Realizing Software Defined Radio – A Study in Designing Mobile Supercomputers

by

Yuan Lin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Scott A. Mahlke, Co-Chair
Professor Trevor N. Mudge, Co-Chair
Professor Marios C. Papaefthymiou
Associate Professor Dennis M. Sylvester
Professor Chaitali Chakrabarti, Arizona State University

ABSTRACT

Realizing Software Defined Radio – A Study in Designing Mobile Supercomputers

by

Yuan Lin

Co-Chairs: Scott A. Mahlke and Trevor N. Mudge

The physical layer of most wireless protocols is traditionally implemented in custom hardware to satisfy the heavy computational requirements while keeping power consumption to a minimum. These implementations are time consuming to design and difficult to verify. A programmable hardware platform capable of supporting software implementations of the physical layer, or Software Defined Radio (SDR), has a number of advantages. These include support for multiple protocols, faster time-to-market, higher chip volumes, and support for late implementation changes. The challenge is to achieve this under the power budget of a mobile device. Wireless communications belong to an emerging class of applications with the processing requirements of a supercomputer but the power constraints of a mobile device – *mobile supercomputing*.

This thesis presents a set of design proposals for building a programmable wireless communication solution. In order to design a solution that can meet the lofty requirements of SDR, this thesis takes an application-centric design approach – evaluate and

optimize all aspects of the design based on the characteristics of wireless communication protocols. This includes a DSP processor architecture optimized for wireless baseband processing, wireless algorithm optimizations, and language and compilation tool support for the algorithm software and the processor hardware. This thesis first analyzes the software characteristics of SDR. Based on the analysis, this thesis proposes the Signal-Processing On-Demand Architecture (SODA), a fully programmable multi-core architecture that can support the computation requirements of third generation wireless protocols, while operating within the power budget of a mobile device. This thesis then presents wireless algorithm implementations and optimizations for the SODA processor architecture. A signal processing language extension (SPEX) is proposed to help the software development efforts of wireless communication protocols on SODA-like multi-core architecture. And finally, the SPIR compiler is proposed to automatically map SPEX code onto the multi-core processor hardware.

To my Shepherd, mom, dad, and Jen

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God – Father, Son and Spirit. Without His love and guidance, I would be lost. I am grateful that He has led me to complete my graduate study at the University of Michigan. Hindsight is always 20/20. I see now through all the difficulties that He is there with me, guiding me and loving me.

I would like to thank both of my advisors, Prof. Trevor Mudge and Prof. Scott Mahlke, for their guidance and support. Without them, this thesis would not have be possible. Both have been excellent mentors to me. It is a great privilege to have worked with both of them. They have provided me with invaluable guidance, support, and opportunities to succeed. I would like to thank Prof. Chaitali Chakrabarti for being my "unofficial PhD advisor". It has been a pleasure working with you. Your helpful insights have made a great impact on my graduate work. I would also like to thank the other members of my dissertation committee, Prof. Marios Papaefthymiou and Prof. Dennis Sylvester. Thank you both for your time and valuable comments.

The work presented in this thesis is a collaboration between many graduate students. It would not have been possible without the help of my colleagues. I would like to thank Hyunseok Lee for his invaluable expertise on wireless communication protocols. Mark Woh and Yoav Harel both have helped me tremendously with the SODA processor's power analysis. They also have contributed greatly in the design of the SODA processor architecture. Yoonseo Choi was responsible for constructing parts of the compiler system.

Without the help of my colleagues, I would have struggled much longer in my PhD journey.

I also had the chance to met many of my fellow graduate students. They made my graduate experiences much more fun and rewarding. I would like to thank Zaher Andraus, Geoff Blake, Jason Blome, Yoonseo Choi, Mike Chu, Nate Clark, Ganesh Dasika, Kevin Fan, Shuguang Feng, Shantanu Gupta, Jeff Hao, Yoav Harel, Amir Hormati, Taeho Kgil, Manjunath Kudlur, Hyunseok Lee, Mojtaba Mehrara, Robert Mullenix, Hyunchul Park, Dave Roberts, Sangwon Seo, Mark Woh, and Hongtao Zhong. I have become good friends with many of them, and I will cherish their friendships for the rest of my life.

Finally, I would like to thank my family. Mom and dad, thank you so much for all of the love and sacrifice that you have done for me. Without you, I won't even get the chance to write this thesis. I am eternally indebted to you. My wife Jennifer Wang, you are a ray of sunshine in my life. I consider myself truly blessed to have married you. You are my best friend and best supporter for the past six years. I can't even begin to imagine my life without you. You are the reason that I am able to finish this PhD thesis. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Untethered digital devices are already ubiquitous. The world has over 3.3 billion active cell phones [8], each a sophisticated multiprocessor. With worldwide wireless semiconductor revenue totaled $24.3 billion in 2005, mobile terminals have arguably become one of the dominant computing platforms. We expect to see both the types and numbers of mobile digital devices increase in the near future. New technologies will improve the mobile phone by incorporating advanced multi-media functionalities. They will also improve the laptop by shrinking the form factor and increasing its battery life. These trends have blurred the line between traditional desktop computing and mobile cellular phones. We are in an era where users are no longer satisfied with computing powers that are confined to their homes or offices. Instead, users want to bring computing powers to wherever they are and whenever they want. To achieve this, we require software applications with the extraordinary computational requirement of a supercomputer running on the power budget of a mobile device – *mobile supercomputers* [12].

Software Defined Radio (SDR) is one of these mobile supercomputing applications. It promises to deliver a cost effective and flexible mobile communication solution by implementing the wide variety of the wireless protocols in software. The operation throughput

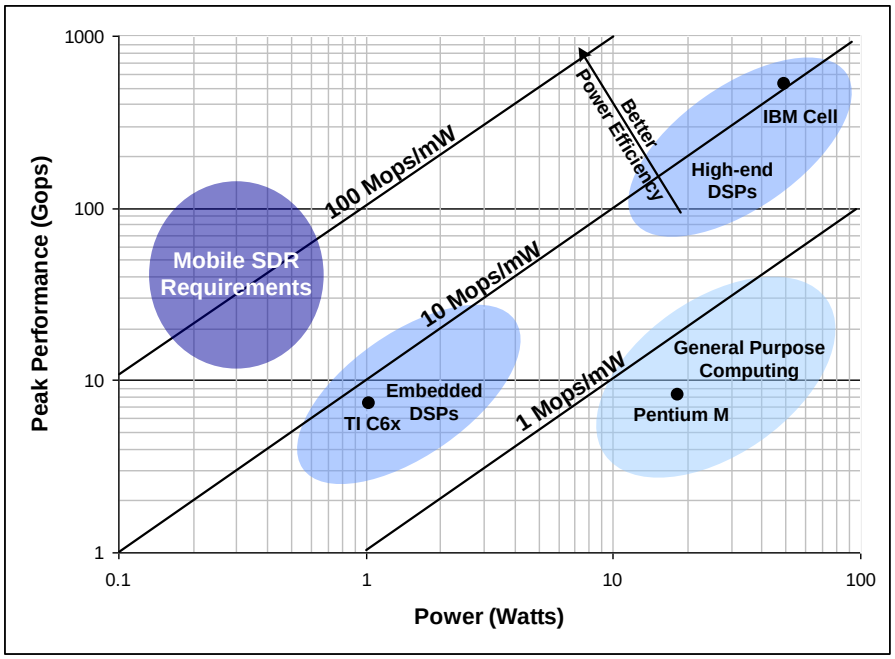


Figure 1.1: Throughput and power requirements of typical 3G wireless protocols. The results are calculated for 16-bit fixed point operations.

requirements of current third-generation (3G) wireless protocols are already an order of magnitude higher than the capabilities of modern DSP processors. This gap is likely to grow in the future. Figure 1 shows the computation and power demands of a typical 3G wireless protocol. Although most DSP processors operate at an efficiency of approximately 10 million operations per second (Mops) per milliwatt (mW), the typical wireless protocol requires 100 Mops/mW. Hence, most wireless protocols to date have been implemented with custom hardware. Although custom hardware can meet the operational requirements, a programmable solution offers many potential advantages:

- A programmable architecture would allow multimode operation, running different protocols depending on the available wireless network-GSM in Europe, CDMA in the USA and some parts of Asia, and 802.11 in coffee shops. This is possible with less hardware than custom implementations require.
- A protocol implementation's time to market would be shorter because it could reuse the hardware. The hardware integration and software development tasks could progress in parallel.

- Prototyping and bug fixes would be possible for next-generation protocols on existing silicon through software changes. The use of a programmable solution would support the specification's continuing evolution; after the chipset's manufacture, developers could deploy algorithmic improvements by changing the software without redesign.
- Chip volumes would be higher because the same chip could support multiple protocols without requiring hardware changes.

1.1 Contribution

In the foreseeable future, it is likely that many mobile communication devices are going to be supported by SDR technology. This thesis presents a set of design proposals for realizing a programmable wireless protocol implementation. In order to design a solution that can meet the lofty requirements of SDR, this thesis takes an application-centric approach – evaluate and optimize all aspects of the design based on the characteristics of wireless communication protocols. Because SDR is an interdisciplinary research topic, this thesis examines multiple research subjects under the overall objective of realizing SDR: computer architecture, DSP algorithm optimizations, programming language design, and compiler construction. We must first understand the workings of wireless protocols and their algorithms. A DSP processor is then designed and optimized for wireless communication algorithms. These wireless algorithms must also be optimized for the DSP architecture. Language and compilation support must be provided to bridge the gap between the programmers and the hardware. This thesis makes the following contributions:

- A programmable multiprocessor architecture, SODA, for supporting third generation wireless protocols within the power budget of a mobile device.
- Design and implementation of wireless protocol's DSP algorithms for SODA.
- A comparison study between the SODA processor and the Ardbeg processor. The Ardbeg DSP processor is a commercial prototype based on the SODA architecture.

- A programming language extension, SPEX, for describing wireless protocols.
- A proposed multiprocessor compiler, SPIR, for the Ardbeg processor.

Wireless Protocol Analysis. Wireless protocols are collections of disparate DSP algorithm kernels working together as one system. It requires both the implementation of each algorithm as well as the construction of the entire system with the algorithms as building blocks. The DSP algorithms consist mostly of long vector arithmetic operations. The system consists of streaming computation where data are processed sequentially through a pipeline of DSP functions.

Software protocol processing provides many advantages over hard-wired solutions. However, the performance requirements for current generation wireless protocols are an order of magnitude higher than the capabilities of modern general purpose and DSP processors. This thesis chooses the W-CDMA wireless protocol as our case study. Workload profiling shows that the 2Mbps W-CDMA baseband processing requires the computational power of approximately seven Pentium 4 processors. In addition, a mobile SDR processor must run on the power budget of a mobile terminal. A typical mobile device allocates around 0.5 Watt for baseband processing, whereas typical general purpose processors consume over 20 Watts.

Processor Design. This thesis proposes a multi-core DSP architecture, SODA, for supporting SDR. SODA consists of one control processor, four data processors, and a shared global memory. The control processor is an embedded general purpose processor that is capable of handling the control-intensive code that is used to manage the overall baseband processing system. The data processors are specialized DSP processors that can perform data-intensive computations.

Because the biggest challenge is meeting the computation requirements while operating within the embedded power envelope, the focus is on designing a power-efficient data

processor. Therefore, we picked an existing low-power embedded processor, ARM Cortex M-3, as SODA's control processor. The design of the SODA data processor is motivated by the observation that the majority of the computation are long vector arithmetic operations. Previous researches have shown that a Single Instruction Multiple Data (SIMD) architecture is a good fit for vector-based computations. However, most existing SIMD-based processors operate on relatively short 4 to 8 element vectors, due to intra-vector data rearrangement difficulties in general purpose computations. Because the SODA data processor is targeted only at the set of DSP algorithms for wireless communication, the data rearrangement issue can be handled efficiently through a specialized vector permutation network. Analysis shows that a wide SIMD datapath that supports 32 element vectors is the most power efficient for wireless baseband processing algorithms.

A commercial SDR processor based on the SODA processor architecture has been developed by ARM Ltd. The Ardbeg processor is also a multi-core DSP processor that consists of 32-lane SIMD data processors. This thesis provides a detailed comparison study between the SODA and Ardbeg processors. This study reconfirms many of the SODA architectural decisions. It also reveals many design shortcomings of SODA, and explains the subsequent design improvements in Ardbeg.

Algorithm Implementations. Each DSP algorithm in W-CDMA is hand coded and optimized for the SODA data processor. The majority of wireless protocols' algorithms operate on large vectors, and are therefore a good fit for a wide-SIMD design. This thesis validates this claim by demonstrating the implementation of key DSP algorithms on SODA. In addition, DSP algorithms usually have multiple different implementations, not all of which can be mapped efficiently onto the wide-SIMD design. This thesis describes a set of DSP algorithm implementations that are suited for the SODA architecture.

Language and Compiler Support. This thesis also proposes a programming lan-

guage and compilation flow for mapping wireless protocols onto SODA-like multi-core DSP architectures. Motivated by the streaming computation of the DSP systems, previous works have proposed using concurrent dataflow models to describe DSP systems. The majority of the compilation research focused on the static dataflow model due to its simplicity and determinism. Although wireless protocols have streaming properties that match the dataflow model, they cannot be described with a static dataflow model. In between long episodes of streaming computation, DSP systems intermittently reconfigure the streaming patterns to account for changes from the users and the environment. This thesis finds that a reconfigurable dataflow model, parameterized dataflow, is better suited for describing wireless protocols.

The SPIR compiler is a function-level compiler, which means that the granularity of an atomic execution unit is a function, not an instruction. Traditional compiler's intermediate representation (IR) is used to model instruction-level interactions. A different IR is needed to model the inter-function behavior. This thesis proposes using the parameterized dataflow model as compiler's intermediate representation. The proposed high-level programming language, SPEX, is a language extension for C. Its purpose is to serve as a guideline for programmers to write stylized C code that can be translated into the parameterized dataflow model. SPIR compiler's backend performs optimization on the dataflow IR and generates multi-threaded C code for the Ardbeg processor.

1.2 Organization

The remainder of this dissertation proposal is organized as follows. Chapter 2 provides our analysis on the software characteristics of our SDR case study – the W-CDMA wireless protocol. In Chapter 3, this thesis proposes the Signal-Processing On-Demand Architec-

ture (SODA), a fully programmable architecture that supports SDR. In Chapter 4, this thesis then shows our SDR algorithm implementations on SODA. In Chapter 5, this thesis presents a comparison study between SODA and its subsequent commercial prototype – Ardbeg. In Chapter 6, Signal Processing language EXtensions (SPEX) are proposed . And finally, in Chapter 7, an multiprocessor compiler is described to automatically parallelize SPEX code onto the Ardbeg processor.

Ultimately, we believe that the need to support many increasingly complex wireless protocols will make the use of programmable systems for these protocols inevitable. And the techniques proposed in this thesis are relevant in designing viable solutions for SDR and other mobile supercomputing applications.

1.3 Acknowledgements

The work presented in this thesis is a collaboration between three professors and eight graduate students. The author of this thesis has been the student leader on all fronts of this project, but other people have made many key contributions to the projects as well. The wireless protocol analysis was led by Hyunseok Lee. Other contributors included the author of this thesis, Yoav Harel and Mark Woh. The architecture study was led by the author of this thesis, with helps from Hyunseok Lee, Mark Woh, and Yoav Harel. Yoav Harel was involved in the architectural design. Hyunseok Lee helped on the wireless protocol analysis and benchmarking. And Mark Woh was responsible for the SODA power analysis. The comparison study between SODA and Ardbeg is done as a collaboration between Mark Woh, the author of this thesis, and Sangwon Seo. The SPEX language extension is proposed solely by the author of this thesis. There are other contributors to the SPIR compiler. They include Yoonseo Choi and Manjunath Kudlur. Yoonseo Choi

was responsible for the Ardbeg code generation. And Manjunath Kudlur developed an optimal software pipelining algorithm as a part of the compiler optimization.

CHAPTER 2

The W-CDMA Wireless Communication Protocol

The goal of this study is to design a programmable solution for wireless communication protocols. The first step of this process is to develop a deep understanding of the underlying requirements and computation characteristics of wireless protocols. The majority of the computation occurs at the physical layer of protocols, where the focus is the signal processing. Traditionally, kernels corresponding to the dominant tasks, such as filters and decoding, are identified. Design alternatives are then evaluated on the subset workload. This approach has the advantage of dealing with a small amount of code. However, we have found that the interaction between tasks in SDR has a significant impact on the hardware architecture. This occurs because the physical layer is a combination of algorithms with different complexity and processing time requirements. For example, high computation tasks that run for a long period of time can often be disturbed by small tasks. Further, these small tasks have hard real-time deadlines, thus they must be given high priority. We believe it is necessary to explore the whole physical layer operation with a complete model.

Among many wireless protocols, we select the wideband code division multiple access (W-CDMA) protocol as a representative wireless workload for study. W-CDMA system

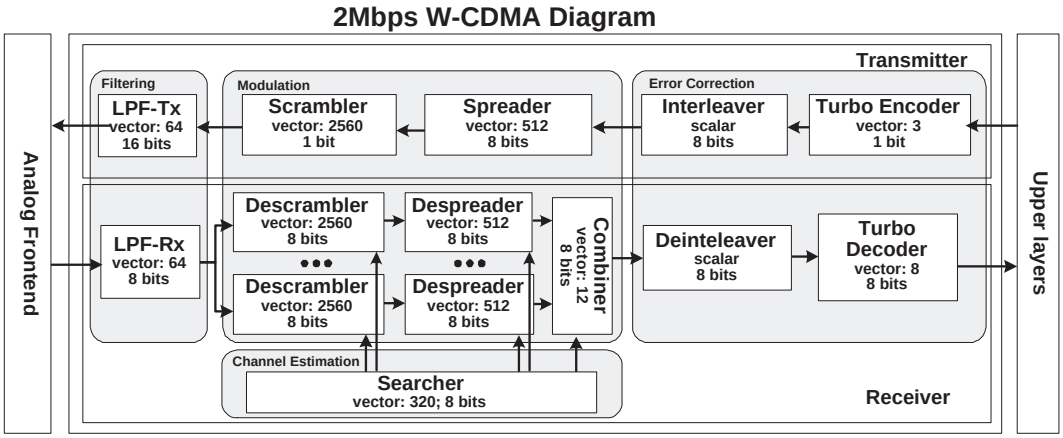


Figure 2.1: Physical layer operation of W-CDMA wireless protocol. Each block includes the algorithm's name, vector or scalar computation, vector width, and the data precision. The algorithms are also grouped into four categories, shown in shaded boxes: filtering, modulation, channel estimation, and error correction.

is one of the dominant third generation wireless communication networks where the goal is multimedia service including video telephony on a wireless link [41]. W-CDMA improves over prior cellular protocols by increasing the data rate from 64 Kbps to 2 Mbps. Additionally, W-CDMA unifies a single service link for both voice and packet data, compared with supporting only one service in previous generations. We have developed a full C implementation of the W-CDMA physical layer to serve as the basis for our study. The implementation can be executed on a Linux workstation and thus studied with conventional architectural tools. In this chapter, Section 2.1 provides a summary of the computing characteristics of the W-CDMA physical layer.

2.1 Protocol Overview

The protocol stack of the W-CDMA system consists of several layers. Each protocol layer provides a specific function in the system. For example the physical layer placed at the bottom of protocol stack is responsible for overcoming errors induced by an unreliable

wireless link, and the medium access control (MAC) layer resolves contention on the shared radio resources. In this section we discuss the computation model of the W-CDMA physical layer.

Due to the high computation demand and tight power budget, the physical layer in most wireless protocols is implemented in ASICs. Although SDR encompasses all protocols layers, this thesis only focuses on the physical layer due to its computation and power importance. The operation of physical layer utilizes both digital and analog circuits. Because the operation frequency of analog circuits such as low noise amplifier and mixers is over GHz level, it is infeasible to achieve programmability with current digital circuit technology. Thus, this thesis narrows down our focus on the physical layer operation performed by digital circuits. Figure 1 shows a high level block diagram of W-CDMA physical layer implemented by digital circuits. It is placed between upper layer protocols and the front-end circuit. The upper layer protocols are implemented on a general purpose processor due to their relatively low computation requirements. The front-end circuit is realized by analog circuit technology.

The operation flow of the W-CDMA physical layer is shown in Figure 2.1. It contains a set of disparate DSP algorithm kernels that work together as one system. There are four major components: filtering, modulation, channel estimation, and error correction. Filtering algorithms are used to suppress signals transmitted outside of the allowed frequency band so that interference with other frequency bands are minimized. Modulation algorithms map source information onto the signal waveforms of the transmitter, and receivers demodulate the signal waveforms back into source information. Channel estimation algorithms calculate the channel conditions to synchronize the two communicating terminals to ensure lock-step communication between the sender and the receiver. Error correction algorithms are used to combat noisy channel conditions. The sender encodes

Algorithms	Configurations	Vector Comp.	Vector Length	Bit Width	Comp. Mcycles/sec
W-CDMA (2Mbps)					
Scrambler	Defined in W-CDMA standard	yes	2560	1,1	240
Descrambler*	12 fingers, 3 base stations	yes	2560	1,8	2,600
Spreader	Spreading factor = 4	yes	512	8	300
Despreader*	12 fingers, 3 base stations	yes	512	8	3,600
PN Code (Rx)	3 base stations	no	1	8	30
PN Code (Tx)	Defined in W-CDMA standard	no	1	8	10
Combiner*	2Mbps data rate	partial	12	8	100
FIR (Tx)	4 filters x 65 coeff x 3.84Msps	yes	64	1,16	7,900
FIR (Rx)	2 filters x 65 coeff x 7.68Msps	yes	64	8,8	3,900
Searcher*	3 base stations, 320 windows	no	320	1,8	26,500
Interleaver	1 frame	no	1	8	10
Deinterleaver	1 frame	partial	1	8	10
Turbo Enc.	K=4	yes	3	1,1	100
Turbo Dec.*	K=4, 5 iterations	yes	8	8,8	17,500
*These algorithms have dynamically changing workloads that are dependent on channel conditions					

Figure 2.2: Workload analysis result of W-CDMA physical layer processing. "Vector comp" indicates whether the algorithm contains vector-based arithmetic operations. "Vector width" lists the native computation vector width. "Bit width" lists the data precision width. "Comp Mcycle/sec" lists the cycle-count of running the algorithm on a general purpose processor.

the original data sequence with a coding scheme that inserts systematic redundancies into the output, which is decoded by the receiver to find the most likely original data sequence.

2.2 Workload Analysis

Workload Profiling. Figure 4.1 shows the result of our profiling. The first column lists the W-CDMA algorithms that have been implemented as a part of this study. The second column lists the corresponding configurations for each of the algorithms. The third and fourth column lists the vector computation information for the algorithms. The fifth column lists the data precision width.

The last column shows peak workload of the algorithms. The peak workload is the minimum performance needed to sustain 2Mbps throughput, under the worst wireless channel condition. For example we need a processor with approximately 8 GOPS in order to finish the transmitter FIR task within 0.67 msec. For peak workload analysis, we compiled our W-CDMA model with an Alpha gcc compiler, and executed on M5 architectural simulator [16]. We measure the instruction count that is required to finish each algorithm. Results are calculated by dividing the instruction count by the maximum processing time of each algorithm. The workloads of Viterbi and Turbo decoder requires further verification because their processing times are not fixed. The data are calculated under the assumption that the processing time of Viterbi decoder is 40 msec and that of Turbo is 25 msec.

