a spreadsheet macro-model for each model is generated. In this case study, the parameter for each kernel macro-model is the frequency count of each kernel loop. Figure 8-2 shows an example of a macro-model for the dot_product kernel.

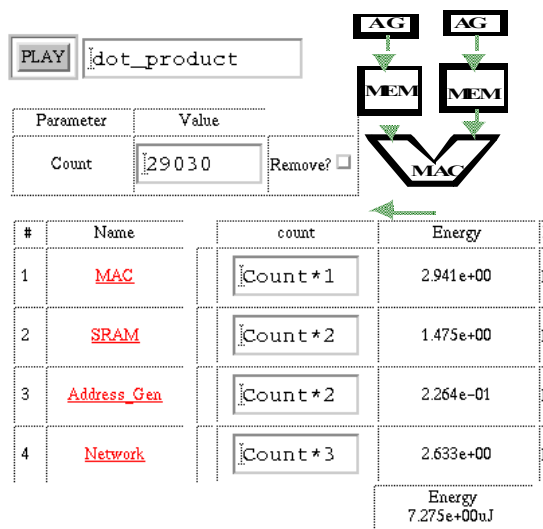


Figure 8-8-2 Kernel Energy Macro-model in Spreadsheet

Using the mappings of all the kernels, the partitioning stage is entered. The partitioning stage is very straightforward in this case. Approach two of the partitioning solution is used since the partition with minimum energy also meets the real-time constraint.

Based on partitioning result (show satellite count of each kernel), the Maia architecture is instantiated. The Maia processor (Figure. 8-2) combines a microprocessor core (embedded ARM8) with 21 satellite processors: two MACs, two ALUs, eight address generators, eight embedded memories (4 512×16bit, 4 1K×16bit), and an embedded low-energy FPGA [Var00].

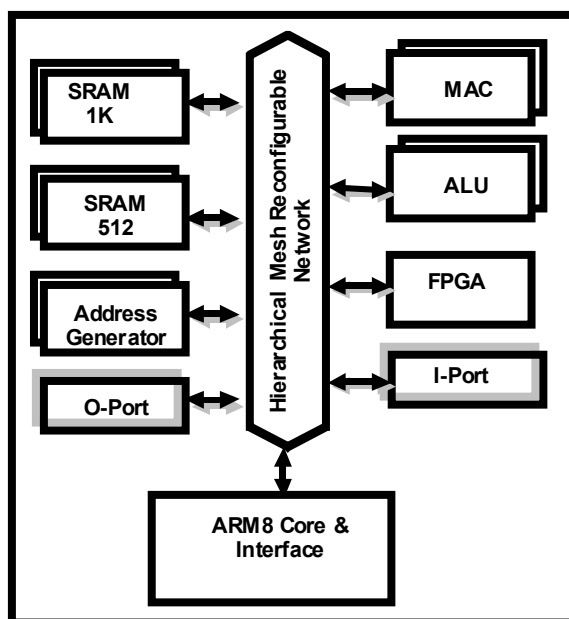


Figure 8-8-3 Components of the Maia Architecture

Reconfigurable Architecture Specific Optimizations

Based on the architecture instance given in the previous section, a floorplan of the satellites and processors is manually designed (shown in Figure 8-4). The next stage is to determine the kind of reconfigurable interconnection network to use. In Section 7.2, three kinds of reconfigurable interconnection network are considered for this system: multi-bus, mesh and hierarchical mesh. The conclusion is that the hierarchical mesh

interconnection can reduce the total satellite interconnection power by a factor 7. Therefore, the hierarchical mesh is chosen to be implemented on Maia.

The final stage involves generating configuration code for each kernel from the C++IF and iteratively optimizing the configuration code. At the end of Section 7.3, the optimizations used for the VSELP kernels configurations have been visited. It is shown that the configuration code can be cut down from 30M cycles/sec to 10M cycles/sec based on the evaluation environment developed in this thesis.

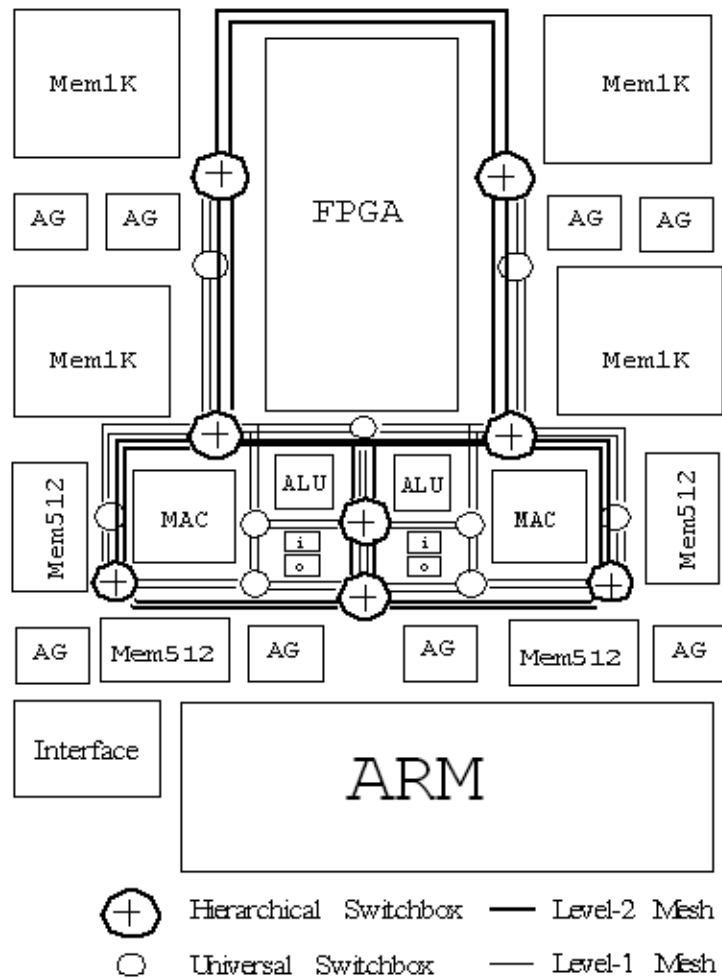


Figure 8-8-4 Maia Floorplan and Hierarchical Interconnect

Results

Figure 8.5 (a) shows the improvement in power consumption of the final implementation of VSELP running on Maia compared to an all-software implementation (ARM8 core). The initial implementation consumes around 37.29 mW, and the final power consumption is estimate to be around 1.79 mW. When compared to the best implementation reported in the literature [Lan98], the total energy efficiency is a factor of 8 better. The estimated energy dissipation of the processor when programmed for a VCELP voice coder (with 1.8mW total power consumption) is presented in Table 8.4, which also includes a breakdown of the energy over the major functions. Dominant kernels are directly mapped onto hardware satellites, and their run-time reconfiguration is performed by the ARM core. Therefore, the kernel energy presented in the table incorporate contributions from both satellite and ARM8 configuration. The program control part of the algorithm is completely mapped to the software.