The results show that there are a set of key DSP algorithms that are responsible for the majority of the computation. These algorithms include the FIR filter, searcher, Turbo decoder, descrambler and despreader. Therefore, a SDR processor must process these algorithms efficiently.

Parallelism in the Protocol. To meet the real-time W-CDMA performance requirement in software, we must exploit inherent algorithmic parallelism. Figure 4.1 columns 3 and 4 show the potential parallelism that can be exploited either through Data Level Parallelism (DLP) or Thread Level Parallelism (TLP). We define DLP as the maximum SIMD variable vector width. The first column represents maximum possible DLP through the maximum number of elements in a vector. The width of element in a vector is shown at the second column. Because a vector operation needs two operands, we represent the element width of each vector separately. We define TLP as the maximum number of different threads that can be executed in parallel.

From this result, we can see that searcher, filter, scrambler, and descrambler contain

a lot of vector parallelism due to intensive vector operation. In addition, we can expect tasks level parallelism from them. For searcher operation we can issue 5120 tasks concurrently. For the case of scrambler and descrambler we can expect tasks level parallelism by bisecting a wide vector into smaller ones. Although sliced vectors are not perfectly uncorrelated, we can execute the smaller vector operations with negligible dependency. At the practical view point, too wide vector is implausible. However turbo decoder, which is one of dominant workloads, contains limited vector and task level parallelism. The vector width of turbo decoder is 8.

Intrinsic Computations. Many DSP algorithms have a large number of multiplication operations. Because multiplication is a power consuming operation, it is advantageous to convert this into other operations. First, the multiplications in the spreader and scrambler can be simplified to an exclusive OR, because both operands are either 1 or -1. Second, the multiplication operations in the searcher, descrambler, despreader, and FIR(Tx) can be simplified into conditional complement operations, because one operand of the multiplications in these algorithms is either 1 or -1, and the other operand is a fixed point number. However, the multiplication of the FIR(Rx) cannot be simplified because both operands are fixed point numbers.

Vector permutations are required for the Turbo decoder, FIR, and searcher, because either output or operand vector needs to be permuted. In Turbo decoder, the core computation operation is the Add-Compare-Select operation, which consists of one vector addition, one vector comparison, one vector permutation, and one vector move operation.

Memory Requirements. Figure 2.3 lists the memory size and throughput requirements for W-CDMA algorithms. Memory usage is divided into data and instruction memory access. Data memory access is further divided into input buffer, output buffer, and scratchpad memories. Input and output buffers are used for IO memory accesses, and

Algorithms	Configurations	Data Memory						Inst. Mem.
		Input Buffer		Output Buffer		Scratchpad		
		KB	MBps	KB	MBps	KB	MBps	
W-CDMA (2Mbps)								
Scrambler	Defined in W-CDMA standard	0.7	15.4	0.7	15.4	0.7	15.4	0.5
Descrambler*	12 fingers, 3 base stations	5.6	123.2	5.6	123.2	0.7	15.4	0.5
Spreader	Spreading factor = 4	0.1	1.9	0.4	7.6	0.1	3.9	0.4
Despreader*	12 fingers, 3 base stations	0.4	7.6	0.1	1.9	0.1	3.9	0.3
Combiner*	2Mbps data rate	0.1	0.1	0.1	0.1	0.1	0.1	0.1
FIR (Tx)	4 filters x 65 coeff x 3.84Msps	0.3	7.6	10.3	245.8	0.1	1996.8	0.2
FIR (Rx)	2 filters x 65 coeff x 7.68Msps	10.5	245.8	2.5	61.4	0.1	1996.8	0.2
Searcher*	3 base stations, 320 windows	20.8	2.1	0.1	0.1	32.0	2654.3	3.1
Interleaver	1 frame	1.2	1.1	1.2	1.1	9.5	1.9	0.1
Deinterleaver	1 frame	26.1	5.2	26.1	5.2	8.7	5.2	0.1
Turbo Enc.	K=4	2.6	4.0	7.8	12.0	0.1	2.0	1.6
Turbo Dec.*	K=4, 5 iterations	61.5	96.0	2.6	4.0	6.4	25600.0	3.4
*These algorithms have dynamically changing workloads that are dependent on channel conditions								

Figure 2.3: Memory requirements for the W-CDMA physical layer algorithms. "KB" is the memory size requirement in KByte. "MBps" is the memory through requirement in KByte-per-second. "Input buffer and output buffer" are the IO memory requirements. "Scratchpad" is the internal memory requirement. As shown in the figure, the overall memory size and throughput requirements for W-CDMA is not very high. Majority of which come from scratchpad memory access of intermediate computation results.

the scratchpad memory is used for storing intermediate computation results. As shown in the figure, the overall memory size and throughput requirements for W-CDMA are not very high. Majority of which come from scratchpad memory access. Most algorithms are streaming DSP algorithms, where each input data is consumed once in a sequential order, and each corresponding output data is produced in the same sequential order. Streaming DSP algorithms do not need to buffer data, which result in smaller memory requirements. The exceptions are the Turbo decoder, searcher, interleaver and deinterleaver – all require their input data to be buffered. Interleaver and deinterleaver do not process their input data in a sequential order. Turbo decoder and searcher process input data multiple times. This is the reason behind the relatively high memory requirements for these algorithms.

Power Budget. [62] has presented an overall evaluation of cellular phones as embed-

ded systems. It has outlined the power budget for the various components in a cellular phone. For W-CDMA physical layer processing, the power budget is typically around 300mW. This varies for difference wireless protocols and mobile devices.

2.3 Summary

W-CDMA wireless protocol has very high performance requirement that goes beyond general purpose desktop processors, while also has sub-watt power budget of a typical mobile terminal. The algorithms have abundance of data-level parallelism, with the majority of the computations being long vector arithmetic operations. The vector arithmetic operations are dominated by 8- to 16-bit fixed point addition/subtraction operations, with some additional multiplication operations. Fixed point divide operations are rarely used, and floating point arithmetic operations are not required. The algorithms also have relatively small memory footprints. The majority of which are used as scratchpad memory for holding intermediate computation results. The IO memory throughput between the algorithms is very low. In addition to W-CDMA, we have also examined several other wireless protocols, including 802.11a [1] and 4G [39], and find these characteristics to be common across all of the protocols. The results of these studies are omitted in this thesis. For more information, please refer to [52], [56] and [84]. A viable SDR processor must exploit these common protocol characteristics in order to meet both the power and performance requirements.

CHAPTER 3

SODA: A DSP Architecture For SDR

3.1 Introduction

The proposed programmable architecture, SODA, can meet the extraordinarily high performance requirements of current and future wireless protocols, use reasonable hardware area, and operate within an embedded DSP processor's power budget. The architecture is made up of four cores, each containing asymmetric dual pipelines that support scalar and 32-wide SIMD execution. The arithmetic units are customized for 16 bits, and the register files and software-controlled scratchpad memories need only a few ports. Our results show that in a 90nm implementation, our architecture meets the throughput requirements of the 2Mbps W-CDMA protocol and 24Mbps 802.11a running at only 400MHz. The area requirement is projected to be $6.7mm^2$. At the nominal operating voltage of 1V, this translates into a power consumption of less than 500mW. This number includes an ARM Cortex-M3 [2] control processor which is responsible for part of the protocol processing.

The main contributions of SODA are the following:

1. A design study of a fully programmable wide-SIMD architecture, SODA, that can meet the power and performance requirements of high-end wireless protocols. The

discussion outlines the architectural design choices and trade-offs required for an application-domain specific multiprocessor architecture for SDR.

2. An evaluation of the wireless algorithms on the proposed architecture which shows that high-end embedded programmable systems can meet the wireless protocols' throughput requirements.

3.2 W-CDMA Analysis Overview

The majority of previous architectural studies on DSP applications have focused mainly on small DSP kernels in existing benchmark suites, such as MediaBench [48]. Such an approach cannot be used here since wireless protocols have complex inter-algorithm interactions that cannot be characterized by studying individual algorithms in isolation. Therefore, we have implemented the complete W-CDMA physical layer in C to study the behavior of wireless protocol operations, as explained in [52]. Through the implementation of this protocol, we have found system-level challenges that have not been addressed in the literature, as well as many algorithmic-level implementation details that could not have been discovered through compiler analysis. Here we summarize our key observations into two categories: protocol system-level behavior and DSP algorithm-level behavior.

3.2.1 System-level Behavior

DSP Kernel Macro-Pipelining – Wireless protocols usually consist of multiple DSP algorithm kernels connected together in feed-forward pipelines. Data are streamed through kernels sequentially. With no data temporal locality, cache structures provide little additional benefits, in terms of power and performance, over software controlled scratchpad memories.

Low-throughput Inter-kernel Communication Traffic – Between DSP algorithm kernels, data are transferred as scalar variables. The traffic throughput of the protocol systems

are not very high (as in 7.68MBps for W-CDMA receiving front-end). This implies that inter-kernel data traffic can be mapped onto low-throughput, low-power inter-connects with minimal performance degradation.

Heterogeneous Inter-kernel Communications – Some inter-kernel communications can be streamed, where the receiving kernel can process input data individually (i.e. filters). Other inter-kernel communications must be buffered, as the receiving kernels require blocks of the data (i.e. Interleaver and Turbo Decoder). Kernels with the same throughput, but different communication patterns will result in dramatically different hardware requirements. Streamed kernels need only small FIFO queues, but the buffered kernels require a large memory space.

Real-time Deadlines – W-CDMA has multiple periodic deadlines. Meeting these deadlines is one of the challenges that has not been addressed in previous published DSP architectural studies. Meeting real-time deadlines requires concurrent execution management for multiple DSP algorithms.

3.2.2 Algorithm-level Behavior

High Data-Level Parallelism – Most of the computationally intensive DSP algorithms have abundant data level parallelism. For example, the searcher, the heaviest workload of the W-CDMA protocol, can be represented by 320-wide vectors.

8 to 16bit Data Width – Most algorithms operate on variables with small values. Our analysis of W-CDMA and 802.11a suggests that the architecture should provide strong support for 8 and 16 bit fixed point operations. 32 bit fixed point and floating point support is not necessary.

Scalar-Vector operations – Because data are transferred as scalar data streams, but processed as vector variables, efficient scalar-vector conversion operations are needed to

convert the inter-algorithm scalar variables into vector variables for intra-algorithm vector processing.

3.3 Architectural Design Tradeoffs for SDR

In this section, we discuss the architectural implications for supporting the wireless protocols. This includes managing concurrent DSP algorithm execution, controlling inter-algorithm communication, meeting real-time deadlines, and supporting high-throughput DSP algorithms.

Control Plane Vs. Data Plane. Complete software implementations of wireless protocols usually require two separate steps: 1) the implementation of the DSP algorithms; and 2) the implementation of the protocol system. The DSP algorithms are computational intensive kernels that have relatively simple data-independent control structures. The protocol system has relatively light computation load, but complicated data-dependent control behavior. Therefore, our proposed SDR solution includes a two-tiered architecture: a set of data processors that are responsible for heavy duty data processing; and a control processor that handles the system operations, and manages the data processors through remote-procedure-calls and DMA operations.

Static Multi-core Scheduling Vs. Multi-threading. Traditional micro-architectural techniques, such as simultaneous multithreading and cache coherency, that were originally developed for server-class multiprocessors provide a convenient abstraction to the programmer but they can be of limited value in high-throughput embedded systems. Strict real-time requirements also require deterministic architectural behavior. This implies that micro-architectural features that trade-off good average-case performance for non-deterministic and poor worst-case performance (e.g., caching, multi-threading and

prediction) are not well suited to these applications.

To reduce hardware complexity and produce efficient deterministic code behavior, we omit multi-threading and cache coherency support. Instead, each protocol pipeline is broken up into kernels, and each kernel is assigned statically to a processing element which is then statically scheduled to execute according to the algorithm data flow. This model grew out of our observations that inter-kernel communication throughput is low, and intra-kernel computational throughput is high. Therefore, the static scheduling approach can result in less communication traffic compare to the case when kernels are split into threads.

Scratchpad Memory with Data Streaming. In addition to data computation, programmers also need to handle the data communications between algorithms. These inter-algorithm communications are usually data streaming buffers that are ideal for non-blocking DMA transfers. While the processors operate on the current data, the next batch of data can be streamed between the memories and register files in the background. Streaming computations have been previously proposed for multi-media processor architectures, including the Imagine processor [10], and the IBM Cell processor [40]. They have shown that scratchpad memories, instead of cache, are best suited for streaming applications. We find that streaming computation fits naturally with wireless protocol computations.

Wide SIMD Execution Unit. The computation intensive DSP algorithms in wireless protocols usually contain operations on very wide vectors. In addition, vector widths and strides are always known during compile time. Although a vector architecture would be a good fit for wide vector computation, the extra hardware support for dynamic vector management is unnecessary, as all data operations can be statically scheduled. This favors a wide SIMD-styled clustered execution. Traditional SIMD architectures have a

short SIMD width due to the difficulties in data alignment. In addition, general purpose SIMD accelerators usually need to support a large range of data sizes (for example, Intel MMX [65] supports 8/16/32/64 bit SIMD operation). Therefore, the bottlenecks of a SIMD system are often the data movement and alignment operations, not data computation operations. Previous studies have addressed this problem through complex multi-ported memories and register files, or a full crossbar interconnect. In the context of the power budgets for mobile devices, these are infeasible solutions.

Wireless protocols' DSP algorithms have well-defined intra-vector data permutation patterns and operate on 8- and 16-bit data. These traits significantly simplify the data movement requirements. Therefore, we can afford to scale up the SIMD width to exploit DSP algorithms' very wide vector operations, with the intra-vector data permutation supported through an SIMD shuffle network.

Asymmetric VLIW Instructions. In addition to the heavy vector computation, there are many small, yet equally important scalar DSP algorithms in wireless protocols. Wide SIMD execution units are too inefficient for these scalar and narrow SIMD operations. Therefore, architectural support for scalar execution is also needed. In most cases, scalar operations can be executed concurrently in VLIW lock-step with the SIMD operations. The VLIW is asymmetric because instructions for SIMD pipeline cannot execute on the scalar pipeline, and vice versa.

3.4 SODA Architecture for SDR

3.4.1 Architecture Overview

The SODA multiprocessor architecture is shown in Figure 3.1. It consists of multiple processing elements (PEs), a scalar control processor, and global scratchpad memory, all

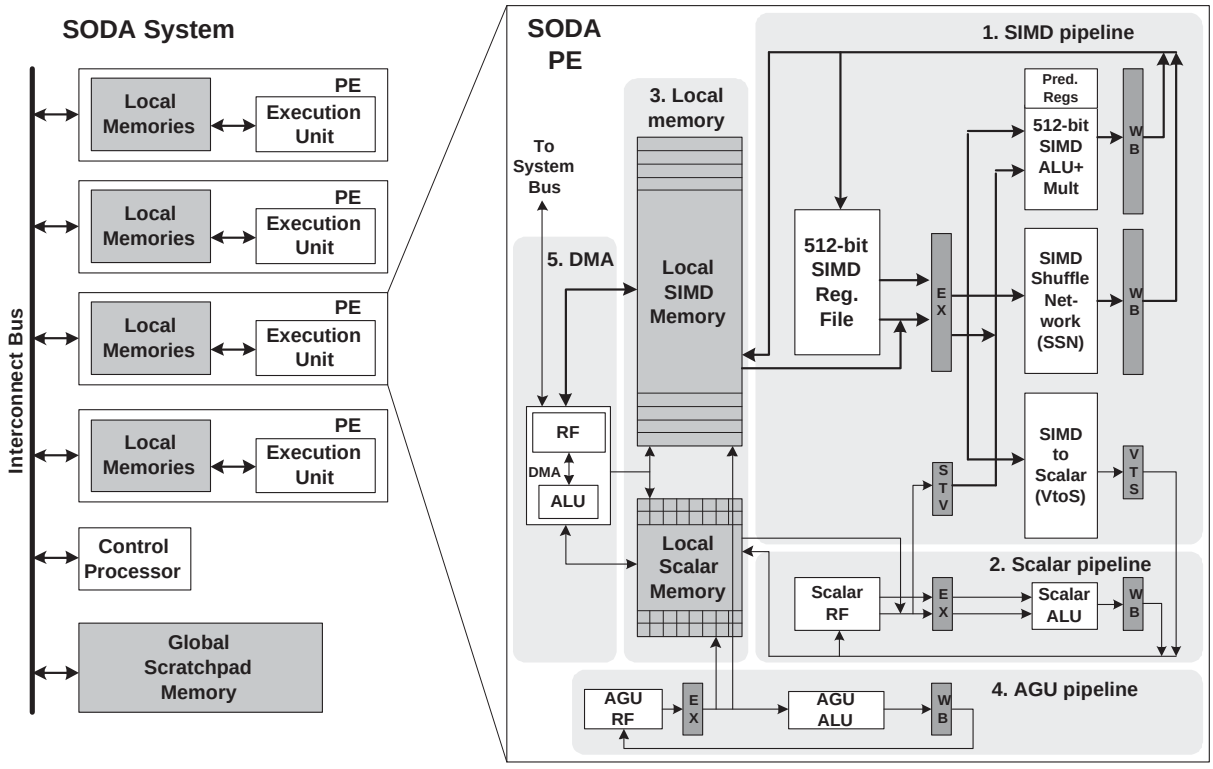


Figure 3.1: SODA Architecture for SDR. The system consists of 4 data processing elements (PEs), 1 control processor, and global scratchpad memory, all connected through a shared bus. Each PE consists of a 32-wide 16-bit SIMD pipeline, a 16-bit scalar pipeline, two local scratchpad memories, an Address-Generation-Unit(AGU) for calculating memory addresses, and a Direct-Memory-Access (DMA) unit for inter-processor data transfer.

connected through a shared bus. Each SODA PE consists of 5 major components: 1) an SIMD (Single Instruction, Multiple Data) pipeline for supporting vector operations; 2) a scalar pipeline for sequential operations; 3) two local scratchpad memories for the SIMD pipeline and the scalar pipeline; 4) an AGU pipeline for providing the addresses for local memory access; and 5) a programmable DMA unit to transfer data between memories and interface with the outside system. The SIMD pipeline, scalar pipeline and the AGU pipeline execute in VLIW-styled lock-step, controlled with one program counter (PC). The DMA unit has its own PC, its main purpose is to perform memory transfers and data rearrangement. It is also the only unit that can initiate memory access with the

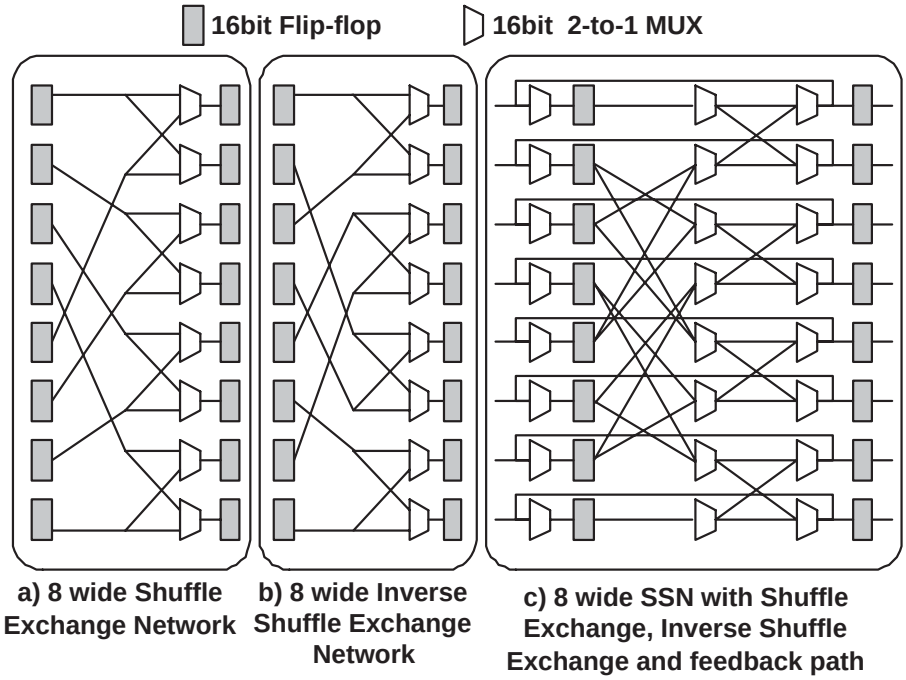


Figure 3.2: 8-wide SIMD Shuffle Network(SSN)

global scratchpad memory.

The SIMD pipeline consists of a 32-way 16-bit datapath, with 32 arithmetic units working in lock-step. It is designed to handle computationally intensive DSP algorithms. Each datapath includes a 2 read-port, 1 write-port 16 entry register file, and one 16-bit ALU with multiplier. The multiplier takes two execution cycles when running at the targeted 400MHZ. Intra-processor data movements are supported through the SSN (SIMD Shuffle Network), as shown in Figure 3.2. The SSN consists of a shuffle exchange (SE) network, an inverse shuffle exchange (ISE) network, and a feedback path. Previous work [82] has shown that any permutation of size N can be done with $2log_2N-1$ iterations of either the SE or ISE network, where N is the SIMD width. For the permutation patterns of SDR algorithms, we found that we can reduce the number of iterations if we include both the SE and ISE networks. In addition to the SSN network, a straight-through connection is also provided for data that does not need to be permutated.

The SIMD pipeline can take one of its source operands from the scalar pipeline. This feature is useful in implementing trellis computations. It is done through the STV (Scalar-To-Vector) registers, shown in the SIMD pipeline portion of Figure 3.1. The STV contains 4 16-bit registers, which only the scalar pipeline can write, and only the SIMD pipeline can read. The SIMD pipeline can read 1, 2, or all 4 STV register values and replicate them into 32-element SIMD vectors. SIMD-to-Scalar operations transfer values from the SIMD pipeline into the scalar pipeline. This is done through the VTS (Vector-To-Scalar) registers, shown in Figure 3.1. There are several SIMD reduction operations that are supported in SODA, including vector summation, finding the minimum and the maximum.

The DMA controller is responsible for transferring data between memories. It is the only component in the processor that can access the SIMD, scalar and global memories. Traditional DMA controllers perform copies from one memory region to another, where regions are either contiguous or have a simple stride access patterns. They are usually implemented as a slave device, controlled through a set of DMA registers and synchronization instructions that are executed on the master processor. In our processor, the DMA is also implemented as a slave device controlled by the scalar pipeline. However, it has the capability to execute its own instructions on its internal register file and ALU, similar to the scalar pipeline. This gives the DMA the ability to access the memory in a wide variety of application-specific patterns without assistance from the master processor. This ability allows inherently scalar memory transfer algorithms, such as interleaving, to be implemented efficiently on the DMA.

3.4.2 Arithmetic Data Precisions

SODA PE only provide support for 8- and 16-bit fixed-point operations. Many embedded processors also have support for 32-bit fixed point or floating point arithmetic operations. We found this to be unnecessary for wireless baseband processing. In W-CDMA wireless protocol, the majority of the algorithms operate on 1- to 8-bit data, with some algorithms operate on 16-bit data. In 802.11a wireless protocol, the majority of the DSP algorithms operate on 16-bit fixed point data, with a few algorithms operate on 8-bit data. Neither protocol requires 32-bit fixed point or floating point support. Therefore, SODA PE is optimized for 8-bit and 16-bit operations in the SIMD lane by supporting 32-lane 16-bit SIMD fixed-point arithmetic operations. Because floating point units require large area, not supporting the floating point provides significant power and area saving. The AGU registers are 12-bit, but only support 8-bit addition and subtraction. This is because AGU is used for software management of data buffers, in which 8 bits are sufficient. The higher 4 bits are used to address different PEs, as well as different memory buffers within PEs.