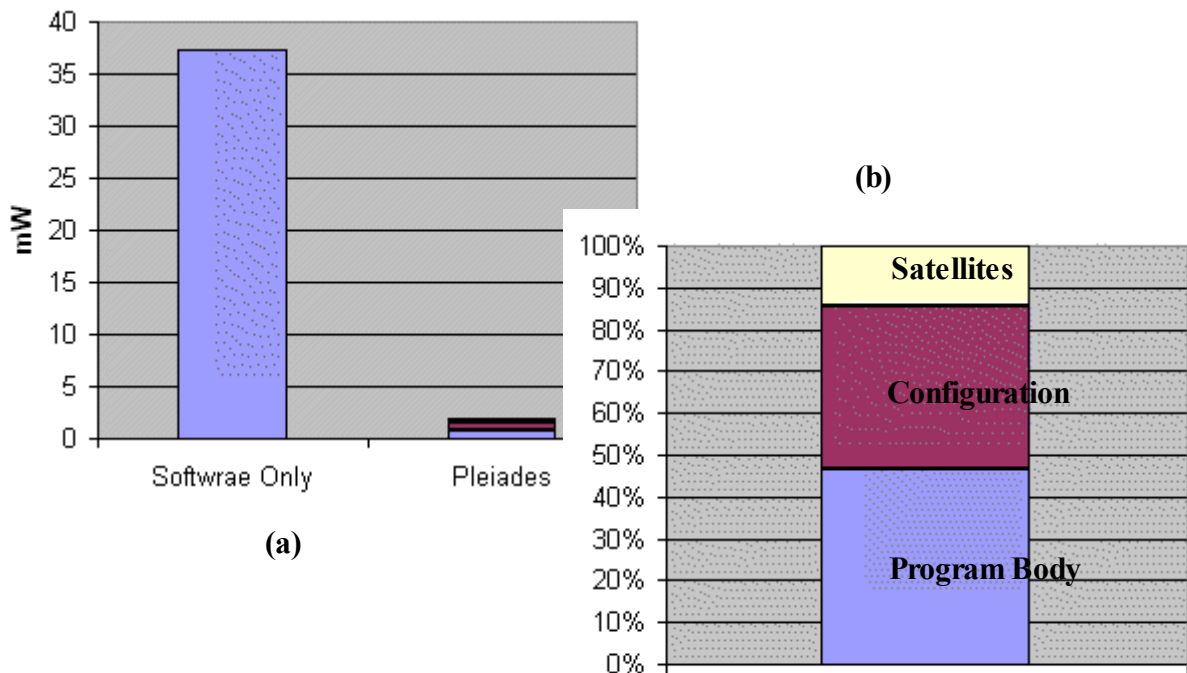


Figure 8-8-5 (a) Final Power Consumption of VSELP on Maia versus a All-Software Implementation (b) Breakdown of Power Consumption

Table 8-2 Estimated Total Energy of VCELP Running on Maia and Percentage Breakdown of Each Kernel

Functionality	Energy consumption (mJ) for 1 sec of VCELP speech processing
Dot product	0.738
FIR filter	0.131
IIR filter	0.021
Vector sum with scalar multiply	0.042
Compute code	0.011
Covariance matrix compute	0.006
Kernels	
Program control	0.838
Total	1.787

Table 8-3 Maia Chip Statistics

Technology	0.25 μ m 6-level metal CMOS
Main Supply Voltage	1 V
Additional Voltages	0.4 V, 1.5 V
Die Size	5.2 mm x 6.7 mm
Transistor Count	1.2 Million transistors
Average Cycle Speed	40 MHz
Average Power Dissipation	1.5 - 2 mW

Table 8.5 shows the chip statistics of Maia. The additional voltages are for the FPGA and low-swing interconnect. More detailed information on the circuit techniques used for the Maia architecture can be found in [Zha00][Zha01]. Figure 8.5 shows a die photograph of the Maia chip.

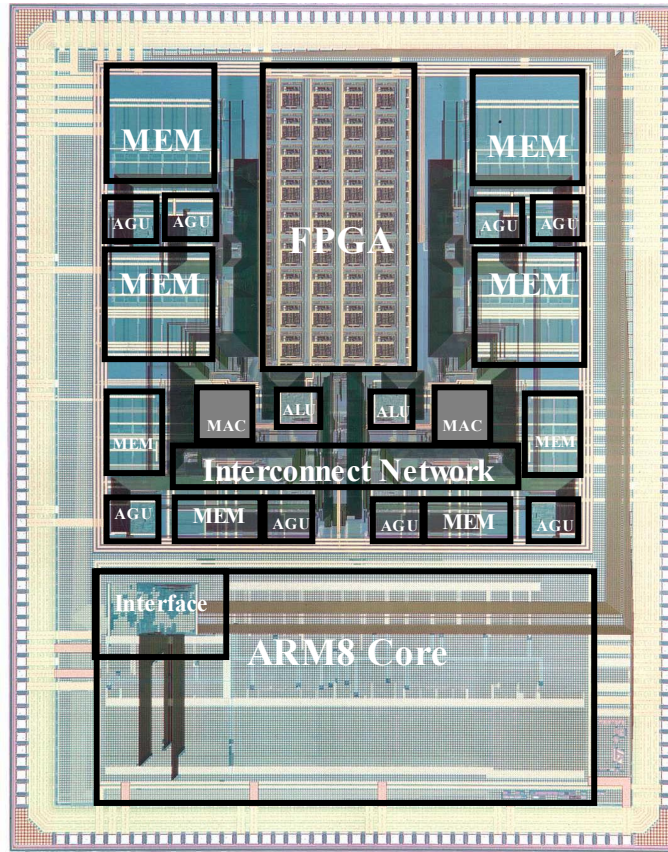


Figure 8-6 Die photo of the Maia chip

9 Conclusion and Future Research

To conclude this thesis, a brief summary of contributions is given in the beginning of this chapter. Research topics on heterogeneous architecture styles and design methodologies that are worthy of future investigations are listed as the final remarks of this thesis.