3.4.3 Vector Permutation Operations

With SODA's SSN network (shown in Figure 3.2), any 32-wide vector permutation can be performed with the maximum of 9 cycles. Combining with predicated move operations, the SSN network can support any vector length permutation. However, for every 32-wide vector permutation, one additional instruction must be used to setup the permutation network. This is because each MUX within the shuffle network requires its own control bit. Each iteration through the SSN requires 64 bits. For a maximum of 9 iterations, this requires 576 bits of control information. The SSN is not very efficient if there are many

permutation patterns that are used frequently. It is better suited for algorithms with only single or a few shuffle patterns. Through our algorithm analysis, we find that this is the case for SDR algorithms. The majority of the algorithms only use one or a few shuffle patterns, which makes the network setup overhead not significant.

3.4.4 Long Vector Arithmetic Operations

Different DSP algorithms have varying levels of data level parallelism. From a DSP programmer's perspective, it is easier to express operations in the algorithm's native vector width, rather than the PE's SIMD width. For example, a 64 tap FIR filter can be expressed most succinctly with 64-wide vectors. We refer to this native vector width as the virtual vector width. Translating assembly code with virtual vector width into code with PE's specific SIMD width is straight forward in most cases. For most SIMD operations, such as add or comparison, this simply requires duplicating the instructions. Some vector permutation operations, such as vector shift up/down, can be translated completely in software, but would benefit from having hardware support: the operation requires one scalar register to store the value that is shifted out, and one scalar register to feed in the value that is shifted in. Vector to scalar operations, such as finding the minimum value of a vector, or calculating the dot product, benefit greatly from hardware support. Overall, most of the hardware support consists of temporary scalar registers that can be read or written by the SIMD units.

3.4.5 Vector-Scalar Move Operations

Most of the inter-kernel communications are via scalar streams, but intra-PE computations are vector operations. Therefore, support for a scalar-vector interface between the scalar and SIMD pipeline is needed. In our architectural diagram, Figure 3.1, this is

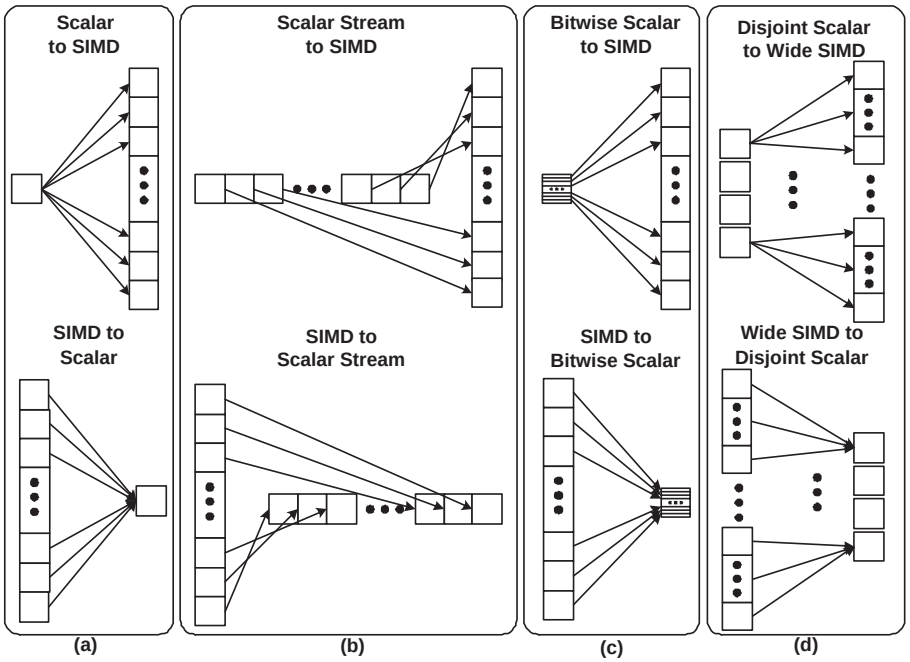


Figure 3.3: Scalar-SIMD Operations for Various DSP Algorithms

shown as the StoV and VtoS units. The StoV network spreads a scalar value into a wide vector, and VtoS network reduces a wide vector into a scalar value. The VtoS network is a 32-to-1 reduction tree. It is pipelined into two stages.

From our benchmark analysis, we found four types of scalar-vector operations, shown in Figure 3.3. *Scalar-SIMD Operations* – These operations spread one scalar variable into a vector, and reduce a vector down to one scalar variable, as shown in Figure 3.3a. Filter, Viterbi Decoder, and Turbo Decoder fall into this category.

ScalarStream-SIMD Operations – These operations transfer an array of scalar variables directly into a vector, as shown in Figure 3.3b. If implemented directly through the scalar-SIMD StoV network, the transfer operations would take up to 32 cycles, as each scalar value is spread sequentially from the scalar to SIMD pipeline. However, with the AGU unit, we can redirect our DMA to transfer data directly into the SIMD memory, bypassing the scalar pipeline. The algorithms in this category include FFT and the RAKE receiver.

BitwiseScalar-SIMD Operations – These operations spread the bits of a scalar value into a vector, as shown in Figure 3.3c. Vectors with 1bit elements are common in DSP. One example is the searcher, which correlates vectors of individual bits that make up the received signal stream. This special expansion/reduction functionality is supported in the StoV and VtoS networks.

DisjointScalar-SIMD Operations – These operations support expansion/reduction operations between multiple disjoint scalar values and wide SIMD vectors, as shown in Figure 3.3d. This feature is useful for algorithms with native vector width less than the SIMD's width. An example is the Turbo decoder. It has a native vector width of 8 elements. Our SIMD datapath can operate on 32 elements at once. Running a sequential version of the Turbo Decoder only utilizes 25% of the SIMD pipeline. Using the sliding window technique, Turbo decoder can be parallelized to process 4 data streams in lock-step SIMD-style execution. However, the 4 data streams require 4 separate scalar values. In order to handle expansion and reduction of multiple disjoint scalar values, the StoV and VtoS networks are modified to include a 4-wide 16bit disjoint scalar (DS) register. For StoV expansion, four separate values are first read sequentially from the scalar pipeline into the DS register, and then 4 8-wide expansions are performed. For VtoS reduction, the SIMD vector is first reduced to 4 16bit values, stored in the DS register. The scalar pipeline then processes these 4 values sequentially. Although the scalar operations are still processed sequentially, because the majority of the computations are vector operations, this scalar overhead has minimal effect on the overall performance.

3.4.6 Algorithm Specific Operations

Implementing customized complex instructions is very common in DSP processors. A typical example of such an instruction is Multiply-Accumulate (MAC) or saturated arith-

Instruction Type	Mnemonics	DSP Kernels	Description
Shuffle	Compare and Select `VCS Vd, Vn, Vm`	Viterbi, Turbo	*t0 = (Vn[0,2,max], Vm[0,2,max]* *t1 = (Vn[1,2,max], Vm[1,2,max]* *Vd = (t0 > t1) ? t0 : t1*
	Butterfly `BFLY Vd, Vn, #imm`	FFT, IFFT	*Vd[i+j*imm*2] = Vn[i+j*imm*2+imm]* *Vd[i+j*imm*2+imm] = Vn[i+j*imm*2]*
Vector to Scalar	Vector Sum `DOT Sd, Vm, Sn`	Searcher Despreader	*Sd = Sn + sum(Vm)*
	Vector Max `MAX Sd, Vm, Sn`	Viterbi Searcher	*Sd = max(max(Vm), Sn)*

Predicated Execution	Predicated Negation
	`ADD.p Vd, Vm, Vn`
	`SUB.p Vd, Vm, Vn`
		Searcher Descrambler	*Vd[i] = (1-2P0[i])*Vm[i] + (1-2P1[i])*Vn[i]*
			*Vd[i] = (1-2P0[i])*Vm[i] - (1-2P1[i])*Vn[i]*
			P0,P1: 32 1-bit Pred. mask vector

Special ALU	Double Multiply `MUL2 Vd1,Vd2, Vm,Vn,Sn,Sm`	FIR	*Vd1[i] = Sn*Vn[i]* *Vd2[i] = Sm*Vm[i]*

Figure 3.4: Special Intrinsic Operations

metic instructions. Figure 3.4 lists the operations commonly found in wireless protocols. The first class of intrinsic operations includes special vector permutations supported by the SSN. The Vector Compare & Select (VCS) operation, needed in Viterbi and Turbo decoders, compares and selects between two adjacent vector elements. The butterfly operation is implemented to enhance the FFT performance. The second type of special instruction is used to convert a vector to a single scalar value. This instruction is heavily used in the synchronization and modulation kernels. The Vector Max instruction is crucial for Viterbi and Turbo since calculating the maximum value of a vector without special hardware support is very inefficient. The last type of special instruction is the predicated negation. This instruction uses two 1-bit vectors to control the sign of the two ALU operands, and conditionally negates the operands before executing an addition (subtraction). This instruction is equivalent to a multiplication of the operands with a 1bit number that can be either +1 or -1. It is used for auto-correlation and modulation in W-CDMA.

3.4.7 Vector Alignment Through Programmable DMA.

The DMA controller is responsible for transferring data between memories. It is the only component in the processor that can access the SIMD, scalar and global memories. Traditional DMA controllers perform copies from one memory region to another, where regions are either contiguous or have a simple strided access patterns. It is usually implemented as a slave device, controlled through a set of DMA registers and synchronization instructions that are executed on the master processor. In our processor, the DMA is also implemented as a slave device controlled by the scalar pipeline. However, it has the capability to execute its own instructions on its internal register file and ALU, similar to the scalar pipeline. This gives the DMA the ability to access the memory in a wide variety of application-specific patterns without assistance from the master processor. This ability allows inherently scalar components of the Turbo decoder, such as the interleaver, to be implemented efficiently on the DMA.

3.4.8 Embedded Low-power Design

In order to achieve the high computational requirements, while maintaining low power requirements, SODA utilizes the following techniques:

Intrinsic Operations – Traditional DSP processors rely heavily on MAC operations to achieve efficiency. We found that many of these multiplication operations are with 1 bit values, and can be implemented by vector logic operations (Predicated Negation) that consume significantly lower power.

Clustered Register Files with 2 Read Ports and 1 Write Port – Each PE's register file is a cluster of three separate files: an SIMD RF, scalar RF, and AGU RF. For a 32-way vector operation, each RF in our PEs requires only 2 ports. As shown in previous

studies [71], increasing the number of register file ports increases the register file's power consumption quadratically. By using less ports, our implementation saves register file power.

Fewer Memory Read/Write Ports – Each PE's local memory is a cluster of 2 memories—4KB for the scalar memory and 8KB for the SIMD memory. In general, two memories consume lower power than one unified memory. In addition, our SIMD memory requires a 512bit read/write port, but our scalar memory only requires a 16bit read/write port. Separate memories allow us to further optimize each for its intended use.

Smaller Instruction Fetch Logic – Our VLIW instructions use a fixed sized instruction width of 64bits split into three fields—Scalar, AGU, and SIMD. Commercial DSP solutions often have variable length instruction widths of 96-128bits. In addition, they do not support SIMD instructions or a vector ISA that allows us to efficiently express long vector operations, effectively reducing an algorithm's code size.

3.5 SIMD Design Tradeoffs

To justify the SODA system configuration with wide SIMD pipelines, we have done a study on the first-order power consumption trade-offs SIMD width and frequency. This study was done using 180nm technology. We estimate that 40GOPS would be required in order to meet the realtime computation requirements of W-CDMA and 802.11a. Furthermore, we find that both W-CDMA and 802.11a can be partitioned into 4 major task groups that are relatively balanced (see Figure 3.6c for the W-CDMA implementation). So, in this thesis, we examine the power consumption of a 4 PE system. In a 4 PE system, each PE will need to supply approximately 10GOPS. On one end of the spectrum, the PE can be a 2-wide SIMD running at 5GHZ; and on the other end, it can be a 256-wide SIMD

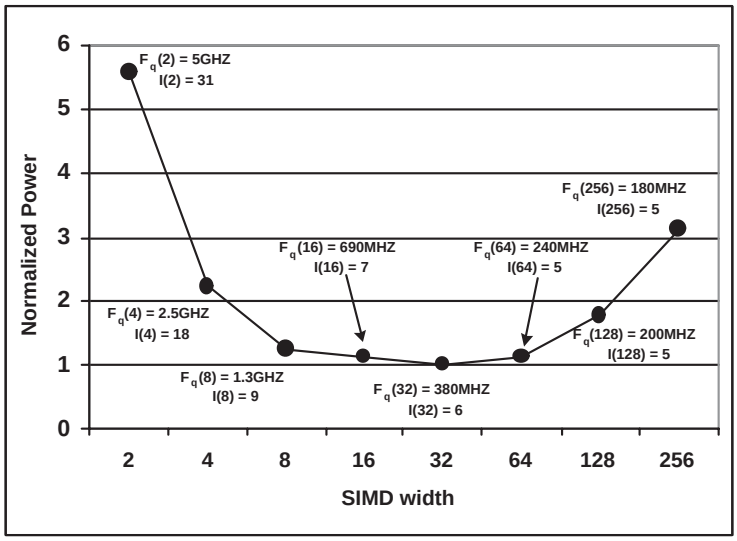


Figure 3.5: Average normalized power of a 4-PE configuration for achieving the computational requirements of W-CDMA and 802.11a in 180nm technology

running at 180MHZ. Intuitively, the optimal SIMD configuration should lie within these two extremes. Figure 3.5 summarizes our findings to determine the power consumption of a PE.

The following is our methodology for calculating the first-order power consumption P for a single PE with a workload requirement of T GOPS, and an SIMD width of w. We assume a homogeneous system with PEs having the same SIMD configuration and frequency. If N is the number of algorithms running on the SIMD pipeline, then $T = \sum_{i=1}^{N} T_i$, where T_i is the workload of the ith algorithm in the protocol.

The required frequency of a PE, $F(w)$, as a function of w, can be calculated by summing up the frequencies of each individual algorithm.

$$F(w) \quad = \quad \sum_{i=1}^{N} \left(\frac{T_i}{V_i} \left\lceil \frac{V_i}{w} \right\rceil \right) \tag{3.1}$$

Where V_i is the native vector width of algorithm i. The first term in the summation,

$\frac{T_i}{V_i}$, calculates the frequency for meeting the computational requirement of T_i GOPS in terms of number of vector operations per second. The second term, $\lceil \frac{V_i}{w} \rceil$, calculates the number of cycles to perform a vector operation of size V_i on a SIMD processor of width w. In the case of $w > V_i$, the SIMD is under utilized since the vector is narrower than its SIMD width. The exception is the Turbo decoder, where we can use the sliding window technique to compute multiple windows in parallel and thereby exploit the wider SIMD width. In our analysis, T_i and V_i are calculated based on the W-CDMA and 802.11a protocol profiling results shown in Figure 3.7.

Given the limitation of silicon technology, there is an upper bound on the achievable frequency. PEs with narrow SIMD width that require ultra high frequency will have to implement deeper pipelines. In this study, we scale up the pipeline depth based on SODA's 5 stage pipeline organization. There are four architectural components that contribute to the overall SIMD pipeline depth: the register file (R), ALU (A), SIMD memory (M), and SSN (S). We ignored the WtoS network, because it is not a part of the SIMD pipeline. Equation 3.2 expresses the overall pipeline depth I, as the sum of the register file pipelines(I_r), the ALU pipelines(I_a), and the maximum of the memory pipeline(I_m) and SSN pipeline(I_s).

$$I(w) = 2I_r(w) + I_a(w) + max(I_m(w), I_s(w)) \tag{3.2}$$

In Equation 3.2, I_r is scaled by 2 due to the two separate pipelines stages for register read and write. Also the maximum of I_m and I_s is used because SSN and memory share the same pipeline stage. The pipeline depth of each component is obtained from synthesized Verilog, for frequency $F(w)$.

Let $E(w)$ be the energy consumption of one cycle of operation for one PE, and let

$P(w)$ be PE's power consumption.

$$\begin{aligned} E(w) &= C + w(L_e I + R_e U_r + A_e U_a) + \\ &\quad M_e(w)U_m + S_e(w)U_s + D_e(w)U_d \qquad (3.3) \\ P(w) &= E(w) \cdot F(w) \qquad (3.4) \end{aligned}$$

In Equation 3.3, C is the constant power overhead, due to the scalar and AGU pipeline, the instruction memory, and the DMA controller. L_e is one 16bit datapath pipeline's flip-flop energy, R_e is one 16bit register file's energy, A_e is one 16bit ALU's energy, M_e is the SIMD memory energy, S_e is the SSN energy, and D_e is the WtoS reduction network energy. All of the above energy results are for one cycle of operation. Because memory, SSN, and reduction network do not scale linearly with the SIMD width, we model them empirically by synthesizing each configuration in Verilog. U_x represents the average utilization factor for component x, gathered from behavioral simulations on the SODA simulator.

Figure 3.5 shows the normalized power as a function of SIMD width for average W-CDMA and 802.11a workloads. Each point is annotated with its operating frequency, $F(w)$, and the number of pipeline stages, $I(w)$. We see that smaller SIMD width configurations consume less power per cycle, but require unrealistically deep pipelines. For example, the 4-wide SIMD configuration requires 18 pipeline stages. Wider SIMD configurations have shorter pipelines with low operating frequencies, but suffer from under-utilization. Figure 3.5 shows that the 32-wide SIMD consumes the lowest power. The 8, 16, and 64 wide SIMD also achieve similar power consumption, and would be acceptable design points. The power numbers have been derived assuming that underutilized SIMD processors still consume dynamic power for the unused SIMD units. However, we can employ simple clock-gating techniques to turn off the unused units, thereby reducing wide

SIMD's power consumptions. In addition, frequency and voltage do not scale linearly with nanometer CMOS technologies. In sub-90nm implementations, narrow width SIMD will result in deeper pipelines. Consequently, the optimal power consumption point is likely to shift to higher SIMD width in future technologies, if leakage can be contained.

3.6 Experimental Evaluation

SODA Behavioral Analysis. In addition to W-CDMA, we have also implemented the 802.11a wireless protocol's physical layer processing. In order to test the performance of SODA, we first developed both W-CDMA and 802.11a physical layer system implementations in C. This approach enables full system performance including memory requirements, synchronization schemes, and non-parallelizable bottlenecks to be evaluated. Next, we hand-coded the benchmarks into the SODA instruction set. To evaluate intra-kernel synchronization and data flow characteristics, we built an inter-processor network simulator based on our PE's cycle-accurate processor simulator, and DMA transfers and bus synchronization schemes.

SODA Area And Power Model. Area estimation of our architecture was calculated using our RTL Verilog model of the SODA architecture. We synthesized our Verilog model using Synopsys Physical Compiler and Cadence Silicon Ensemble for 400MHz using the TSMC 180nm library. The memories were generated with the Artisan SRAM memory generator. The estimated area for 90nm and 65nm processes were calculated using a quadratic scaling factor.

Dynamic power was estimated using utilization factors of each PE derived from our behavioral simulations of the W-CDMA and 802.11a protocols on our system simulator. For each PE we then took the dynamic power results from Physical Compiler and used the

utilization factors to calculate the dynamic power of that PE. The dynamic power of the memories were derived form the Artisan SRAM memory generator. Using the synthesized model of the PE, we extracted the interconnect power and added it to the dynamic power then summed up the PEs to calculate the total dynamic power of the system.

To scale to 90nm and 65nm, we used the Predictive Technology Models (PTM) [6] and simulated in SPICE based on a delay of 20 F04 in 180nm at 1.8V, 90nm at 1V, and 65nm at 0.8V.

The leakage power was estimated at 30% of the total power. This is in accordance with ITRS specifications for 90nm technology. We used the more conservative approach, since the PTM leakage results were lower than industry trends.

3.6.1 Protocol System Implementations

For this discussion we will focus on W-CDMA instead of 802.11a, because its behavior is more complex. Figure 3.6 shows the real-time W-CDMA 2Mbps DCH (Dedicate CHannel) execution trace on SODA: Figure 3.6a shows the system execution of 1 frame of data; Figure 3.6b shows one slot of execution using the macro-pipelining message passing protocol; and Figure 3.6c shows the functional mapping of W-CDMA onto SODA. DCH is a full duplex channel consisting of the DPDCH (Dedicated Physical Data CHannel) for uplink and the DPCH (Dedicated Physical CHannel) for downlink. In the W-CDMA specification, DCH also includes the uplink DPCCH (Dedicated Physical Control CHannel), which is not modeled in this study. The uplink and downlink channels are mapped onto their own PEs. This assignment achieves a relatively balanced workload across the 4 PEs.

To better understand W-CDMA execution, consider Figure 3.6a. The horizontal axis is time, and the vertical axis lists the SODA's PEs, and their real-time processing utilization.

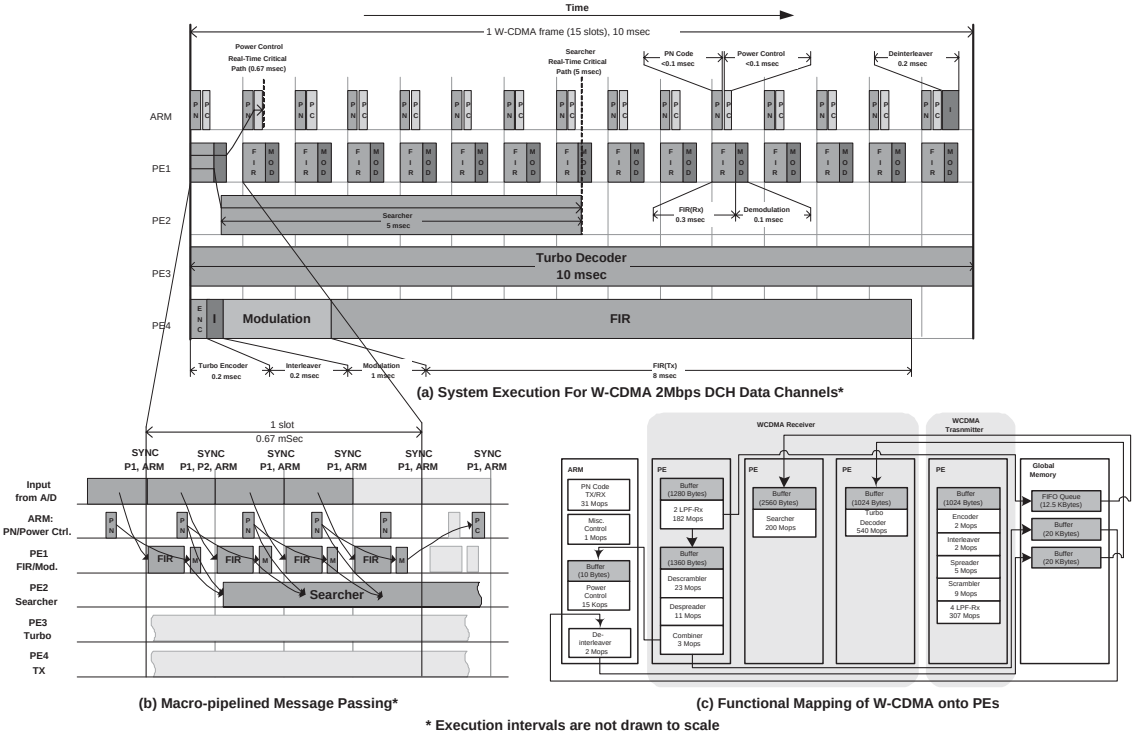


Figure 3.6: W-CDMA 2Mbps DCH data channel implementation. The kernel mapping is shown with the algorithm mapping and memory allocation on the PEs, control processor, and global memory. The execution trace is shown with two periodic real-time deadlines: power control and searcher.

The utilization of PE1, PE2, PE3, and PE4 are 60%, 50%, 100% and 94% respectively. One W-CDMA frame contains 15 slots. There are two hard real-time deadlines that have to be met in W-CDMA. The first one is the power control critical path that controls the transmission power based on received signal quality. It needs to update periodically once per slot (0.67 msec). The critical path is the channel between the FIR Rx filter, Demodulation and Power Control. This is a streaming channel with minimal memory storage requirements. The other real-time critical path is the channel from the FIR filter to the searcher. This needs to complete within 5 msec and requires a large amount of data buffering. Figure 3.6b shows the multi-PE execution using our macro-pipelined message passing protocol. Data is streamed from one PE to the next, synchronizing only on the

macro-clock boundary.

From the above analysis, we see that while throughput is essential, fast intra-processor kernel switching, and efficient inter-processor communication are also essential. Because wireless protocols have static run-time characteristics, compile-time scheduling of the kernels is sufficient to reduce unnecessary context switching overhead. Our macro-pipelining message passing technique reduces the inter-PE synchronization overhead by pipelining data transfers, and exploiting the streaming nature of inter-kernel communication.

3.6.2 Performance and Power Results

Performance Results. Figure 3.7 provides a characterization of each kernel algorithm in W-CDMA and 802.11a in terms of extent of vectorization, vector width, bit width, etc. This characterization was used to define the SODA architecture. Figure 3.7 also lists the throughput and latency of each kernel algorithm when implemented on SODA. The raw computations are measured in terms of number of execution cycles on a general purpose Alpha processor. The SODA computation, on the other hand, is the number of execution cycles required by the SODA vector ISA. It can be seen that large speedups are possible in many cases. For instance, W-CDMA's searcher algorithm, which requires 26.5 Gops on a general purpose processor, is reduced to 200 Mops on SODA. Part of the speedup is due to SODA's wide SIMD execution, and part of it is due to the fact that SODA assembly code is hand-optimized.

Because W-CDMA is designed to support mobile communications, its workload is highly dependent on the environment conditions. In this study, the descrambler, despreader, combiner, and searcher are benchmarked with the worst case environment condition, because they include real-time deadlines that must be met under the heaviest workload. The Turbo decoder is benchmarked with the average case workload because it

Algorithms	Configurations	General Purpose Processor (Alpha)				SODA
		Vector Comp.	Vector Length	Bit Width	Comp. Mcycles/sec	Comp. Mcycles/sec
W-CDMA (2Mbps)						
Scrambler	Defined in W-CDMA standard	yes	2560	1,1	240	9
Descrambler*	12 fingers, 3 base stations	yes	2560	1,8	2,600	23
Spreader	Spreading factor = 4	yes	512	8	300	5
Despreader*	12 fingers, 3 base stations	yes	512	8	3,600	11
PN Code (Rx)	3 base stations	no	1	8	30	23
PN Code (Tx)	Defined in W-CDMA standard	no	1	8	10	8
Combiner*	2Mbps data rate	partial	12	8	100	3
FIR (Tx)	4 filters x 65 coeff x 3.84Msps	yes	64	1,16	7,900	307
FIR (Rx)	2 filters x 65 coeff x 7.68Msps	yes	64	8,8	3,900	182
Searcher*	3 base stations, 320 windows	no	320	1,8	26,500	200
Interleaver	1 frame	no	1	8	10	2
Deinterleaver	1 frame	partial	1	8	10	2
Turbo Enc.	K=4	yes	3	1,1	100	2
Turbo Dec.*	K=4, 5 iterations	yes	8	8,8	17,500	540
*These algorithms have dynamically changing workloads that are dependent on channel conditions						
802.11a (24Mbps)						
FFT	64 points	yes	64	16	15,600	240
IFFT	64 points	yes	64	16	15,600	240
Equalizer	64 points	yes	54	16	960	120
QAM	64 constellation points	no	1	4	1	2
DQAM	64 constellation points	no	1	4	3	2
FIR (Tx)	1 filter x 33 taps x 20Msps x 2	yes	33	16	3,040	160
FIR (Rx)	1 filter x 33 taps x 40Msps x 2	yes	33	16	6,080	320
Freq. Sync.	Defined in 802.11 standard	partial	16	16	190	10
Time Sync.	Defined in 802.11 standard	partial	16	16	190	10
Interpolator	Farrow structure cubic 8 taps	yes	2048	16	4,800	250
Interleaver	1 frame	no	1	1	290	60
Deinterleaver	1 frame	no	1	16	290	60
Conv Enc.	K = 7, Rate = 3/4	partial	6	1	100	40
Viterbi Dec.	K = 7, Soft input	partial	64	8	35,000	398
Scrambler	Defined in 802.11 standard	partial	7	1	340	34
Descrambler	Defined in 802.11 standard	partial	7	16	340	34

Figure 3.7: Kernel Algorithms in W-CDMA and 802.11a and their performance on a GPP and SODA.

			Area		W-CDMA 2Mbps		802.11a 24Mbps	
	Components	Units	Area mm^2	Area %	Power mW	Power %	Power mW	Power %
PE	SIMD+scalar Data Mem (8KB+4KB)	4	6.1	23%	87	3%	67	2%
	SIMD Register File (16x512bit)	4	1.9	7%	1077	37%	874	27%
	SIMD ALUs and Multipliers	4	6.7	25%	314	11%	609	19%
	SIMD Pipeline+Clock+Routing	4	1.5	6%	1127	38%	1157	36%
	Intra-processor Interconnect	4	1.1	4%	53	2%	53	2%
	Scalar Pipeline+Inst Mem+Inst Fetch	4	3.1	11%	274	9%	329	10%
System	ARM (Cortex-M3)	1	0.6	3%	5	< 1%	10	< 1%
	Global Scratchpad Memory (64KB)	1	3.6	14%	10	< 1%	80	2%
	Inter-processor Bus with DMA	1	2.0	7%	3	< 1%	26	1%
Total	180nm (1.8V @ 400MHZ)		26.6	100%	2950	100%	3206	100%
Est.	90nm (1V @ 400MHZ)		6.7		447		486	
	65nm (0.8V @ 400MHZ)		3.5		236		257	

Figure 3.8: System Area and Power Summary

has flexible deadlines that allow its inputs to be buffered. This is why it is acceptable for the decoder to take 540Mcycles(1.35 seconds) to finish one second of computation.

Power and Area Results. Figure 3.8 lists the area and power breakdowns of the SODA system. The wide SIMD design means the SIMD pipeline and clock logic consumes the largest amount of power. The SIMD register file is also one of the major power consumers (37% in W-CDMA and 27% in 802.11a), due to heavy utilizations during vector computations. SIMD memory power is higher for W-CDMA (87mW) than for 802.11a (67mW). This is because most 802.11a algorithms have vector width less or equal to 64, so the SIMD register values do not spill into the memory. In contrast, W-CDMA has more algorithms with long vectors that need to be buffered in memory. The SIMD ALU power consumption is significantly higher for 802.11a than for W-CDMA, because 802.11a's FFTs requres many 16-bit multiplications, whereas the majority of the W-CDMA computations are additions. In our synthesized design, a 16-bit multipliers consumes approximately 10x more power than an 16-bit adder. This is the principal reason why 802.11a consumes more power than W-CDMA. The intra-processor interconnect consumes very little power for both 802.11a and W-CDMA. Finally, low inter-PE

communications implies that the bus power consumption is also not a concern (3mW for W-CDMA and 26mW for 802.11a). Overall, the results show that a power efficient wide SIMD multi-PE architecture can be designed using simple register files, partially connected intra-processor interconnect, and a low power bus-based inter-processor network.

The area results, shown in Figure 3.8, indicate that the ALUs with multipliers and the scratchpad memories take the largest area in the PE. In addition, the PE's local memories (48KB) occupy a larger area ($6.1mm^2$) than the 64KB global memory ($3.6mm^2$), because the local memories are dual-ported, with one port dedicated to the DMA, whereas the global memory is a 32bit single ported memory. The intra-processor interconnect, including the SSN and the WtoS reduction network, is only 4% of the total area. This means that the interconnect network is not a limitation for 32-wide SIMD systems. Of course, if this number were scaled to hundreds, then the interconnect network may start to become a limitation.

Technology Scaling and Power Optimizations. At 180nm, SODA's power consumption is 3W. This is too high for embedded mobile devices. A typical cellular phone's power budget for the physical layer is around 200mW [62]. To see if this constraint can be met, we have estimated the power and area of SODA at state-of-the-art technology nodes of 90nm and 65nm using the Predictive Technology Models. Designs in both technologies fall within the range of acceptable power consumption — 450mW and 250mW respectively. There are other factors that we have ignored that would further reduce power consumption. These include a greater use of custom design, and the observation that many of the W-CDMA algorithms need only 8bit arithmetic. Our studies were based on unoptimized synthesis. In a volume production setting, much of the datapath would be implemented with custom designs to significantly reduce space and power. We previously synthesized an 8bit 32 wide version of SODA, and its power consumption in 90nm was

Algorithm	Configuration	Technology	Power	Throughput	Normalized Power
Turbo Decoder	1 SODA PE K = 5	180nm	800mW	2Mb/s	400 mW/Mbps
	TI Turbo Coprocessor K = 5	130nm	600mW	13.44Mb/s	89.2 mW/Mbps
FIR Filter	1 SODA PE 65 taps	180nm	489mW	200Msps	0.0376 mW/Msps/tap
	ASIC 8 taps	180nm	36mW	550Msps	0.0082 mW/Msps/tap

Figure 3.9: Power efficiency comparison between SODA-based and ASIC implementations for FIR filter and Turbo decoder. The Turbo decoder ASIC data are taken from TI Turbo Coprocessor [26], and FIR filter ASIC data are taken from [77].

about 300mW. However, 802.11 and many next generation protocols use 16bit algorithms, thus an 8bit solution will not meet future demands. There are still many important 8bit algorithms, such as Viterbi decoder. This means that power optimization techniques such as clock-gating, dynamic precision and voltage scaling can be used to reduce power consumption by dynamically adjusting between 8bit or 16bit computations and between different SIMD widths. We are investigating these issues.

3.6.3 The Cost of Programmability

This section examines the cost of programmability by compare the power consumption of SODA-based and ASIC DSP algorithm implementations. The results are shown for the two algorithms with the highest workloads in W-CDMA – FIR filter and Turbo decoder. Searcher also has very high workload requirement, but it is omitted in this study because its operating behavior is similar to the FIR filter. As shown in Figure 3.9, ASICs achieve higher throughput with lower power for both FIR filter and Turbo decoder. The difference in energy efficiency is between 4-5x for both algorithms. There are several factors that contribute to the difference. The most power-hungry components in SODA are the SIMD register file and SIMD pipeline registers. For ASICs, the size and the number of ports for

these registers can be customized for better efficiency. In addition, it is common in these DSP algorithms to perform vector permutation operations followed by vector arithmetic operations. In SODA, this would require multiple operations with unnecessary accesses to the register file for storing intermediate results. An ASIC design can build specialized datapath by chaining the data permutation operations with the arithmetic operations, which bypasses the register file. There are many potential architectural improvements that can be applied to SODA to reduce the energy efficiency gap between SODA and ASICs. Some of these optimizations, such as operation chaining, are implemented in the Ardbeg processor. Details are discussed later in Chapter 5.

3.7 Summary

In this chapter, we describe and discuss architectural trade-offs for designing a domain specific processor for Software Defined Radio. We describe and motivate our multiprocessor, programmable wide SIMD architecture. We show that our architecture is capable of meeting the processing requirements of 3G wireless protocols (W-CDMA and 802.11a) within acceptable power budgets when using a state-of-the-art technology node. Our choice of these two dissimilar protocols was to stress the flexibility of our solution. Further process scaling will enable the support of even more demanding protocols (such as UWB) in a power-efficient manner.

CHAPTER 4

W-CDMA Algorithm Implementations

4.1 Introduction

Because SODA employs an ultra wide 32-lane SIMD pipeline, we cannot rely on existing compiler technology to automatically generate efficient SODA assembly code from high-level C code. In this chapter, we will to present the design and assembly code implementations of the key W-CDMA DSP algorithms on SODA. 802.11a algorithms are similar to those found in W-CDMA, thus their implementations are omitted here. DSP algorithms usually have multiple different implementations. Due to the wide-SIMD design, not all implementations can be mapped efficiently. This chapter describes a set of DSP algorithm implementations that map well on the SODA architecture.

For this study, we first implemented the entire W-CDMA physical layer in C. The C code is compiled with an Alpha gcc compiler and executed on the M5 architectural simulator [16]. The workload profile is shown in Figure 4.1. As shown in the figure, three key algorithms are responsible for the majority of the W-CDMA computations. They are the Turbo decoder, the searcher, and the FIR filters. The rest of the algorithms consumes relatively insignificant amount of computation. The rest of this chapter is going

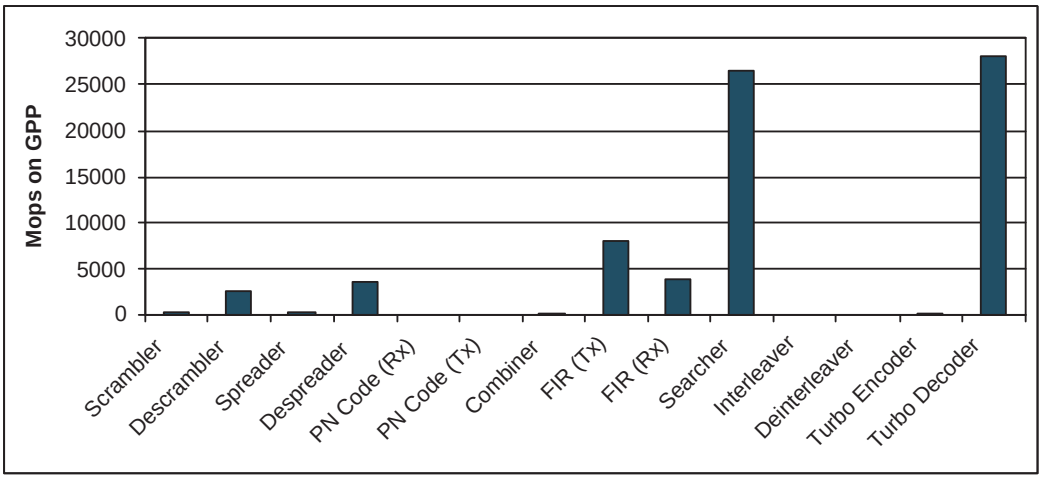


Figure 4.1: 2Mbps W-CDMA workload profile in Mops on a general purpose processor (GPP)

to explain the implementation of these algorithms on SODA. The searcher, descramblers, and despreaders are parts of the rake receiver algorithm, which are all explained in the rake receiver section. Because Turbo decoder requires convolutional decoders as its building block, the implementation of convolutional decoder and Turbo decoders are explained in two separate sections.

4.2 FIR Filter

W-CDMA FIR filters are used to filter out signal terms that exist outside of an allowed frequency band in order to reduce the interference. Let $x[n]$ be the input sequence to be filtered, c_i; c_i be the filter coefficients; and N be the number of filter coefficients. The output filtered sequence, $y[n]$, is expressed as:

$$y[n] = \sum_{i=0}^{N-1} c_i \cdot x[n-i] \tag{4.1}$$

Equation 4.1 is known as the direct form FIR filter, shown graphically in figure 4.2a.

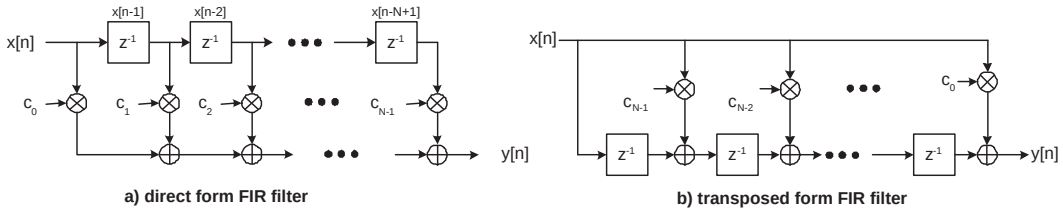


Figure 4.2: FIR filters expressed in direct form and transposed form. The two filter forms are mathematically equivalent

In the figure, Z^{-1} are delay buffers that holds the previous $N-1$ input signals. In direct form FIR filter, the multiplication operations, $c_i \cdot x[n-i]$, are independent of each other, which naturally leads to SIMD parallelization. In W-CDMA filters, the number of filter coefficients equals to 65. This means that all of the multiplication operations can be done through two 32-wide SIMD operations and 1 scalar operation. 64 of the 65 filter coefficients are stored in 2 entries of the SIMD register file, with the last filter coefficient stored in the scalar register file. The most recent 65 elements of input signals, $x[n-64]$ $x[n]$, are stored in the same format in the SIMD and scalar register files. The summation of the multiplications are done through SODA PE's SIMD-to-scalar summation operation. The result $y[n]$ is stored in the scalar pipeline. And finally, because each new $y[n+1]$ only requires the most recent 65 input signals, the oldest input signal, $x[n-64]$, needs to be deleted from the register, and the newest input signal, $x[n+1]$, needs to be added to the register file. This can be done through a SIMD shift down operation, with the most recent input element inserted in the top lane of the SIMD register file.

The bottleneck of the direct form filter implementation is the summation operation because a 32-wide summation operation requires 3 cycles to complete. One alternative is the transpose-form FIR filter, shown in figure 4.2b. The two filter forms are mathematically equivalent. The implementation of the transposed filter has both the input and output signals stored in the scalar pipeline. Each input signal, $x[n]$, is spread into a

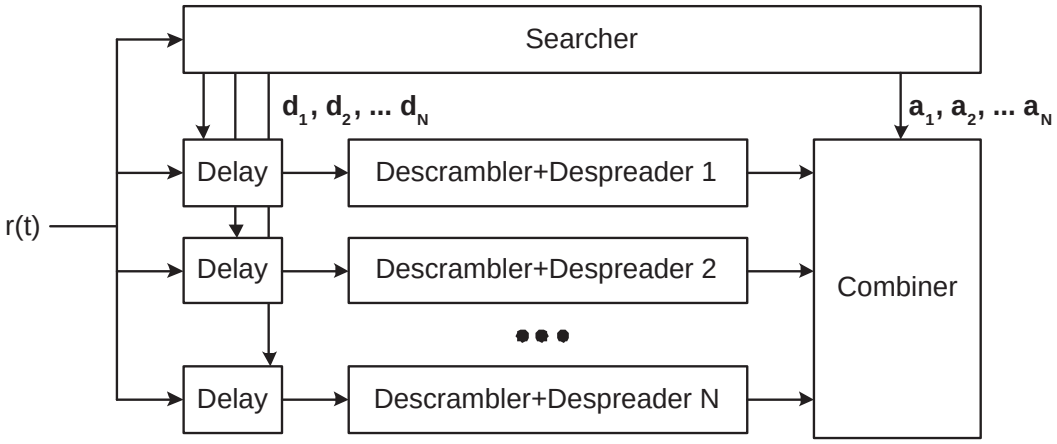


Figure 4.3: W-CDMA rake receiver. It consists of a searcher, despreader/descrambler pairs, and a combiner. Due to multi-path fading effect, a searcher is used to find the synchronization points for each delayed version of the same signal stream. Each despread/descrambler pair correspond to one of the delayed signal stream. And the combiner combines the different paths together.

32-wide SIMD vector through the STV register. The SIMD multiplication and shift operations are the same as the direct form filter's implementation. The output signal, $y[n]$, is just the shifted out value from the SIMD pipeline. However, direct form's summation operation is replaced by a simpler SIMD add operation. This results in a faster FIR filter implementation.

4.3 Rake Receiver

Modulation maps source information onto the transmitting signal waveform. Demodulation extract the source information from the received signal. In the W-CDMA physical layer, two sets of codes are used for modulation: channelization codes and scrambling codes. Channelization codes are used so that the same radio medium can be used to transmit multiple different signal streams. Scrambling codes are used to extract the signal of a specific terminal among many transmitting terminals. On the receiver side,

despreader is used to decode the channelization codes and descrambler is used to decode scrambling codes. Demodulation requires the transmitter and receiver to be perfectly synchronized. However, radio transmission suffers from multi-path fading effect, where multiple delayed versions of the same signal stream are received due to environment interference. Therefore, a searcher is used to find the synchronization point of each one of the delayed signal streams. And each of these delayed signal is decoded with its own despreader and descrambler. These despreader/desrambler pairs are called rake fingers. The decoded output of the rake fingers are then combined together as the output of demodulation. The searcher, despreaders, descramblers, and combiner are together called rake receiver. More detailed explanations of rake receiver can be found in [52]. Figure 4.3 shows the overall diagram of the W-CDMA rake receiver. In the diagram, there are N rake fingers. d_1 d_N are the delayed factors for the N rake fingers. a_1 a_N are the scaling factors for the N rake fingers.

In W-CDMA, signal transmission rate is constant at 3.84M samples per second, where each sample is a complex number consisting of two 4 6 bit values. It is also common practice for the receiving ADC to oversample to improve signal quality, thereby producing two complex values for each sample. W-CDMA transmissions are divided into frames. Each frame is 10ms, containing 38,000 sample points. Each frame is further divided into 15 slots, each containing 2560 samples. In W-CDMA protocol standard, the receiver is not fully specified. Its only requirement is to decode the signal correctly from a fully specified transmitter. Therefore, many of the rake receiver's parameters used in this study are estimated by us, whereas commercial implementations may have different configurations. The estimated parameters include the searcher's window size, block size, peak searching algorithm, and the oversampling factor. On the other hand, descrambling and despreading codes are fully specified, as well as the despreading factor. The following two subsections

will discuss the implementation details of the search and the rake fingers.

4.3.1 Searcher

Searcher is called once per W-CDMA frame. There are two types of searchers — full searcher and quick searcher. Full searcher is called for the first of every 8 frames, and the quick searcher is called for the other 7 of every 8 frames. Both types of searchers consist of four steps: correlation, filtering out high frequency noises, detecting peaks, and global peak detection.

Correlation is the most computationally intensive operation in the searcher. If we let r be the correlation input, and R be the correlation output, then it is defined as:

$$R[t] = \sum_{i=0}^{L_{cor}-1} C_{sc}[i] \cdot r[i+t], where 0 \leq t \leq (L_s - 1) \tag{4.2}$$

In equation 4.2, C_{sc} are the correlation coefficients, L_{cor} is the correlation window size, and L_s is the searcher window size. In the full searcher, 1 correlation is done with $L_s = 5120$. In quick searcher, 4 correlations are done with $L_s = 320$. In both searchers, $L_{cor} = 320$. Comparing equation 4.2 with equation 4.1, we notice that the correlation operation is essentially a 320-tap filtering operation. Therefore, the same transposed form can be applied to the correlation operation to replace the summation operation with cheaper SIMD addition operations. The SIMD implementation of correlation is the same as that of the FIR filter, except that it requires 10 SIMD registers to hold the coefficients for 320-tap coefficients.

Filtering out high frequency noises occurs once for each W-CDMA frame. If we let R be the signal, then filtering equation is given as:

$$R[t] = R[t] \cdot C1 + C2[i] \tag{4.3}$$

$C1$ and $C2$ are both predefined constants ranging between 1 to 8. This is an inherently parallel operation, where each element in the array can be calculated independently.