9.1 Summary of Contributions

This thesis presents a design methodology for low-power heterogeneous reconfigurable digital signal processors. Unlike most of the state-of-the-art system design environments that deal with only hardware and software, this design methodology takes into account platforms ranging from ASIC, micro-processor to fine-grained and coarse-grained reconfigurable processors. At the same time, while most design methodologies consider optimizing designs only for performance and area, the methodology in this thesis emphasizes the energy efficiency metric.

The design methodology has the same concept as the Y-chart approach [Kie98] that relies on characterizations of both the algorithm to be implemented and the architecture to be targeted to. In this thesis, efficient and effective tools and models for the algorithm and architecture characterizations are described. The combination of the top-down and bottom-up information allows the designs to be optimized at all levels and the overall system performance to be quickly evaluated. This is also an ideal tool to be utilized if the algorithm designers would like to obtain more information on the implementation costs or if the architecture designer would like to find out the effects of their architecture decisions.

The algorithm and architecture characterizations are used to guide the step-by-step procedure of the design methodology. The procedure starts out to extract kernels. In the next step, the kernels are mapped to different architectures and the implementation costs are recorded. A partitioning and satellite instantiation step follows the mapping step. The final stage of the methodology includes tools and methods to optimize the software and hardware of the resultant heterogeneous architecture. Two designs are given to show the effectiveness of the methodology.

As mentioned in chapter 3, this methodology is developed to serve as the core for both the architecture instantiation stage and the algorithm compilation stage. In the case of using the methodology to instantiate an architecture to support many algorithms, each algorithm should be carried all the way to the partitioning stage. Feeding the results to the satellite instantiation step gives us the desired architecture and the rest of the methodology can be used without change. In the case of compiling an algorithm to an existing architecture, all steps can be used (except for the optimizations for reconfigurable interconnect since the architecture is fixed). The designers have to perform some manual re-writing of the code and a resource constraint based mapping tools should be supplied to speedup this process (many research and tools exist to meet this requirement [VLIW compilation]).

9.2 Future Research Directions

There are many extensions to the research described in this thesis. The mainly come from two areas: heterogeneous reconfigurable architecture or design methodologies for heterogeneous architectures. They are listed in the following two sections.

9.2.1 Heterogeneous Reconfigurable Architecture

- The use of reconfigurable architecture is undoubtedly gaining an important role in the system-on-a-chip design platforms. Applying reconfigurable architecture to implement not only the dataflow intensive computations (as presented in this thesis) but also the control oriented computation (Layer 1, Layer 2 or Layer 3 software for network protocol processing) or data stream based computation (e.g. data routing, shuffling and interleaving) is a very promising approach.
- As shown in the design examples, the reconfigurable software still takes up a large portion of the resultant cycle/energy even though the optimizations used in this thesis can reduce the cost significantly. Part of the reason is that the configuration is done via the micro-processor. One potential optimization is to have a dedicated configuration code generator or DMA to take care of the configuration data movements.

9.2.2 Design Methodology for Heterogeneous Reconfigurable Architectures

Methodology and tools to assist in building these complex architectures still require more investigation in the following areas:

- Multithreaded system: The algorithm input to our methodology has a simple control flow (the main threads spawns off kernel processes, all kernel processes join back to the main threads at the same time). This is an essential problem to solve first because mapping algorithms with more complicated control flows can utilize the basic quantitative information from the simple case. For example, a task graph (similar to SDF in Ptolemy), using a simple thread (as used in our methodology) as a task, can be the highest level of system specification. Developing a methodology to map algorithms from a concurrent system specification to heterogeneous reconfigurable architectures should be addressed.

- Compilation and estimation tools for FPGA: Because of the uncertainty of the LUT placement and interconnect routing, it is often hard to predict the performance/energy/area of an algorithm implemented on FPGAs. However, to have a system design methodology that allows fast quantitative feedback of implementations. Good estimation models have to exist for all architectures. Therefore, better tools to map algorithms for fine-grained FPGA and obtain good estimation of the P-D-A metrics are needed.

9.2.3 Conclusions

Hardware or software, it is no longer the only choice of implementation for digital communication systems anymore. The design methodology described in this thesis helps the designers to look at one more dimension (reconfigurable architecture) in the architecture space. It also helps the designer to consider more carefully the energy metric, which is very important in the wireless designs. The design of system-on-a-chip will only be more complex, and the components more heterogeneous. It is important extent this research to incorporate more architecture styles, algorithm specifications or design metrics. Methodologies like this not only help in designing better systems, but also educate the algorithm designers to know about architectures and the architecture designers to understand the overall system.