Detecting peak is implemented as a sequential loop over the searcher window to find all of the peaks. Global peak detection selects the peaks with the maximum correlation values. In our implementations, 4 peaks are selected. We did not spend any effort parallelizing these two steps, because the computation requirements are very small compared to the correlation operation. However, if more complex algorithms are used for detecting peaks, then we may need to reevaluate these algorithms and examine different parallelization options.

4.3.2 Rake Fingers and Combiner

The number of rake fingers is a design parameter, which also varies dynamically depending on environment conditions. We designed our implementation to have a maximum of 12 fingers, where the input for the each finger is the delayed input based on offsets calculated by the searcher.

Because each input sample is a complex number, it is separated into two data streams. If we let inR and inI be the real and imaginary input streams, $outR$ and $outI$ be the real

and imaginary output streams, then the descrambler operations is defined as following:

$$outR[i] = cR[i] \cdot inR[i] + cI[i] \cdot inI[i] \tag{4.4}$$

$$outI[i] = cR[i] \cdot inI[i] - cI[i] \cdot inR[i] \tag{4.5}$$

In equation 4.4 and equation 4.5, cR and cI are the predefined descrambling codes. The operations are inherently parallel, and they can be implemented very easily with SIMD operations. However, due to the delay buffering, the input for each descrambler can potentially start for anywhere within one W-CDMA frame. Therefore, the difficulty here is to align the input to the SIMD boundary. If the data is not aligned, then the first and last SIMD operation may not utilize the full 32-wide SIMD lane. Because descrambler operations on W-CDMA slots have 2560 sample points, the misalignment is tolerable in terms of inefficiency.

Despreader down-samples is based on predefined channelization code. Possible down-sampling rates are 4, 8, 16, 32, 64, 128, 256, 512. Because the W-CDMA transmission rate is constant at 3.84M samples per second, despreader's down-sample rate determines the received data rate. To achieve 2Mbps data rate, 3 independent channels of despreader are used, each with down-sample rate equals to 4. The down-sampling is performed by multiplying each sample point with a predefined value of either 1 or -1. In the case of rate 4 down-sampling, every 4 data samples are added together for one output value. In terms of SIMD implementation, because the multiplication operations are with either 1 or -1, it can be implemented as predicated negation operations. The down-sampling operation is implemented using SIMD summation operation. However, because every 4 values are summed together, not all 32 SIMD lanes, SODA's summation operation allows partial summation, where a vector of size 1, 2, 4, 8,and 16 are produced as the output of the

summation. These vectors are buffers, and then read sequentially into the scalar pipeline.

The combiner sums up the data points from each of the rake fingers. Because its computation is relatively small, we did not spend any effort parallelizing this algorithm. Currently, it is implemented using the scalar pipeline only.

4.4 Convolutional Decoder

Convolutional decoders based on the MAP algorithm have superior performance than Viterbi decoder. In our study, we implemented the MAX-Log-MAP algorithm, which is an approximation of the MAP algorithm that operates in the log-domain, allowing multiplications in the original MAP algorithms to be implemented by additions. A complete implementation study on every type of MAP algorithm is beyond the scope of this study. However, the techniques explained in this thesis can be applied in the implementation of other MAP algorithms.

Let s_k be the trellis state values at time k, then the likelihood values at time k, L_k, is defined by:

$$L_k = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \tag{4.6}$$

The first term, $\alpha_{k-1}(s_{k-1})$, is the *alpha* metric that calculates the probability of the current state based on the input values before time k. The second term, $\gamma_k(s_{k-1}, s_k)$, is the branch metric that calculates the probability of the current state transition. The third term is the *beta* metric that calculates the probability of the current state given the future input values after time k. Alpha and beta calculations are defined recursively as

shown below:

$$\alpha_k(s_k) = max(\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)) \quad (4.7)$$

$$\beta_k(s_k) = max(\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})) \quad (4.8)$$

As shown, the alpha computation is forward recursive and beta computation is backward recursive. Let s^1 and s^0 be the 1-branch and 0-branch trellis state transitions. The soft output value, log-likelihood calculation (LLC) at time k, is defined by subtracting the maximum likelihood values of the 1-branch state transitions from the maximum likelihood values of 0-branch state transitions.

$$LLC_k = \max_{s^1}(L_k) - \max_{s^0}(L_k) \quad (4.9)$$

Trellis Computation Implementation. The majority of the convolutional decoder operations are spent on trellis state updates. In this section, we present an efficient implementation of the trellis computation.

Figure 4.4a shows the two types of trellis computation for an 8-state trellis. The blue and red edges correspond to 0- and 1-branch transitions. Figure 4.4b shows the SIMD implementation of the alpha trellis computation. Beta trellis computation is not shown; it follows the same sequence of operations. Trellis computation can be divided into two steps, branch-metric calculation (BMC) and add-compare-select calculation (ACS). In the BMC stage, the inputs are loaded as scalar values from the scalar local memory. The scalar value is then duplicated into a vector using the STV registers. The input vector, In, is correlated with constant metric values, m, to calculate the branch metric values for 0-branch and 1-branch. The correlation function, shown in the figure as M, is defined as $b[i] = In[0] * m[i][0] + In[1] * m[i][1], i : 0 - 7$. Since the metric values $m[i]$ are either 1

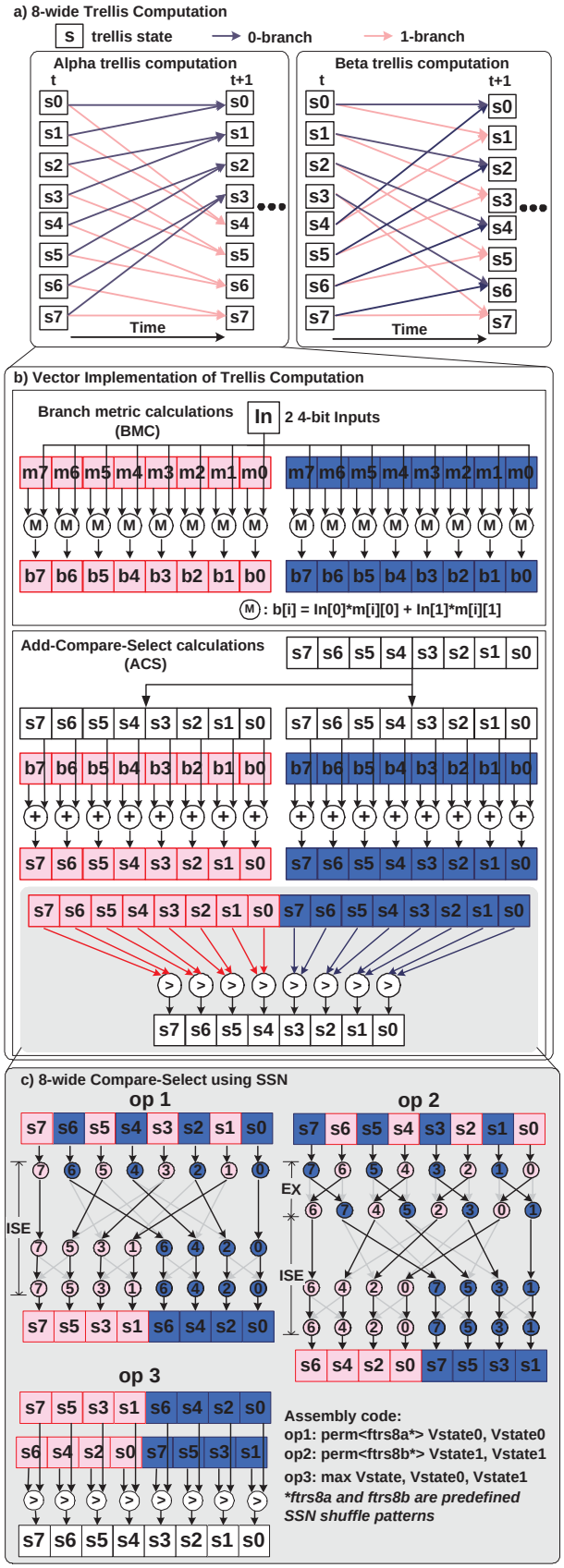


Figure 4.4: Trellis state computation, and SIMD implementation using the SSN

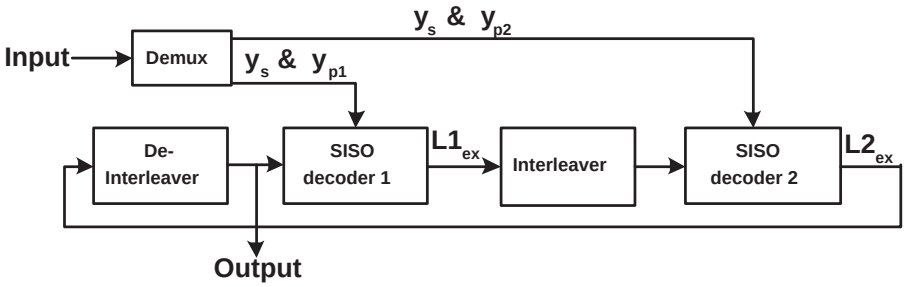


Figure 4.5: Block Diagram of a Turbo Decoder

or -1,we can use predicated add/subtract instructions, where m is stored as a predicate bitvector.

In the ACS step, the trellis state vector, s_t, adds both 0-branch and 1-branch metrics, and compares each pair of values to select the next trellis state vector s_{t+1}. The SSN network is used to rearrange the vectors between SIMD operations. The rearrangement step and the assembly code are shown in Figure 4.4c. Before this compare-and-select step, we first interleave the 0-branch and 1-branch metric values (not shown in the figure). Then two SIMD permutation operations are performed using the SSN. The first permutation operation (op1) takes one cycle, using the ISE (Inverse Shuffle Exchange) pattern. The second permutation operation (op2) takes two cycles, with one additional EX (Exchange only) permutation. Finally, a SIMD compare-and-select operation (op3) is performed to choose the next trellis state values.

4.5 Turbo Decoder

Turbo Decoder Overview. Turbo decoder [14] consists of two component SISO decoders with interleavers between them as shown in Figure 4.5. The observed input sequence, y, is split into two streams and fed into the two component decoders. Both component decoders receive the systematic input y_s, and their respective parity inputs

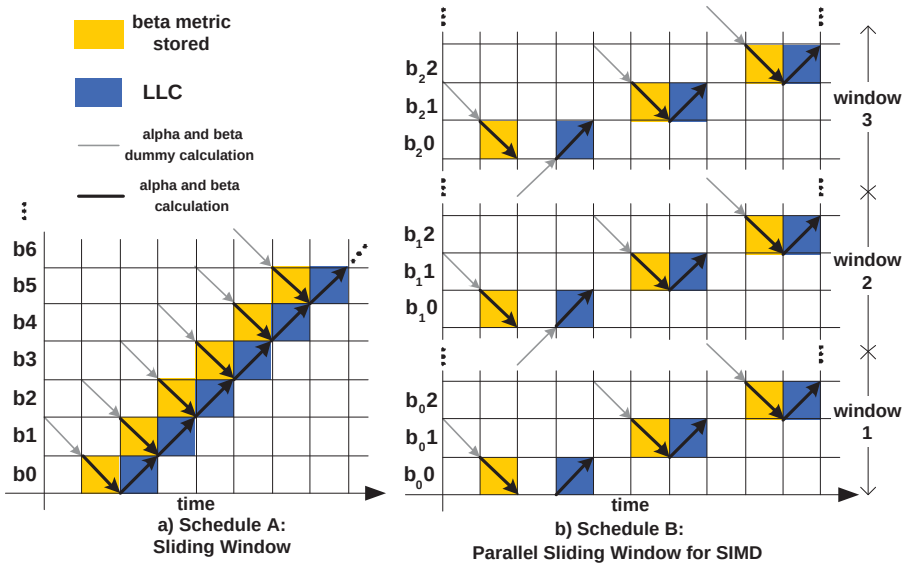


Figure 4.6: Parallel MAX-Log-MAP Scheduling

y_{p1} and y_{p2}. In each iteration, data first goes through the de-interleaver, and is decoded by the first component decoder. The result is then fed into the interleaver, and decoded by the second component decoder, the result of which is fed back into the de-interleaver. The extrinsic outputs from the two SISO decoders are labeled $L1_{EX}$ and $L2_{EX}$. This iterative process is repeated several times until the stopping criteria condition has been satisfied. In this section, we present a software Turbo decoder implementation for W-CDMA: rate $\frac{1}{3}$, K=4 RSC encoder with block interleaving.

Parallel Window Trellis Implementation. In W-CDMA, Turbo decoder uses K=4 MAX-Log-MAP decoder as its component SISO decoder. K=4 MAX-Log-MAP decoder's trellis state size is 8, which under-utilizes the 32-lane SIMD unit. MAX-Log-MAP decoder can be parallelized by dividing the decoding block into smaller sub-blocks, and performing alpha-beta-LLC computations on each sub-block independently. To account for the potential BER performance degradation, additional dummy calculations have to be performed before the alpha and beta computations in each sub-block. Figure 4.6a shows one possible schedule with the alpha, beta, and dummy beta calculations. For

simplicity, the length of the dummy calculations in Figure 4.6 is the same as the number of alpha and beta calculations.

There have been many studies analyzing the trade-offs between different sliding-window and parallel-window scheduling algorithms [58] [83] [18]. Most of these studies assume ASIC architectures with one or more dedicated alpha and beta processors that can execute concurrently. For a software implementation, concurrent execution requires the alpha and beta calculations be expressed as two independent threads. If they are implemented as a single thread, we would have to rely on the compiler to discover independent instructions that can execute in parallel. However, SIMD processors cannot process multiple threads or multiple instructions at the same time. For instance, if the schedule in Figure 4.6a is implemented on our DSP processor, the alpha and beta calculations would be serialized. The parallel sliding window schedule, shown in Figure 4.6b, is better suited for a software implementation on SIMD-based processors. In this schedule, one Turbo decoding block is broken up into N parallel windows. These multiple windows are processed concurrently with the same instruction sequence. Within each window, alpha, beta, and dummy calculations are computed sequentially. Compared to the schedule in Figure 4.6a, this schedule requires N-1 extra dummy alpha calculations to initialize the starting alpha metric for all of the windows except the first one. For W-CDMA Turbo coding, the trellis size is 8, and thus for the 32-wide SIMD processor, 4 windows can be processed in parallel.

Interleaver Implementation. While the SISO decoder computations can be parallelized, interleaving is a data shuffling function that cannot utilize the processing power of the SIMD pipeline. Figure 4.7 shows the computation time of 1 iteration of the Turbo decoder for two processing scenarios. With SISO parallelization, the interleaver underutilizes the processor's resources and limits the overall achievable throughput. In order

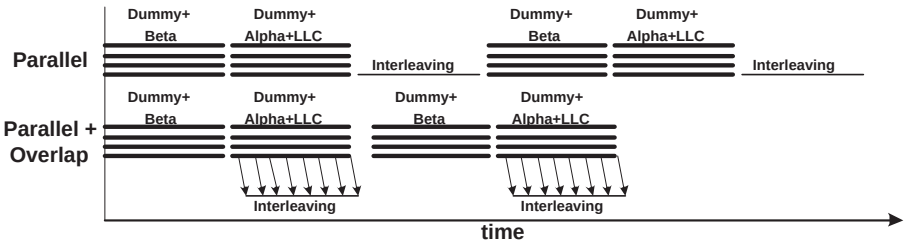


Figure 4.7: Computation time of 1 iteration of Turbo decoding for parallel processing vs. parallel processing with overlapping interleaving

to alleviate this sequential bottleneck, we propose a technique to overlap the interleaving operation in the background of the SISO decoder. This is based on the observation that in a MAX-Log-MAP decoder, output data is produced one element at a time during the LLC computation.

In this method, the interleaving is done during the memory transfer. It requires the DMA controller to be programmed to generate the source and destination addresses for each memory transfer. If the memory transfer rate is faster than the output rate of SISO decoder, then the latency of interleaving can be completely hidden behind the computation.

In the block interleaving specified in W-CDMA, each block element's address can be calculated by adding the row offset and the column offset. In our implementation, this requires 2 additions, 3 memory reads, and 1 memory write, which translates into 9 cycles. The SISO decoder produces an output every 9.25 cycles, enabling us to completely overlap the interleaving latency with the computation latency.

4.5.1 Performance Results

SISO Decoder Throughput Analysis. In this section, we examine the achievable SISO decoding throughput as a function of algorithm specifications and architectural

configurations. Let K be the RSC encoder constraint length, then the size of trellis state is defined as $S = 2^{K-1}$. We assume that the SIMD width, W, is equal or greater than trellis state width: $W \geq S$. To fully utilize the SIMD pipeline, we implemented the parallel sliding window schedule, where N windows are processed in parallel, and $N = \frac{W}{S}$. If we let M be the total number of sub-blocks for one block of Turbo decoding, then each window computes $\frac{M}{N}$ sub-blocks. In the case where $W < S$, trellis computation can still be implemented. The details are omitted due to space limitations.

Let T_{block} be the average number of cycles to compute alpha, beta, LLC, and dummy computations for one sub-block of size L. If we let T_α be the number of cycles to compute one SIMD alpha trellis update, then the latency to compute one alpha trellis update is $\frac{T_\alpha}{N} + 3C_L$, where C_L is the number of cycles to load one scalar value from memory. The SIMD alpha trellis latency is divided by N, because N windows of trellis are computed at once. Three scalar loads are needed for loading two inputs (rate $\frac{1}{2}$ decoding), and one extrinsic value. The SIMD beta trellis updates, T_β, follow the same set of operations as the alpha computation, with three scalar loads. The SIMD LLC computation, T_{LLC}, computes N decoded bits at once. The SIMD dummy trellis computations are also done in groups of N windows, with each window requiring three scalar memory loads. For a sub-block of size L, the dummy alpha and beta calculations have to be done on at least $5K$ (K equals 4 here) elements to stabilize the trellis states and not affect the overall error correction performance. The overall latency T_{block} is shown in Equation 4.10, where T_d is the total dummy computation latency for one sub-block.

$$T_{block} = T_d + L(\frac{T_\alpha + T_\beta + T_{LLC}}{N} + 6C_L) \quad (4.10)$$

$$T_d = 5K(\frac{T_{d\alpha} + T_{d\beta}}{N} + 6C_L) \quad (4.11)$$

As shown in Equation 4.11, T_d is a function of dummy alpha and dummy beta computations. Let $T_{d\alpha}$ be the number of cycles for one SIMD dummy alpha trellis computation, and $T_{d\beta}$ be the number of cycles for one SIMD dummy beta trellis computation. In our implementation, $T_{d\alpha} = 10$ and $T_{d\beta} = 10\frac{N}{M}$. Dummy beta calculations are scaled by $\frac{N}{M}$ because the beta trellis states of the N sliding windows need to be initialized once for every M sub-blocks. In W-CDMA, Turbo decoding block size = ML, and ranges from 320 bits to 5114 bits [41]. Given $L = 100$, the number of sub-blocks, M, varies from 4 to 52. In our throughput calculation, we assume the longest $T_{d\beta}$ latency with $M = 4$, and the total Turbo decoding black size = 400.

In our implementation, $T_\beta = 11$, $T_\alpha + T_{LLC} = 25$, $T_d = 220$ and N, the number of windows processed in parallel, is 4 . Alpha and LLC computations have been grouped together because they are executed together. A scalar load takes 3 cycles, but if we use prefetching instruction, we can shorten it to 1 cycle: $C_L = 1$. Six scalar load operations are required for alpha and beta. The length of one sub-block, L, is 100. Based on the numbers shown above, the overall latency is $T_{block} = 1720$.

Architectural Implications. As shown in Equation 4.10, increasing the number of concurrent sub-blocks, N, decreases cycle count. This can be achieved by increasing the SIMD width W. However, doubling W doubles the size of the processor, which also doubles the power consumption. The other trade-off is the length of a sub-block, L, as longer sub-blocks reduce the relative ratio of dummy calculations per decoded output. However, longer sub-blocks also require more memory to store alpha metric values. The constraint between SIMD memory and sub-block size is $E_v \geq 2WL$, where E_v is the size of local SIMD memory. This means that we should choose the largest sub-block size that can fit in the SIMD memory. Our DSP processor has an 8KB SIMD memory, which holds 128 512-bit entries. With 28 entries reserved for holding spilled temporary register values,

the sub-block size L is chosen to be 100.

Throughput Results. The overall decoding throughput of the Turbo decoder is determined by I times the combined latencies of the 2 SISO decoders and the 2 interleavers, where I is the number of iterations. In our implementation, because the interleaver latencies are hidden, the throughput is only dependent on the SISO decoders' performance. Equation 4.12 shows the Turbo decoder throughput, R_{Turbo}, as a function of the processor's clock speed C_p, the number of Turbo iterations I, the average latency for a SISO decoder to produce one bit of decoding output T_{1bit}, and additional computations for extrinsic value scaling C_M.

$$R_{Turbo} = \frac{C_p}{2I(T_{1bit} + C_M)} \tag{4.12}$$

Because the Turbo decoder is a block decoder, we define SISO decoder latency as $T_{1bit} = \frac{T_{block}}{L} = 17.2$, where T_{block} is the latency for processing one data block of size L. With our SDR processor running at 400MHZ, $C_p = 400M$, and $C_M = 2$, we are able to achieve $1.73Mbps$ and $2.08Mbps$, with $I = 6$ and $I = 5$ respectively. Note that W-CDMA's DCH (Data CHannel) requires a data rate of 2Mbps.

If we wish to achieve higher throughput, we will need to resort to other optimizations techniques. We can scale up the frequency, increase the SIMD width, or map the algorithm onto multiple processors. In particular, our SIMD pipeline has a 32-wide 16-bit datapath, but most Turbo decoder computations only require 8-bit precision. With some extra hardware logic, we can support two 8-bit computation on every 16-bit datapath, making our SIMD pipeline a 64-wide 8-bit datapath. This can potentially double the overall throughput of the SISO decoder. Compiler optimization techniques, such as software pipelining, are also viable options. Finally, our previous study [56] has shown that our

DSP processor consumes approximately 800mW in 180nm technology. Scaling down to 90nm, the same throughput can be achieved with a power consumption of approximately 100mW.

4.5.2 Related Work

There have been numerous studies on parallelizing MAX-Log-MAP for ASICs [58], [83]. Although these studies provide interesting insights into high performance Turbo decoder design, most of these techniques cannot be applied to SDR. The existing software implementations can be separated into two groups. The first group includes implementations on mainstream DSPs, such as TI's C6X that achieves throughput of 286Kbps [26], Motorola's Starcore that achieves throughput of 1.8Mbps [44], and ST-Microelectronics' ST120 that achieves throughput of 540Kbps [60]. The second group includes ASIC and programmable FPGA accelerators for RISC processors. These include the XiRisc processor implementation with a throughput of 270Kbps [73] and Tensilica's Xtensa-based microprocessor with a throughput of 1.48Mbps [32]. Our processor achieves a comparable throughput of 2Mbps for W-CDMA. However, a detailed comparison with prior solutions is difficult because of lack of implementation details.

4.6 Summary

In this chapter, we presented a study on SDR's algorithm designs for SODA. The Turbo decoder, the rake receiver, and the FIR filters are responsible for the majority of the W-CDMA computations. Design and implementation of each algorithm is explained in this chapter. We have shown that algorithm-level optimizations plays an important role in designing an efficient SDR solution.

CHAPTER 5

The ARM Ardbeg SDR Processor

5.1 Introduction

In recent years, we have seen an increase in the number of wireless protocols that are applicable to different types of communication networks. Traditionally, the physical layer of these wireless protocols is implemented with fixed function ASICs. Software Defined Radio (SDR) promises to deliver a cost effective and flexible solution by implementing a wide variety of wireless protocols in software. With the tremendous benefits of SDR, it is likely that many mobile communication devices are going to be supported by SDR technologies in the foreseeable future. There have been tremendous interests in the SDR technology within the high-tech industry. Recently, Samsung was the first to announce a mobile phone that supports TD-SCDMA/HSDPA/GSM/GPRS/EDGE standards using a SDR baseband processor [9].

SODA Processor Architecture. The SODA multi-core architecture was proposed for supporting 3G wireless baseband processing. SODA consists of an ARM control processor, 4 data processing elements (PEs), and a shared global scratchpad memory. Designed for long vector arithmetic operations, each SODA PE includes a wide 512-bit

SIMD unit that is capable of operating on 32 16-bit elements concurrently. In addition, each PE also has a scalar datapath, local scratchpad memories, address generation unit (AGU) and direct memory access (DMA) support.

Ardbeg Processor Architecture. A commercial prototype, Ardbeg, based on SODA has been developed. Ardbeg shares many features with SODA. It is a multi-core architecture, with one control processor and multiple data PEs. Each data PE contains a 512-bit wide SIMD datapath. However, compared with SODA, Ardbeg optimizes the architecture specifically for wireless applications with the addition of algorithmic specific hardware. SODA was designed to test the feasibility of a fully programmable wireless baseband solution by purposely avoiding algorithm-specific designs. The Ardbeg architecture is optimized to achieve higher computational efficiency while maintaining enough flexibility to support multiple protocols. While SODA was focused on supporting 3G wireless protocols, Ardbeg is also designed to scale for future protocols. Overall, Ardbeg achieves between 1.5-7x speedup over SODA for wireless protocols' DSP algorithms while operating at a lower clock frequency.

The evolution of the SODA to Ardbeg was a process with many design choices. Each of the choices contributed to the superior performance of Ardbeg. The major design decisions can be grouped into three categories:

1. **Optimized Wide SIMD Design.** SODA was originally designed in 180nm technology. With 90nm technology targeted for Ardbeg, the SIMD datapath choices need to be re-examined. We re-evaluated the SIMD width, and found that SODA's original 32-lane 512-bit SIMD datapath is still the best SIMD design point in 90nm. On the other hand, the SIMD shuffle network had to be redesigned to support faster vector permutation operations. Compared with SODA's two cycle SIMD multiplier, 90nm technology also allows us to design a single cycle SIMD multiplier, which provides significant speedup for several key SDR algorithms.
2. **LIW Support for Wide SIMD.** For W-CDMA and 802.11a, the SODA SIMD ALU unit is utilized around 30% of the total execution cycles. LIW execution on SODA SIMD pipeline was considered to increase the low utilized SIMD units, but was abandoned due to the concern about the extra power and area costs of adding

more SIMD register file ports. We revisited this concern when designing Ardbeg in order to improve the computational efficiency. The result was Ardbeg issuing two SIMD operations each cycle. Not all combinations of SIMD instructions are allowed. Ardbeg implements a restricted LIW designed to support SDR algorithms' most common parallel execution patterns with minimal hardware overhead. Our analysis shows that having this restricted LIW support would provide better performance and power efficiency over single-issue SIMD datapath, but also that having a multi-issue VLIW does not provide any additional performance benefit over a simple two-issue LIW.

3. **Algorithm Specific Hardware Acceleration.** A set of algorithm specific hardware is also added to the Ardbeg architecture. These include an ASIC accelerator for Turbo decoder; block floating point support; and fused permute and arithmetic operations. This set of algorithm specific hardware was chosen to achieve higher computational efficiency while maintaining enough flexibility to support multiple protocols.

The rest of the chapter is organized as follows. Section 5.2 gives a brief description of the overall architectures of SODA and Ardbeg. Section 5.3 presents the architectural evolution from SODA to Ardbeg. We provide experimental results and analysis to explain the rationale behind the major Ardbeg architectural design decisions. Section 5.4 presents the performance results of the two architectures for various wireless protocols. Section 5.5 provides a survey of the current SDR processor solutions.

5.2 Architectural Overview

Because the majority of the SDR computations are based on vector arithmetics, our previous work on SODA has demonstrated that having a wide SIMD datapath can achieve significant speedup while maintaining low power consumption. With a 32-lane SIMD datapath, SODA was able to achieve an average of 47x speedup for W-CDMA DSP algorithms over a general purpose processor. However, as an initial research prototype, many architectural optimizations were overlooked. Ardbeg has improved upon the base SODA architecture, as will be illustrated in the subsequent sections. This section provides

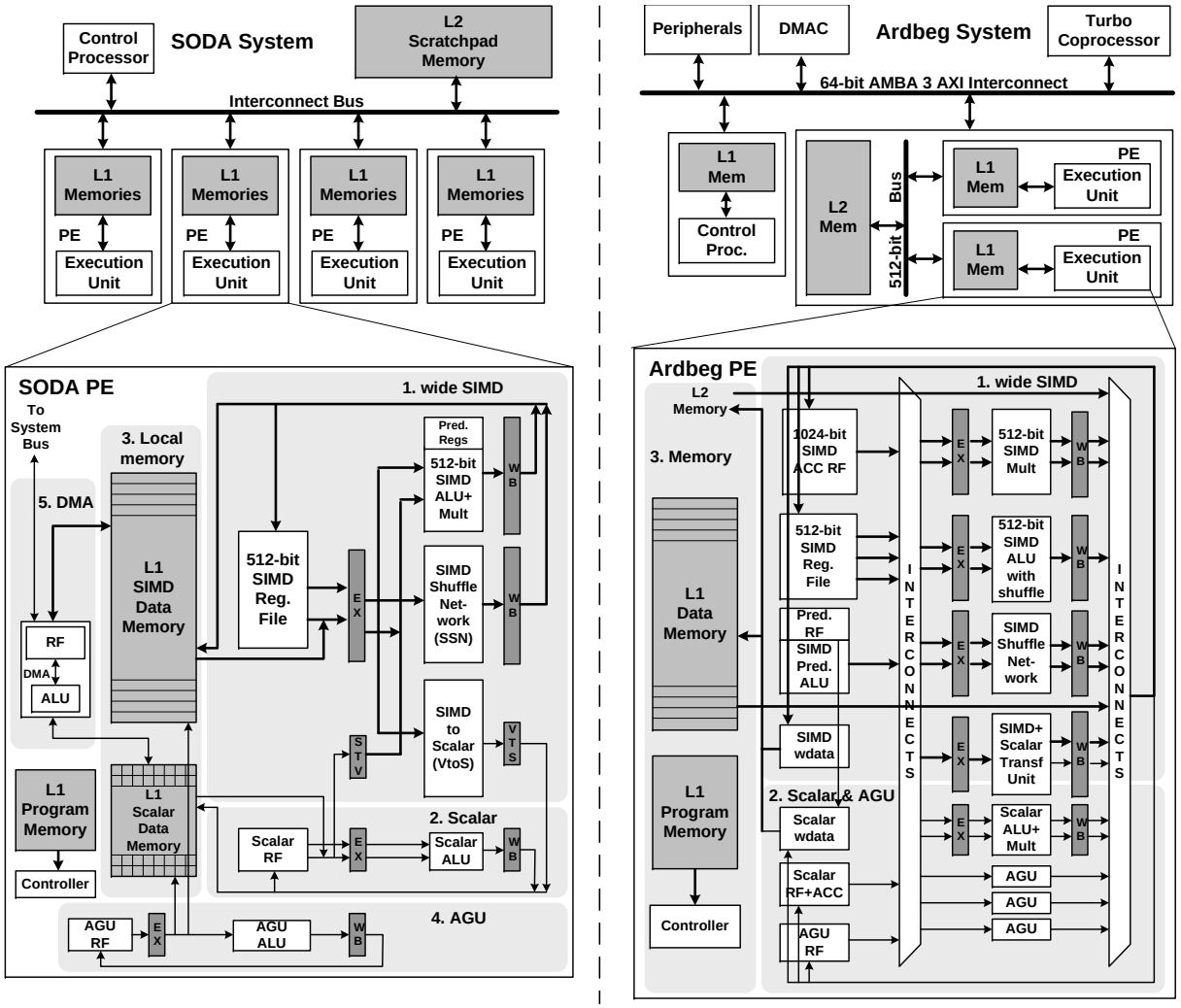
an overview of the SODA and Ardbeg architectures and summarizes the differences.

5.2.1 SODA Architectural Overview

The SODA multicore system is shown on the left in Figure 5.1. It consists of four data PEs, a scalar control processor, and a global L2 scratchpad memory, all connected through a shared bus. Each SODA PE consists of five major components: 1) a SIMD datapath for supporting vector operations; 2) a scalar datapath for sequential operations; 3) two local L1 scratchpad memories for the SIMD pipeline and the scalar pipeline; 4) an AGU pipeline for providing the addresses for local memory access; and 5) a programmable DMA unit to transfer data between memories. The SIMD, scalar and AGU datapaths execute in lock-step, controlled with one program counter (PC).

The SIMD datapath consists of a 32-lane, 16-bit datapath, with 32 arithmetic units working in lock-step. It is designed to handle computationally intensive DSP algorithms. Each datapath includes a 2 read-port, 1 write-port 16 entry register file, and one 16-bit ALU with multiplier. Synthesized in 180nm technology, the multiplier takes two execution cycles when running at the targeted 400MHZ. Intra-processor data movements are supported through the SSN (SIMD Shuffle Network). The SSN consists of a shuffle exchange (SE) network, an inverse shuffle exchange (ISE) network, and a feedback path. Various SIMD permutation patterns require multiple iterations of the SSN network. SIMD-to-scalar (VTS) and scalar-to-SIMD (STV) units are used to transfer data between the SIMD and scalar datapath.

Shortcomings of SODA. Because SODA processor was originally designed in 180nm technology, many SIMD datapath components were pipelined to achieve the target frequency. These include multi-cycle SIMD multiply and permutation operations. Through benchmarks, we found these two operations to be essential in many of the DSP compu-



Comparison summary of the architectural features of SODA and Ardbeg

	SODA	Ardbeg
PE Architecture		
Organization	SIMD + scalar + AGU	SIMD + scalar + AGU
Execution Model	SIMD/Scalar LIW	SIMD/Scalar and SIMD/SIMD LIW
PE Frequency	400MHz (180nm)	350MHz (90nm)
SIMD Architecture		
SIMD Datapath	single issue	ALU + memory + SSN
SIMD Width	512 bits	512 bits
Data Precision	16-bit FXP	8/16/32-bit FXP
Block Floating Point	no	yes
SIMD Predication	yes	yes
SIMD Mult Latency	2 cycles	1 cycle
SIMD Shuffle Network	32-lane 1-stage iterative perfect shuffle	128-lane 7-stage Banyan network
Reduction Network	reduction tree	pair-wise operation
SIMD Reg File	2 read/1 write ports, 16 entries	3 read/2 write ports, 15 entries
L1 Memory	8KB	16KB~64KB
L2 Memory	64KB	256KB~1MB
Others		
Coprocessor	no	Turbo coprocessor
Compiler Opti.	no	software pipelining

Figure 5.1: SODA and Ardbeg architectural diagrams, and a summary of the key architectural features of the two designs.

tations. Memory access latencies are also multi-cycle, but this still holds true in 90nm technology. Because the SIMD datapath can only issue one instruction per cycle, these memory operations stall the entire pipeline. The combination of these inefficiencies results in a relatively poor 30% SIMD ALU utilization. Ardbeg has improved upon these areas in the SODA design. The details are explained in Section 5.3.

5.2.2 Ardbeg Architecture

The Ardbeg system architecture is shown on the right in Figure 5.1. Similar to the SODA architecture, it consists of multiple PEs, an ARM general purpose controller, and a global scratchpad memory. The overall architecture of the Ardbeg PE is also very similar to the SODA PE, with a 512-bit SIMD pipeline, scalar and AGU pipelines, and local memory. Ardbeg was designed using the OptimoDE framework [23]. The framework allowed the creation of custom VLIW-style DSP architectures and evaluating many architectural design trade-offs quickly. These trade-offs will be discussed in the next section. The instruction set for Ardbeg was derived from the NEON extensions [3]. The bottom portion of figure 5.1 also provides a side-by-side comparison between the two architectures.

The Ardbeg system has two PEs, each running at 350MHz in 90nm technology. In addition, it includes an accelerator dedicated to Turbo decoding. In comparison, in the SODA system, Turbo decoding is allocated to one of the four PEs. Both the Ardbeg and SODA PEs have three major functionalities: SIMD, scalar, and AGU.

The SODA and Ardbeg PEs both support 512-bit SIMD operations. The SODA PE only supports 16-bit fixed point operations, whereas the Ardbeg PE also supports 8-, 32-bit fixed point, as well as 16-bit block floating point operations. One of the key differences between Ardbeg and SODA is that the Ardbeg PE supports LIW execution on its SIMD pipeline, allowing different SIMD units to execute in parallel. In the SODA

PE, only one SIMD operation can be issued per cycle. Also, SODA's SIMD permutation network is a single stage, multi-cycle perfect shuffle network, whereas Ardbeg's SIMD permutation network is a 7-stage, single-cycle Banyan network. In terms of number of registers, the Ardbeg PE has additional SIMD and scalar accumulators to hold the output of the multiplier. Ardbeg supports a 1-cycle multiplier, whereas SODA's multiplier requires 2 cycles. A write buffer to memory is also added to Ardbeg. Both Ardbeg's local and global memories are larger than the SODA's memories. In addition, instead of the separate scalar and SIMD memories in SODA, Ardbeg has one unified local scratchpad memory. Because many DSP algorithms don't have much scalar code, it is more efficient to share the memory space between the SIMD and scalar datapath.

5.3 Architectural Evolution From SODA to Ardbeg

5.3.1 Optimized Wide SIMD Design

Since the majority of the SDR algorithms operate on very wide vectors, SODA used a wide SIMD datapath namely, a 512-bit 32-lane SIMD datapath. Ardbeg has also adopted the 512-bit SIMD datapath, and extended it to support 64-lane 8-bit and 16-lane 32-bit SIMD arithmetics. The SIMD shuffle network (SSN) is redesigned to provide better performance at lower power. With a target frequency of 350MHz, implementing Ardbeg in 90nm also allows for a single-cycle SIMD multiplication unit. The rest of this section explains our rationale for these architectural design decisions.

SIMD Width Analysis. The SODA architecture was designed using a 180nm process technology. A 32-lane configuration was found to be the most energy efficient SIMD configuration. One of the first Ardbeg design considerations is to determine if SODA's proposed 32-lane SIMD is still the best configuration in 90nm. In this study, we exam-

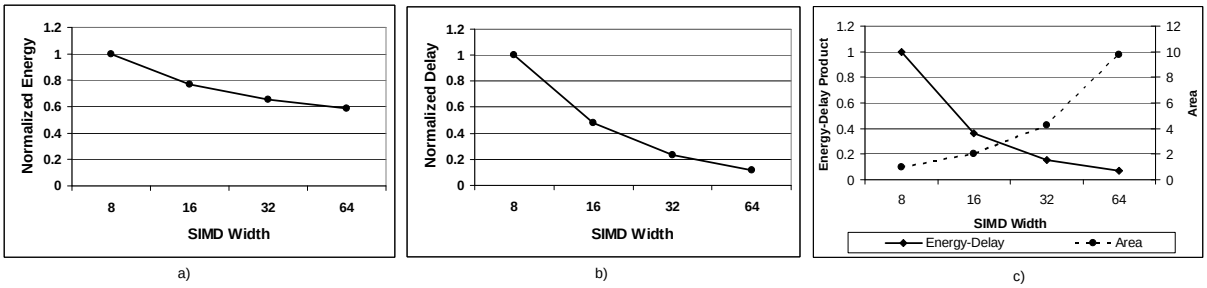


Figure 5.2: Plots of normalized energy, delay, and energy-delay product versus area plots for different Ardbeg SIMD width configurations running 3G Wireless algorithms. The results are normalized to the 8-wide SIMD design.

ine SIMD configurations ranging from 8-lane to 64-lane. Figure 5.2a and b shows the normalized energy and delay for different SIMD width Ardbeg processors synthesized for 350MHz in 90nm for various key SDR algorithms like FFT, FIR, W-CDMA Searcher, and Viterbi. All values are normalized to the 8-wide SIMD configuration.

The figures show that as SIMD width increases, both delay and energy consumption decreases. The delay result is expected as wider SIMD configurations can perform more arithmetic operations per cycle. While power consumption of a wider SIMD is greater, however, because wider SIMD takes less number of cycles to perform the same number of arithmetic operations, the overall energy consumption is lower for wide SIMD. Figure 5.2c shows the energy-delay product and the area of these SIMD configurations. A 32-lane SIMD configuration has better energy and performance results compared to the 8-lane and 16-lane SIMD configurations. A 64-lane SIMD configuration has slightly better results than the 32-lane SIMD configuration. If energy and delay are the only determining factors, then implementing Ardbeg with a 64-lane SIMD configuration is probably the best design choice. However, in a commercial product, area is also a major design factor. As SIMD width increases, area increases at a higher rate than the decrease in either energy or delay. Taking area into account, Ardbeg chose to keep SODA's 32-lane SIMD datapath

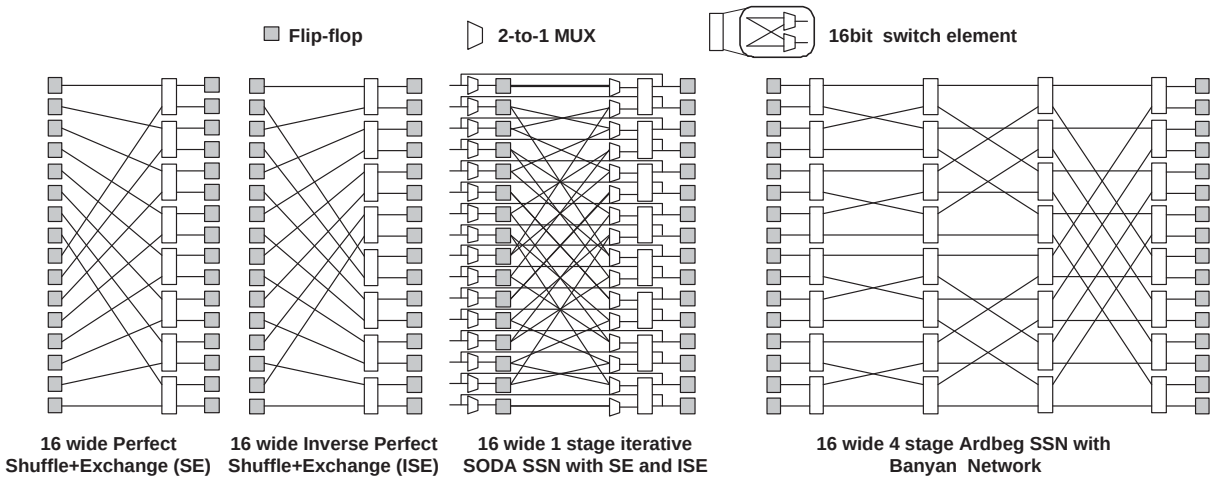


Figure 5.3: SIMD shuffle network for the SODA PE and the Ardbeg PE. For illustration clarity, these examples show 16-wide shuffle networks. The SODA PE has a 32-wide 16-bit 1-stage iterative shuffle network, and the Ardbeg PE has a 128-lane 8-bit 7-stage Banyan shuffle network.

configuration.

SIMD Permutation Support. It is very common for DSP algorithms to rearrange the vector elements before vector computations. One of the central design challenges in designing a wide SIMD architecture is the vector permutation support. A partially connected SIMD shuffle network (SSN) was employed in SODA as shown in Figure 5.3. It is a 32-lane single stage iterative shuffle network consisting of a perfect shuffle and exchange (SE) pattern, a inverse perfect shuffle and exchange (ISE), and a feedback path. The SODA SSN was designed in 180nm technology. Multi-stage networks were considered, but the delay for the multi-stage network was more than one clock cycle running at 400MHz. In addition, there were concerns that the area for a multi-stage network may be too large. Therefore, a multi-cycle iterative shuffle network was chosen for SODA. In designing Ardbeg's shuffle network in 90nm, we evaluated several SIMD configurations and network topologies.

We first examined the performance and energy trade-offs of a wider SSN. Figure 5.4a

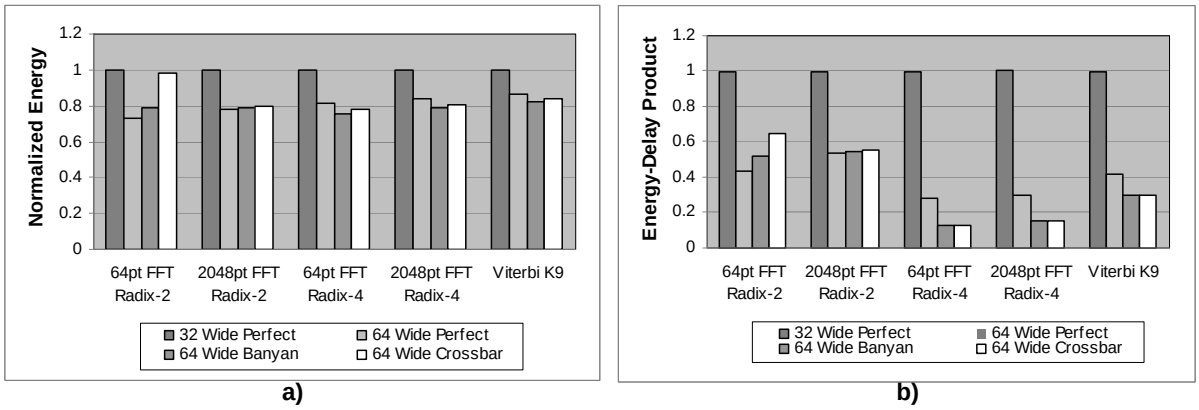


Figure 5.4: Normalized energy and energy-delay product for key SDR algorithms running on Ardbeg for different shuffle network topologies.

provides the normalized energy of key SDR algorithms for a 32-lane SODA SSN and a 64-lane SODA SSN. The SIMD datapath is still 32-lane for both SSN configurations. The 64-lane SSN operates on two 32-lane SIMD vectors by reading from two SIMD register file ports. Filter algorithms are excluded from this study because their SIMD implementations do not use the SSN. Compared to the 32-lane network, a 64-lane network consumes approximately 20% less energy across all benchmarks, despite the fact that the 64-lane network consumes more power than the 32-lane network. This is because these DSP algorithms operate on long vectors, where the vector width is greater than the SIMD width. Because many long vector permutations require extra instructions to store intermediate permutation results, the number of instructions required to perform long vector permutations does not always scale linearly with the width of SSN. A smaller SSN requires higher number of extra instructions than a larger SSN, which results in more frequent SIMD register file accesses and other execution overhead.

We then examined the performance and energy trade-offs of different network topologies. In addition to SODA SSN's iterative SE/ISE network, we also examined 64-lane Banyan network and full crossbar. The SE/ISE and the Banyan networks are shown in

Figure 5.3. The Banyan network is a flattened 7-stage network that can perform 64-lane 16-bit vector permutations in a single cycle. Energy and energy-delay products of these three networks are shown in Figure 5.4. For radix-2 FFT, a 64-lane iterative SE/ISE network is slightly better than a 64-lane Banyan network, because there exists an implementation of this algorithm that is optimized specifically for the SE/ISE network. However, if an algorithm requires more complex permutation patterns, such as the radix-4 FFT and Viterbi algorithms, the single-cycle Banyan network has shorter delays than the multi-cycle iterative shuffle network. Though the difference in energy consumption between the iterative SE/ISE network and 64-lane Banyan is not very large, Figure 5.4b shows that the single-cycle Banyan network has better energy-delay product than the iterative SE/ISE network. Overall, the Banyan network performs as well as the full crossbar, and with $\tilde{1}$7x area saving compared to the crossbar. Therefore, Ardbeg's SSN is implemented with the Banyan network. In addition to supporting 16-bit permutations, Ardbeg's Banyan network can also support 32-lane 32-bit and 128-lane 8-bit vector permutations.