10 Bibliography

[Abn01] A. Abnous, "Low Power Heterogeneous Reconfigurable Digital Signal Processor", *PhD Dissertation*.

[Bag97] Baganne, A., Philippe, J.L., Martin, E. (Edited by: Mukund, P.R.; Sridhar, R.; Gabara, T.; Carothers, J.D.) "Rapid prototyping of telecommunication systems on mixed HW/SW architectures" *Proceedings. Tenth Annual IEEE International ASIC Conference and Exhibit*, New York, NY, USA: IEEE, 1997. p.178-82.

[Chu89] Chu, C. M. and Potkonjak, M. and Thaler, M. and Rabaey, J. HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications. *Proceeding of the International Conference on Computer Design*, pages 432-435, 1989.

[Gue99] L. Guerra, J. Fitzner, D. Talukdar, C. Schlager, B. Tabbara, V. Zivojnovic, "Cycle and Phase Accurate DSP Modeling and Integration for HW/SW Co-Verification", *Design Automation Conference*, 1999.

[Gos] G. R. Goslin, "A Guide to Using Field Programmable Gate Arrays for Application-Specific Digital Signal Processing Performance", *Proceedings of SPIE*, vol. 2914, p321-331.

[Goe98] M. Goel and N. R. Shanbhag, "Low-power equalizers for 51.84 Mb/s very high-speed digital subscriber loop [VDSL] modems", *Proceedings of IEEE Workshop on Signal Processing Systems*, Oct. 1998, Boston.

[Dav97] B.P. Dave, G. Lakshminarayana and N. K. Jha, "COSYN: Hardware-Software Consynthesis of Embedded Systems", *Design Automation Conference*, pp.703-708, 1997.

[Dav99] Bharat P. Dave, "CRUISADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems", *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 1999.

[Dic98] Robert P. Dick and Niraj K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Digest of Technical Papers of 1998 IEEE/ACM International Conference on Computer-Aided Design*.

[Gal95] D. Galloway, "The transmorpher C Hardware Description Language and Compiler for FPGAs", in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machine*. pp. 136-144, April 1995.

[Hen99a] Tim Hentschel, Matthias Henker and Gerhard Fettweis, "The Digital Front-End of Software Radio Terminals", *IEEE Personal Communications*, pages 40-46, August 1999, Vol. 6 No. 4.

[Hen99b] J. Henkel, "A Methodology for Minimizing Power Dissipation of Embedded Systems through Hardware/Software Partitioning", *Proceedings of???*

[Kal95a] A. Kalavade and E. A. Lee, "Hardware/Software Co-design Using Ptolemy: A Case Study", p397-414, Codesign, *Computer-Aided Software/Hardware Engineering*, edited by J. Rozenblit and K. Buchenrieder

[Kal95a] A. Kalavade and E. A. Lee, "The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection", *Proceedings of???*

- [Kel94] W. Kelly and W. Pugh, "Selecting affine mapping based on performance estimation", *Parallel Processing Letters*, vol. 4, (no. 3), Parallelization Techniques for Uniform algorithms Workshop, Sept. 1994, p.205-19.
- [Lip91] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, and O.P. McArdle. "PHIDEO: a silicon compiler for high speed algorithms." In *Proceedings of the European Conference on Design Automation*, pages 436--441, Amsterdam, The Netherlands, February 1991.
- [Meh95] R. Mehra and J. Rabaey, "
- [Sch95] B. Schonert, J. Villasenor, S. Molloy and R. Jain, "Techniques for FPGA Implementation of Video Compression Systems", *International Symposium on Field-Programmable Gate Arrays*, 1995.
- [She01] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee, "FPGA Hardware Synthesis from MATLAB", *Proceedings of VLSI Design 2001*.
- [Var94] A. Varma and C. S. Raghavendra, "Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice", *IEEE Computer Society Press*, 1994.
- [Vis95] K.A. Vissers, G. Essink, P.H.J. van Gerwen, P.J.M. Janssen, O. Popp, E. Riddersma, W.J.M. Smits, and H.J.M. Veendrick, "Architecture and programming of two generations video signal processors," *Microprocessing and Microprogramming*, vol. 41, pp. 373-390, 1995.
- [Zha99] N. Zhang, A. Poon, D. Tse, R. Brodersen and S. Verdu, "Trade-offs of Performance and Single Chip Implementation of Indoor Wireless Multi-Access Receivers", 1999 IEEE Wireless Communications and Networking Conference. pp. 226-30 vol. 1.