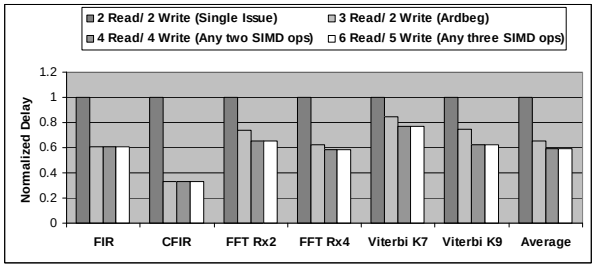
Reduced Latency Functional Units. In SODA, the 180nm process technology put a constraint on the latency of the functional units. Because SODA's target frequency was set to 400 Mhz, the multiplier had to be designed with a 2-cycle latency. For Ardbeg, the target frequency is set at 350 Mhz due to the control latency for controlling the LIW pipeline. With 90nm process technology, Ardbeg implements power efficient SIMD multipliers with single cycle latency. Because many DSP algorithms require large number of multiplication operations, the single-cycle SIMD multiplication results in up to 2x performance improvements.

Ardbeg Function Units	# of SIMD RF Ports Required
Memory Load/Store	1 read / 1 write
SIMD Arithmetic	2 read / 1 write
SIMD Multiply	2 read / No write (ACC RF)
SIMD Shuffle	2 read / 2 write
SIMD+Scalar Transfer Unit	1 read / 1 write
ACC-to-SIMD Move	1 read / 2 write
SIMD Comparison	2 read / 1 write (Pred. RF)

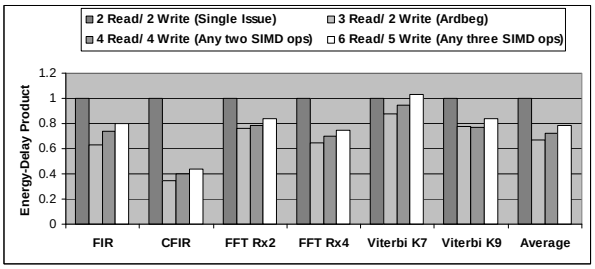
a) This table lists the function units in Ardbeg, and the number of SIMD register file ports required for each unit. At most two SIMD operations can be issued every cycle.

	Mem.	Arith.	Mult.	Shuffle	Trans.	Move	Comp.
Mem.	NA	--	--	--	--	--	--
Arith.	High	NA	--	--	--	--	--
Mult.	High	Mid	NA	--	--	--	--
Shuffle	Low	High	Mid	NA	--	--	--
Trans.	High	Mid	High	Mid	NA	--	--
Move	Low	Low	High	Low	Low	NA	--
Comp.	Low	Low	Low	Low	Low	Low	NA

b) Shaded box means Ardbeg can issue instructions on these two function units in the same cycle. "High/Mid/Low" represent the relative usage frequency for each pair of function units within wireless protocols.



c) Normalized delay for various key SDR kernels running on Ardbeg with different VLIW configurations.



d) Normalized energy-delay product for various key SDR kernels running on Ardbeg with different VLIW configurations

Figure 5.5: Ardbeg VLIW support. Ardbeg has 7 different function units, as listed in sub-figure a. These seven function units share 3 SIMD register file read and 2 write ports. At most two SIMD operations can be issued per cycles, and not all combinations of SIMD operations are supported. Different LIW configurations are evaluated in terms of delay and energy-delay product, as shown in sub-figure c and d. The results are shown for software pipelined Ardbeg assembly code.

5.3.2 LIW SIMD Execution

For W-CDMA and 802.11a, the SODA SIMD ALU unit is utilized around 30% of the total time. The poor utilization is mainly due to the fact that SODA's SIMD datapath is shared with the memory access unit and the SSN. Not being able to utilize the functional units increase register file accesses and also execution time. LIW execution on the SIMD pipeline was considered for the SODA architecture to reduce these problems, but was abandoned due to the concern about the extra power and area costs of adding more SIMD register file ports. In SODA, the SIMD register file was the largest power consumer, accounting for approximately 30% of the total power. When designing Ardbeg, we re-evaluated LIW execution to decrease execution time. We investigated the possibility of including LIW execution as a mechanism to reduce register file power.

To determine the effectiveness of LIW we analyzed different kernels within the set of wireless protocols and found how often functional units could be scheduled together. There are 7 SIMD function units in Ardbeg's SIMD datapath as listed in Figure 5.5a, along with their register port requirements. The values listed in Figure 5.5b represent the frequency that the functional units could be scheduled together. We can see that there are few instruction combinations that occur in high frequency in the algorithms. This suggests that we could implement LIW and potentially reduce the number of register file ports in order to save power while increasing throughput.

We have studied the performance and energy efficiency trade-offs for supporting various LIW configurations in Ardbeg. We examined configurations with different different number of SIMD register file read and write ports: single issue with 2 read and 2 write ports, restricted 2-issue LIW support with 3 read and 2 write ports, full 2-issue LIW support with 4 read and 4 write ports, and full 3-issue instruction LIW support with 6 read and 5 write ports. The performance and energy efficiency results are shown in Figure 5.5c and d. The results are shown for software pipelined scheduled code. The performance is normalized to the cycle count for the single issue Ardbeg. We found that LIW support is beneficial for many key SDR algorithms. This indicates that there is still instruction-level parallelism within SIMDized Ardbeg assembly code. However, we also find that a 2-issue LIW configuration is enough to capture the majority of the instruction-level parallelism, as a 2-issue configuration results in a similar speedup as 3-issue configuration. This is because a significant portion of the instruction-level parallelism is already exploited through SIMD execution. Also, many SIMD operations cannot execute in parallel simply because of data dependencies between these operations.

LIW execution is supported in Ardbeg, but with restrictions on the combinations of instructions that can be issued in a cycle. This results in slower speedup than a full 2-issue

LIW, but provide better energy-delay product due to lesser number of SIMD register file ports. The set of valid Ardbeg LIW instruction combinations are shown in Figure 5.5b as shaded boxes. Among these LIW combinations, overlapping memory accesses with SIMD computation is the most beneficial because most DSP algorithms are streaming. The SIMD arithmetic/multiplication with SIMD-scalar transfer combination is most beneficial for filter-based algorithms. And, the SIMD multiply with move combination is most beneficial for FFT-based algorithms. The responsibility is left to the compiler to produce valid instruction schedules that can utilize this capability. Overall, Ardbeg's SIMD datapath can achieve an average of 60% SIMD ALU utilization while supporting only a subset of LIW execution.

5.3.3 Application Specific Hardware Acceleration

Designing an application specific processor for SDR is a balancing act between programmability and performance. A processor must be flexible enough to support a multitude of wireless protocols. However, too much flexibility results in an inefficient architecture that is unable to meet the stringent performance and power requirements. SODA was designed to meet the throughput requirements of 3G wireless protocols, such as W-CDMA and 802.11a. In addition to these 3G protocols, Ardbeg also designed with future wireless protocols in mind. Therefore, hardware accelerators were added to Ardbeg to increase computational and energy efficiency.

Turbo Coprocessor. Turbo decoding is one of the error correction algorithms used in the W-CDMA wireless protocol for the 2 Mbps data communication channel. It is the most computationally intensive W-CDMA DSP algorithm. In addition, it is the most difficult algorithm to vectorize. Unlike the wide vector arithmetics of other SDR algorithms, W-CDMA Turbo decoder operates on narrow 8-wide vectors. Parallelization

techniques can be applied to utilize the 32-lane SIMD datapath by processing four 8-wide vectors concurrently [55]. However, this requires concurrent memory accesses for the 4 vectors. Because the SODA and Ardbeg PEs only have one memory port, serialized memory accesses become the bottleneck of the algorithm. Software pipelining cannot help, because the main loop in the decoder has data dependencies between consecutive loop iterations. The combination of these factors makes Turbo decoder the slowest algorithm on the SODA and Ardbeg PEs. The SODA and Ardbeg PEs can sustain 50-400 Mbps of data throughput for various FIR and FFT algorithms, but only 2 Mbps for Turbo decoder. The SODA PE was targeted at 400 MHz because of the computational requirements of the Turbo decoder. Offloading the Turbo decoder to a coprocessor allows the Ardbeg PE to lower the target frequency to 350 MHz.

Because of the high computational requirements, one SODA PE is dedicated solely for Turbo decoding, accounting for roughly 25% of the total power consumption. In 90nm implementation, a SODA PE would be able to maintain 2 Mbps while consuming an estimated power of 111mW. In contrast, in 130nm, an ASIC Turbo decoder is able to support 13.44 Mbps while consuming 262 mW [75]. In 90nm technology, this roughly translates to 21 mW for sustaining 2 Mbps throughput. Therefore, in the case of Turbo decoder, the cost of programmability is approximately 5x in terms of power consumption. Furthermore, since 2 Mbps is the maximum throughput for a SODA PE running at 400 Mhz, higher decoding throughput, as required by 3.9G, would require either higher frequencies or multiple PEs. Both these considerations led Ardbeg to offload Turbo decoding on a coprocessor. Other DSP systems aimed at wireless communications, such as the Phillips' EVP, have also taken a similar approach.

5.3.3.1 Application Specific Instruction Set Extensions

Many wireless protocols can share the same error correction ASIC accelerator, but the approach of using more ASIC accelerators is not viable due to the inherent differences in the protocols. However, while the algorithms are different, they share many commonalities within their basic computational blocks. This allows us to increase computational efficiency by adding algorithm-specific instructions common to many algorithms.

Block Floating Point Support. Large point FFTs are used in many wireless protocols. Even though the input and output data are 16-bit numbers, the intermediate results often require higher precision. Block floating point (BFP) provides near floating point precision without its high power and area costs. In floating point, each number has its own mantissa and the exponent. In BFP, each number has its own mantissa, but the exponent is shared between a block of numbers. BFP is commonly used in ASIC design, but very few programmable processors have provided direct hardware support. A key operation in BFP is finding the maximum value among the block of numbers. Most DSP processors support this operation in software. However, for the 32-lane Ardbeg SIMD datapath, this is very inefficient in software, as all lane values must be compared. In Ardbeg, a special instruction is implemented that finds the maximum value in a 32-lane 16-bit SIMD vector. BFP support allows the Ardbeg PE to operate in the 16-bit SIMD datapath mode for FFT computations, instead of the 32-bit SIMD datapath mode that would have been required to satisfy precision requirements. Though FFT is where BFP is currently used, any algorithm which requires higher precision can utilize the BFP instruction extensions.

Fused Permute-and-ALU Operations. It is common in many DSP algorithms to first permute the vectors before performing arithmetic operations. An example is the butterfly operation in FFT, where vectors are first shuffled in a butterfly pattern before

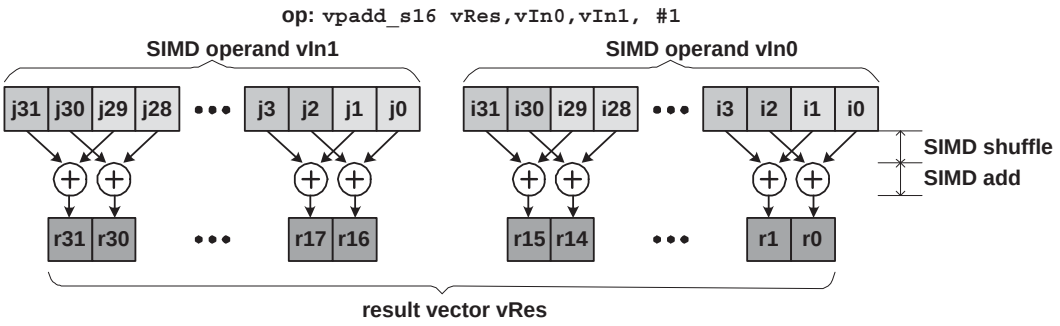


Figure 5.6: Ardbeg's pair-wise butterfly SIMD operation implemented using a fused permute and ALU operation. The figure shows pairs of 2-element butterfly. Ardbeg supports pairs of 1-,2-,4-,8-,and 16-element butterfly of 8- and 16-bits. This butterfly uses the inverse perfect shuffle pattern because the input to each SIMD ALU lane must come from the 2 inputs of the same SIMD lane.

vector adds and subtracts are performed. In an earlier design of the SODA PE, the SSN was placed in front of the SIMD ALU, so that permute-and-arithmetic operations could be performed in one instruction. However, arithmetic operations that do not require permutations always go through the SSN, increasing the number of pipeline stages and power consumption. So in the final SODA PE design, the SSN was taken out of the arithmetic pipeline, and placed as a separate unit, as shown in Figure 5.1. To support the permute-and-arithmetic operations, a separate permutation operation was needed. The result of this permutation operation is written back to the SIMD register file, only to be read out in the next cycle for the arithmetic operation, thereby increasing register file access power in SODA.

The Ardbeg PE addresses this problem by including two shuffle networks. The 128-lane SSN is a separate unit that can support many different permutation patterns. In addition, a smaller 1024-bit 1-stage shuffle network is included in the same pipeline stage in front of the SIMD ALU. This 1-stage shuffle network only supports inverse perfect shuffle patterns between different groups of lanes. This shuffle pattern implements the various pair-wise butterfly operations shown in Figure 5.6. In the figure, the shuffle and

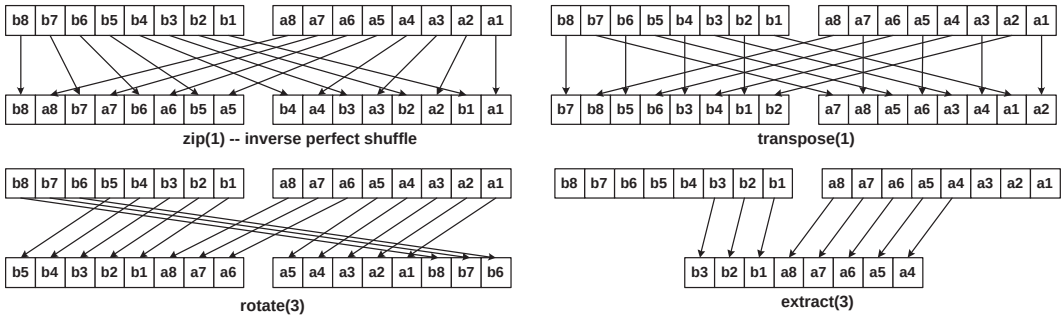


Figure 5.7: SSN shuffling patterns used for matrix transpose.

add operations are performed in the same cycle. This shuffle network is used to accelerate FFT and various other algorithms that use butterfly-and-addition operations. Because these fused butterfly operations are the majority of the permute-and-arithmetic patterns, Ardbeg is able to benefit from the best of both designs. A 2048-Point FFT is able to gain 25% speedup using fused butterfly operations.

SIMD Support for Interleaving. Interleavers are very common in wireless protocols. They are used to protect the transmission against burst errors by rearranging the data sequence. Unlike most other DSP algorithms, there is no data processing or computations involved in interleaving; interleavers simply rearrange the data sequence in different patterns to account for varying types of transmission environments.

Interleaving is essentially a long vector permutation operation, where the vector width is far greater than the SIMD width. This is a challenge because the SODA and Ardbeg's SSN can only permute vector patterns of SIMD width. If we let N be the size of the vector, then a general purpose permutation algorithm would take $O(N)$ time. However, for certain permutation patterns, different types of SIMD shuffle patterns can be utilized to speed up the permutation latency. The Ardbeg SSN supports a set of predefined permutation patterns for efficient implementation of certain interleaving patterns. For example, one commonly used interleaver is the matrix transpose operation, where the

input vector is organized in a $M \times N$ matrix, and the output vector is transposed into a $N \times M$ matrix. A $O(log(N))$ algorithm exists that uses the zip, transpose, extract, and rotate shuffling patterns [2] as shown in Figure 5.7. Using these predefined patterns, a 192 element vector can be transposed in just 37 cycles. This translates to an average speedup of 4x for interleaving kernels for Ardbeg in comparison with SODA.

5.3.4 Hardware Support for Multi-core Scheduling

Interrupt handling in DSP systems can have long interrupt latencies. Interrupt latency is the time from when an interrupt is triggered to when device generating the interrupt is serviced. The reason typical DSP systems have long latencies is because the time it takes for the control processor to see the interrupt and also the time to allow the pipeline and all requests to finish before the interrupt can be handled. In DSP systems where a control processor is present there is not only interrupt latency overhead but also the overhead to determine the next activity to process. This control processor not only has to react to the interrupt but also determine what next to run in the scheduler.

To implement wireless DSP systems we used a simple co-operating multitasking thread model. This model consists of two priority level of threads: High priority threads from interrupts and Deferred Procedure Calls (DPC). High priority threads run until they are blocked or finish and are called from interrupts. DPCs are long running background tasks and can be pre-empted. The problem is that all these are running on the control processor. When a thread blocks or finishes or a DE task completes, the control processor will be interrupted and the time it takes to determine what to do can be a large overhead. If we consider the interrupt frequency of a DVB system due to interrupts from ADC and DE execution completion, the majority of interrupts occur within 8192 cycles and almost 35% of all interrupts occur under 2048 cycles. For DVB, the interrupt overhead is almost

	W-CDMA	802.11a	DVB
Throughput	Voice: 12Kbps Data: 384Kbps/2Mbps	24Mbps, 54Mbps	5Mbps, 15Mbps
Filtering	Complex FIR 65-taps	FIR 33-taps	FIR 16-taps
Modulation	Scrambler/Descrambler Spreader/Despreader Combiner	FFT/IFFT 64 points QAM/IQAM 64 points	FFT 2048 points Scrambler/Descrambler QAM/IQAM 4/16/64 points
Synchronization	Searcher	Interpolator	Equalizer Channel Est.
Error Correction	Interleaver Viterbi K=9 Turbo Decoder K=4	Interleaver Viterbi K=7	Bit Interleaver Viterbi K=7

Figure 5.8: DSP algorithms that are used in W-CDMA, 802.11a and DVB wireless protocols.

250 cycle. This means that the DE units are wasting anywhere from 3-10% of it's possible utilization just waiting for a signal to start the next task. In Ardbeg there are 2 DE's so that is a potential of 6-20% performance reduction.

To reduce this overhead and provide faster interrupt response a simple hardware sequencer is added. The function of this hardware sequencer is to handle task queuing, where simple activity can happen without control processor intervention. This will allow for dramatically lower latencies. The control processor loads the event data of the next event that the DE will perform into the DE's task buffer. When the DE interrupts because of completion it will also check the task buffer to see if it can start the next task. If there is data it will start without a response from the control processor. This allows the DE to start the next task and allow the control processor to handle the interrupt without wasting the DE's processing time.

5.4 Results and Analysis

For the overall protocol performance evaluations, we have implemented three different wireless communication protocols which represent a wide spectrum of wireless communi-

cation applications. These are W-CDMA [41], 802.11a [1], and DVB-H [5]. W-CDMA is a widely used in 3G cellular protocols. 802.11a is chosen to represent the workload of a typical WiFi wireless protocol. DVB-H (Digital Video Broadcasting - Handheld) is a standard used for digital television broadcasting for handheld receivers. These protocols are chosen to stress the flexibility of the SODA and Ardbeg systems. Both SODA and Ardbeg are able to support real-time computations for these protocols.

The characteristics of these three protocols are listed in Figure 5.8. These protocols consist of the following four major algorithm categories: filtering, modulation, synchronization, and error correction. Filtering is used to suppress signals transmitted outside of the allowed frequency band so that interference with other frequency bands are minimized. Modulation algorithms translate digital signals into analog wave patterns consisting of orthogonal signals. Synchronization algorithms synchronize the two communicating terminals to ensure lock-step communication between the sender and the receiver. Error correction algorithms are used to recover data from noisy communication channels.

The RTL Verilog model of the SODA processor was synthesized in TSMC 180nm technology. The estimated power and area results for 90nm technology were calculated using a quadratic scaling factor based on Predictive Technology Model (PTM) [6]. The Ardbeg processor was developed as part of the OptimoDE framework [23]. The architectural model is written in OptimoDE's hardware description language. A Verilog RTL model, a cycle-accurate simulator, and a compiler are generated by OptimoDE. The Ardbeg processor was synthesized using Synopsys physical compiler to place and Cadence Encounter to route with clock tree insertion. Ardbeg's PE area is 75% larger than SODA's estimated 90nm PE area. The total system area is comparable between the two systems because SODA contains 4 PE's compared to Ardbeg's 2 PE's. Ardbeg was targeted for 350 MHz while SODA was targeted for 400 MHz.

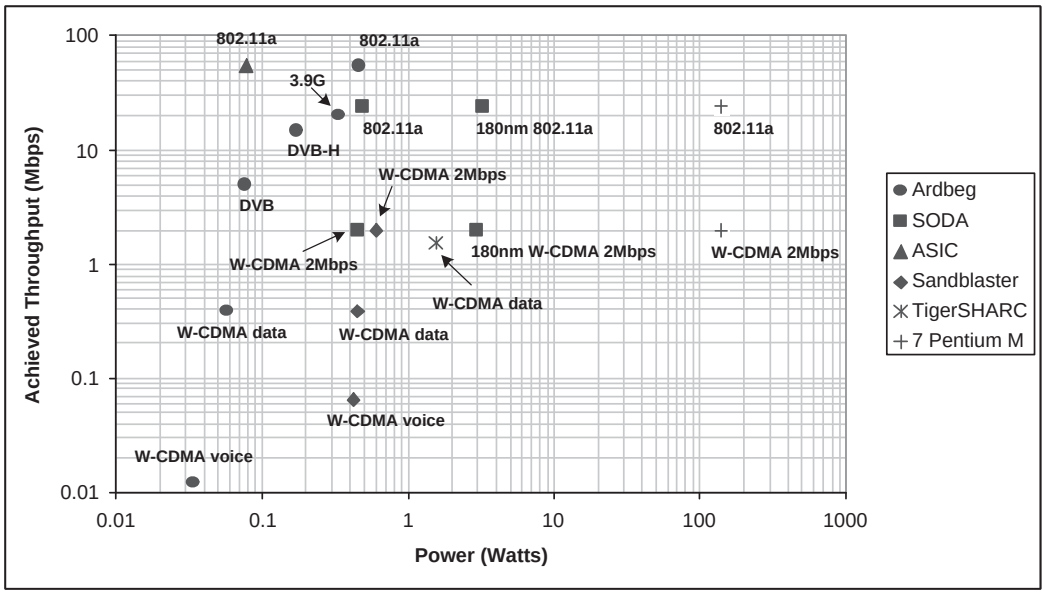


Figure 5.9: Throughput and power achieved for SODA and Ardbeg for W-CDMA, 802.11a and DVB. ASIC 802.11a, Pentium M, Sandblaster, and ADI TigerSharc results are also included for comparison purposes. Results are shown for processors implemented in 90nm, unless stated otherwise.

5.4.1 Wireless Protocols Results

Evaluation results show that an Ardbeg multicore system synthesized in 90nm technology is able to support 3G wireless processing within the 500 mW power budget of a mobile device [62]. Figure 5.9 shows the power consumption required to achieve the throughput requirement of W-CDMA, 802.11a, and DVB. The graph includes the numbers for the SODA and Ardbeg systems, as well as an ASIC implementation for 802.11a, Sandbridge's Sandblaster, Analog Devices TigerSHARC, and Pentium M implementations. General purpose processors, such as Pentium M, require a power consumption two orders of magnitude greater than the 500 mW power budget. On the other end of the spectrum, an ASIC solution is still 5x more power efficient than any SDR solution. Overall, Ardbeg is more power efficient than SODA for all three wireless protocols. Because Ardbeg is designed to handle high-throughput wireless protocols, its performance for low-throughput

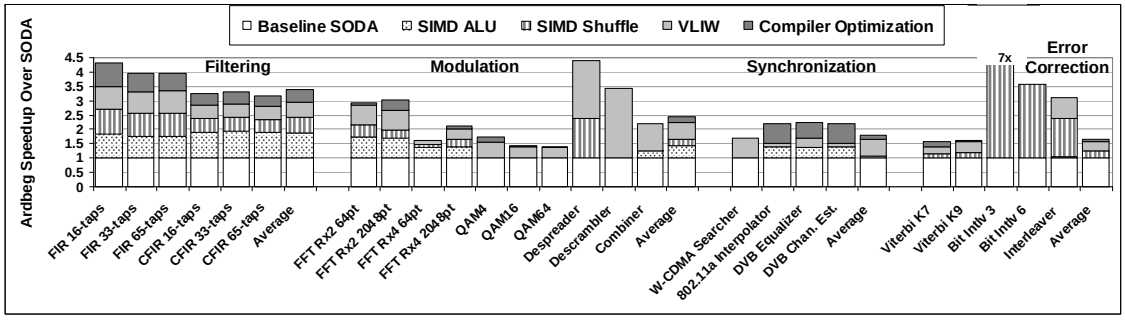


Figure 5.10: Ardbeg speedup over SODA for the key DSP algorithms used in our wireless protocol benchmarks. The speedup is broken down into the different architectural optimizations. These include optimized SIMD ALU, wider 1-cycle SIMD shuffle network, reduced SIMD memory latencies through LIW execution, and compiler optimizations with software pipelining.

W-CDMA voice channels is not as efficient. Both SODA and Ardbeg are very competitive compared to other SDR processors, including Sandbridge's Sandblaster and Analog Devices' TigerSHARC. The major sources of Ardbeg's efficiency are: the restricted LIW execution, application specific instruction set extensions, and larger shuffle network.

5.4.2 Wireless Algorithms Analysis

In this section, we present a performance analysis of the key DSP algorithms in each of the four algorithm categories: filtering, modulation, synchronization, and error correction. Details of the kernels can be found in [52].The speedups are consolidated in Figure 5.10. The speedup analysis is further broken up into the Ardbeg architectural improvements that were highlighted in the Section 5.3. These improvements include: optimized SIMD ALU, wider single cycle SIMD shuffle network, and LIW execution. The OptimoDE framework used to design Ardbeg generates a compiler which performs optimizations like software pipelining and other compiler optimizations which we also report.

Filtering. Finite Impulse Response (FIR) filters are widely used in wireless communication protocols. Both the SODA and Ardbeg PEs can support the computation

requirements of filters for real-time 3G wireless protocol processing. Figure 5.10 shows the Ardbeg PE's speedup over the SODA PE for various filter configurations. On average, Ardbeg achieved a 3.4x speedup over SODA.

Multiply-and-accumulate (MAC) operations are the central arithmetic operation for filtering. For complex filter arithmetics, multiplications are even more important as every complex multiplication requires four MAC operations. The SODA PE has a two cycle multiplier (180nm), whereas the Ardbeg PE has a single cycle multiplier (90nm). A significant portion of Ardbeg's speedup is due to the faster multiplier.

In this analysis, both SODA and Ardbeg implement a vectorized version that requires one 64-wide SIMD vector permutation operation for processing each sample point. The SODA PE only has a 32-wide SIMD permutation network, compared to the Ardbeg's 64-wide network. The permutation operation takes 3 cycles on SODA, but only one cycle on Ardbeg. Because memory is accessed for each sample, LIW support on the Ardbeg PE is able to hide the multi-cycle memory latencies. Finally, software pipelining and other compiler optimizations help better utilize the Ardbeg's LIW datapath.

Modulation. Fast Fourier Transform (FFT) is widely used in OFDM protocols like 802.11a/g and DVB. Figure 5.10 shows the Ardbeg PE's speedup over the SODA PE for various FFT configurations. On average, Ardbeg achieves a 2.5x speedup over SODA. Like the filters, there is about a 50% speedup attributed to single cycle multiplies. This speedup is less for a Radix-4 implementation because multiplications are reduced by 25%. Another 50-100% speedup is attributed to the fused operations. The butterfly operation is implemented efficiently by fusing multiplication with add or subtract operations. Another benefit is that Ardbeg allows specialized shuffle operations, followed by ALU operations to be computed in one cycle. Finally, the LIW scheduling provides the remaining speedup. By allowing overlapped memory operations, Ardbeg can overlap the memory loads of the

next butterfly with the current butterfly's operation.

Modulation in W-CDMA consists of three kernels: descrambler, despreader, and combiner. The despreader gains significant speedup (almost half) by utilizing Ardbeg's wide shuffle network. The descrambler implementation on Ardbeg is a direct translation of the SODA version. Ardbeg gains, because in every cycle, it can overlap the memory and ALU operations. The combiner, like the despreader and descrambler, benefits from the LIW scheduling as well as the 1 cycle multiplication. All three kernels benefit greatly from LIW scheduling because each iteration of the inter-loop of these kernels are small and independent. This allows the overlap of memory loads and stores, shuffle operations, and ALU operations in the same cycle.

Synchronization. Synchronization in W-CDMA is accomplished by the searcher, which achieves almost 1.5x speedup on Ardbeg versus SODA. The gain in performance due to Ardbeg's pipelined memories and LIW scheduling is offset by performance loss due to its SIMD predicate support. The number of instructions needed to calculate the predicate values on the Ardbeg PE is 4 cycles, whereas the SODA PE can perform the same task in 2 cycles. This is because SODA's predicate values are stored in the SIMD register file, whereas Ardbeg's predicate values are stored in a dedicated register file. Although Ardbeg's dedicated register file is able to compute different predicate patterns more quickly, it takes longer to load the predicate values into the SIMD datapath. Because all of searcher's predicate patterns can be pre-computed, SODA's faster predicate read latency proves to be more beneficial. This accounts for a 20% cycle difference. The major benefit of Ardbeg's LIW scheduling is hiding the memory's multi-cycle access latencies. Because half of every loop iteration can be overlapped, the Ardbeg searcher still results in almost 2X speedup despite its inefficient predication support.

802.11a interpolator, DVB equalizer, and DVB channel estimation are all similar to the

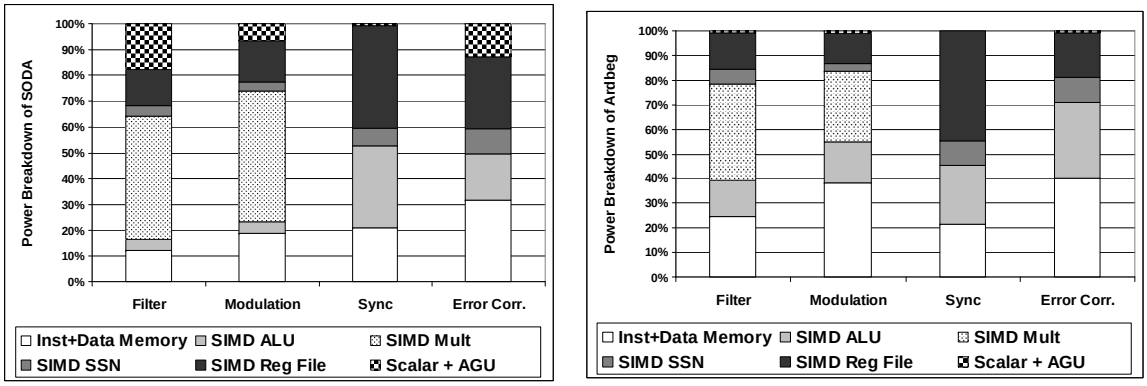
FIR operations, and their speedup rationales are similar to those of the FIR. The only difference is that these algorithms have intra-iteration data dependencies that cannot exploit the LIW datapath. Software pipelining is beneficial by scheduling different loop iterations onto the LIW datapath.

Error Correction. There are two commonly used error correction algorithms in wireless communication – Viterbi and Turbo decoders. As mentioned in the previous section, the Turbo decoder in Ardbeg is offloaded to an accelerator. However, the Viterbi decoder is still implemented by the Ardbeg PE. As shown in Figure 5.10, Ardbeg's Viterbi implementation does not gain significant speedup over the SODA Viterbi implementation, ranging from 1.2x to 1.6x. The reason is because the Viterbi computation does not have multiplication operations, so the optimized SIMD ALU does not help. In addition, there are data dependencies between consecutive loop iterations, so software pipelining techniques do not help. The majority of the speedup comes from hiding the memory access latency through LIW execution on the SIMD pipeline.

Interleavers are also widely used in many wireless protocols. As mentioned in the last section, a few SIMD shuffle patterns are added to accelerate these algorithms. As shown in Figure 5.10, the Ardbeg interleaver implementations gain a significant speedup, up to 7x speedup over SODA's implementation. The speedup is solely due to the Ardbeg's SSN. Because the majority of the interleaver instructions are SIMD permutation operations, Ardbeg's single cycle 64-wide SSN has a significant advantage over SODA's multi-cycle 32-wide SSN.

5.4.3 Wireless Algorithm Power Breakdown

Figure 5.11 shows the power breakdown on Ardbeg and SODA for the key kernels in each wireless kernel category. Because SODA was synthesized in 180nm, and Ardbeg was



(a) SODA algorithm power consumption breakdown

(b) Ardbeg algorithm power consumption breakdown

Figure 5.11: SODA and Ardbeg power consumption breakdown for the four key kernel algorithms. The power consumptions are normalized to their respective total.

synthesized in 90nm, a direct comparison is not possible. Instead, the power consumption values are normalized with respect to each kernel's total power, and a comparison of the normalized power consumption of each architectural component is presented.

Comparing Figures 5.11(a) and (b) leads to several important observations. First, many filtering and modulation algorithms have a large number of multiply operations. The results show that a smaller percentage of Ardbeg's power is spent on SIMD multiply than SODA. Ardbeg's multiplier is custom designed for 90nm running at 350MHz, whereas SODA used a library generated multiplier in 180nm. Second, a higher percentage of Ardbeg's power is generally spent on memory because it has a larger memory than SODA. SODA has a decoupled scalar and SIMD memory structure. It reads values from both the SIMD and scalar memories, relieving the stress on the SIMD memory, but allocating more power the scalar memory. Third, a higher percentage of Ardbeg's power is spent on SIMD ALU power. This is due to the more complex SIMD ALU with support for the permute-and-ALU operations. These fused permute-and-ALU operations speed up the computation, and also reduce the number of SIMD register file accesses. Fourth, the fused permute-and-ALU operations are also the reason why Ardbeg and SODA have

comparable SIMD register file power utilizations for most of the algorithms, even though the Ardbeg PE has more read and write ports. The exception is the synchronization algorithms where fused permute-and-ALU operations are not used and so the register power is comparatively higher.

Filter on SODA shows a much larger power contribution due to the multiplier. This is because the multiplier in Ardbeg was custom built to optimize power and performance. Also, the scalar power is more significant in filter on SODA because of the implementation trades SIMD ALU operations with more scalar operations. In Modulation, there is also a major benefit from the optimized multiplier. In FFT, the majority of operations are multiply which is why an optimized multiplier in Ardbeg allocates less power to the multiplier and more the the ALU. In Synchronization, the code itself in SODA is very similar to Ardbeg. The only difference is on SODA we use the ALU to generate predicates, while Ardbeg loads them from memory. In Error Correction, we see that SODA uses more power in the shuffle network than Ardbeg and the difference is allocated to the Register File Power. This is mainly due to the wide shuffle network in Ardbeg which allows it to perform less shuffle operations than SODA for the interleaver kernel. Viterbi on both implementations are fairly similar, which is why the interleaver dictates the power breakdown.

5.5 DSP Processor Architecture Survey

There has been tremendous interest in SDR in the industry, resulting in a wide range of proposed architectural solutions from many leading semiconductor companies. The proposed SDR solutions can be categorized into two different design philosophies – SIMD-based and reconfigurable architectures, as explained in [68]. SIMD-based architectures

	ARM Ardbeg	SODA	Infineon MuSIC	ADI TigerSharc	Icera DXP	Phillips EVP	Sandbridge Sandblaster
# DSPs	2	4	4	8	NA*	1	4
PE frequency (MHz)	350	400	300	250	1000	300	600
# 16-bit SIMD lanes	32	32	4	2x4	4	16	4
VLIW support on SIMD	restricted	no	yes	yes	yes	yes	yes
Max # of EX stages	1	2	4	2	20	NA*	2
Scalar datapath	yes	yes	no	no	no	yes	no
Hardware coprocessor	yes	no	yes	no	no	yes	no
Scratchpad memory	yes	yes	yes	yes	yes	yes	yes
Shared global memory	yes	yes	yes	no	NA*	no	no

Figure 5.12: Architectural comparison summary between proposed SIMD-based SDR processors. *For the Icera DXP and the Phillips EVP, some of the architectural details are not released to the public at this time.

usually consist of one or few high-performance DSP processors. The DSP processors are usually connected together through a shared bus, and managed through a general purpose control processor. Some SIMD-based architectures also have a shared global memory connected to the bus. Both Ardbeg and SODA fall under the SIMD-based architecture category. Reconfigurable architectures are usually made up of many simpler processing elements (PEs). Depending on the particular design, these PEs range from the fine-grain ALU units to the coarse-grain ASICs. The PEs are usually connected together through a reconfigurable fabric. The rest of this section will present existing design solutions in these two categories.

5.5.1 SIMD-based SDR Processor Architecture

In addition to Ardbeg and SODA, there are several other SIMD-based SDR architectures. These include Infineon's MuSIC [17], Analog Device's TigerSHARC [31], Icera's DXP [45], Phillips's EVP [81], and Sandbridge's Sandblaster [33]. A comparison between these architectures, SODA, and Ardbeg are listed in Figure 5.12. These are all embedded systems that consist of 1 to 8 high performance DSP processors. Because data are accessed

in a regular pattern, all of the processors use software-managed scratchpad data memories instead of caches to reduce power. Even though most of these processors are designed in 90nm technology, they operate at relatively low frequencies to reduce power. The exception is the Icera DXP, which chose to implement a deeply pipelined high frequency design. Its SIMD ALUs are chained so that a sequence of vector arithmetic operations are performed before the data are written back to the register file. This has the advantage of saving register file access power at the cost of a less flexible SIMD datapath.

Most SIMD-based SDR processors support VLIW execution by allowing concurrent memory and SIMD arithmetic operations. Analog Device's TigerSHARC goes one step further, and provides concurrent SIMD arithmetic operations by having two 4-lane SIMD ALU units that are controlled with two instructions. With 32 lanes, Ardbeg and SODA have the widest SIMD design. Wider SIMD datapaths have higher power efficiency, but also require higher levels of data-level parallelism within the software applications. Because the majority of SDR's computation are on wide vector arithmetics, the 32-lane SIMD can be utilized fairly well. In addition, Ardbeg's execution stage is optimized so that any arithmetic operation can finish in one cycle. As we showed in the algorithm analysis, having single cycle ALU provides significant speedup for SDR algorithms. And finally, like Ardbeg, some other commercial solutions also chose to incorporate accelerators for error correction algorithms, including Viterbi and Turbo decoders.

5.5.2 Reconfigurable SDR Processor Architectures

Reconfigurable array based SDR Solutions. Wireless protocols can be broken down into key computational patterns, which can be as fine-grained as a sequence of arithmetic operations, or as coarse-grained as DSP kernels. There have been numerous SDR solutions based on fine-grained computation fabrics. Examples of such solutions

include picoArray [13], and the XiSystem's XiRisc [57]. The XiRisc, also includes a scalar/VLIW processor, with the reconfigurable logic acting as an accelerator. One of the major drawbacks of the fine-grain computation fabrics is the high communication cost of data shuffling within the computation fabrics. The coarse-grained reconfigurable architectures contain a system of heterogeneous coarse-grained processing elements, with each type of PE tailored to a specific DSP algorithm group. Examples include Intel RCA [22], QuickSilver [7] and IMEC ADRES [59]. Both RCA and QuickSilver have 3 or 4 different types of PEs, ranging from simple scalar processors to Application Specific Instruction Processors to serve as Viterbi and Turbo accelerators. These heterogeneous SDR systems provide a trade-off between overall system flexibility and individual kernel computational efficiency. Different wireless protocols require very different types of DSP algorithms and a heterogeneous systems is more-likely to under-utilize their hardware, resulting in less efficient overall system operation.

Heterogeneous MIMD based SDR Solutions. These MIMD styled architectures contain a system of heterogeneous coarse-grained processing elements, with each type of PE tailored to a specific DSP algorithm group. Examples include Intel RCA [22], and QuickSilver [7]. Both RCA and QuickSilver have 3 or 4 different types of PEs, ranging from simple scalar processors to ASIP(Application Specific IP) Viterbi/Turbo accelerators. These heterogeneous SDR systems provide a trade-off between overall system flexibility and individual kernel computational efficiency. While a homogeneous processor system can distribute the system workload among PEs, a heterogeneous processor system must provide enough units for the worst case workloads of each type of PE. W-CDMA and 802.11 require very different types of DSP algorithms. Therefore, heterogeneous systems are more-likely to under-utilize their hardware, resulting in less efficient overall system operations.

5.5.3 VLIW-based DSP Architectures

The TI TMS320C64x DSP processors [4] are highly parallel VLIW machines, that can achieve high performance. However, because data level parallelism is much more prevalent than instruction level parallelism, the benefits offered by VLIW are not utilized in wireless applications. The instruction execution power consumption is relatively higher than other solutions, and thus the overall computational efficiency is lower. These solutions typically cannot meet the SDR performance and power requirements by themselves. Therefore, they often include ASICs accelerators for performance enhancements.

5.5.4 Vector/SIMD based Multi-media Solutions

Vector and SIMD embedded processors have been very popular in the multi-media domain. Among them are the IBM Cell Processor [40], VIRAM Project [46], and Imagine Project [10]. IBM's Cell processor is architecturally similar to our design at the system-level, with a controller (PPE) and multiple SIMD processors (SPE). However, the SPE is a generic SIMD-based processor, whereas ours is a domain-specific VLIW+SIMD processor targeted at SDR. Although the Cell processor has higher overall computational throughput than our processor, it was never designed to be a mobile solution, and its power consumption is 100x greater than the budget for a wireless protocol. Imagine [10] uses SIMD-based execution, where each instruction is a VLIW operation. The VIRAM [46] design is an improved vector processor designed for multi-media workload. Again, these processors were not designed specifically for wireless applications, but instead for general multimedia applications. The SODA architecture is designed specifically for wireless protocols and, as a result, can execute them much more efficiently.

5.6 Summary

Software defined radio promises to revolutionize the wireless communication industry by delivering a low-cost multi-mode baseband processing solution. Ardbeg is a commercial prototype based on SODA. Aspects of the SODA design are kept intact, such as the wide 512-bit SIMD datapath and the coupled scalar and SIMD datapath. Application-specific design trade-offs are made to achieve higher computational efficiency while maintaining enough flexibility to support multiple protocols. The evolution of SODA to Ardbeg focused on three main categories: optimized wide SIMD design, LIW support for wide SIMD and algorithm specific hardware acceleration. The results show that Ardbeg's architectural optimizations allow it to achieve between 1.5-7x speedup over SODA across multiple wireless algorithms.

CHAPTER 6

Language Extensions for Software Defined Radio

6.1 Introduction

Some of the key advantages of Software Defined Radio include flexibility and lower cost. These advantages are based on the assumption that software solutions are easier and more flexible than hardware solution. However, if the users are forced to program SDR processors in machine code, then implementing a software solution is not easier nor more flexible than a hardware solution. Therefore, software tool support is a first-order design consideration in providing a viable SDR solution.

Software development for uniprocessor DSPs is hard, and SoC DSP architectures, such as SODA and Ardbeg processors, make this hard problem even harder. There is a clear need for better language support to help manage the complexity of mapping DSP systems onto DSP hardware. SPEX (Signal Processing EXtension) is a language extension designed to address these problems for embedded streaming DSP systems. It has two design objectives: allowing programmers to express the inherent parallelism within streaming DSP systems, and providing an efficient interface for the compiler to generate code for embedded DSP hardware. It is designed to support all aspects of embedded DSP

computations. This includes dataflow constructs to describe streaming computations, vector arithmetic operations to describe DSP computations, and real-time constructs to describe real-time operations and deadlines. The focus of this chapter is on the dataflow constructs to describe streaming computations. The contribution of this work is the application of using the dataflow constructs to bridge the gap between wireless protocol descriptions and SoC DSP architectures. There has been many existing languages that provide support for vector DSP and real-time operations. These language constructs are summarized at the end of this chapter.

Parameterized Dataflow Computation Model. Dataflow computation models have been proposed to describe streaming computations. However, many streaming DSP systems also have critical control flow operations. In between long episodes of streaming computation, DSP systems intermittently reconfigure their streaming patterns to account for changes from the users and the environments. SPEX is based on the parameterized dataflow (PDF) computation model, where the dataflow is described with a set of parameters. Each parameter is a variable with a finite set of possible values, describing a set of possible dataflow configurations. We propose a three-stage run-time execution model to provide efficient computation on embedded multi-core hardware. During the first stage, a static dataflow is initialized by assigning a constant value to each parameter variable. The second stage is the stream computation using a compiler-generated static synchronous dataflow schedule. The third stage finalizes the stream computation with updates to the dataflow variables and states.

Streaming Communication Model. Although parameterized dataflow model is good for describing reconfigurable streaming computation, its First-In First-Out (FIFO) communication pattern is inadequate for describing DSP system's complex streaming communication patterns. Therefore, we propose a modified pseudo-dataflow computation

model where data can be shared among dataflow actors. Complex streaming patterns can be constructed using these actors as basic building blocks.

SPEX Language Extension. Parameterized computation models have been proposed before for modeling DSP systems [15]. However, given the popularity of existing languages such as C and C++, it is challenging for programmers to adopt a completely new concurrent programming paradigm. SPEX aims to reduce this challenge by implementing the parameterized dataflow model as a language extension to the familiar C language syntax. To provide efficient code generation for embedded DSP architectures, we also find that some of C's language features cannot be supported or must adopt different semantic meanings consequently. Even though SPEX is applied to C in this study, it is general enough to be applied to any programming language. In fact, we also have previously published a paper on applying SPEX onto the C++ programming language [54]. The dataflow extension consists of two parts: a set of language primitives and constructs for describing the parameterized dataflow model and a set of language restrictions to limit the expressiveness of the host language.

The remainder of this chapter is organized as follows. Section 6.2 provides our analysis of the operation characteristics of DSP systems, our rationale for using the parameterized dataflow computation model, and our modifications to the dataflow model for supporting streaming communication patterns. Section 6.3 describes SPEX's modified parameterized dataflow computation model for supporting stream computations.

6.2 Modeling Wireless Protocols

In this section, we first describe our rationale for using the parameterized dataflow model for streaming computation. We then describe our rationale for modifying the

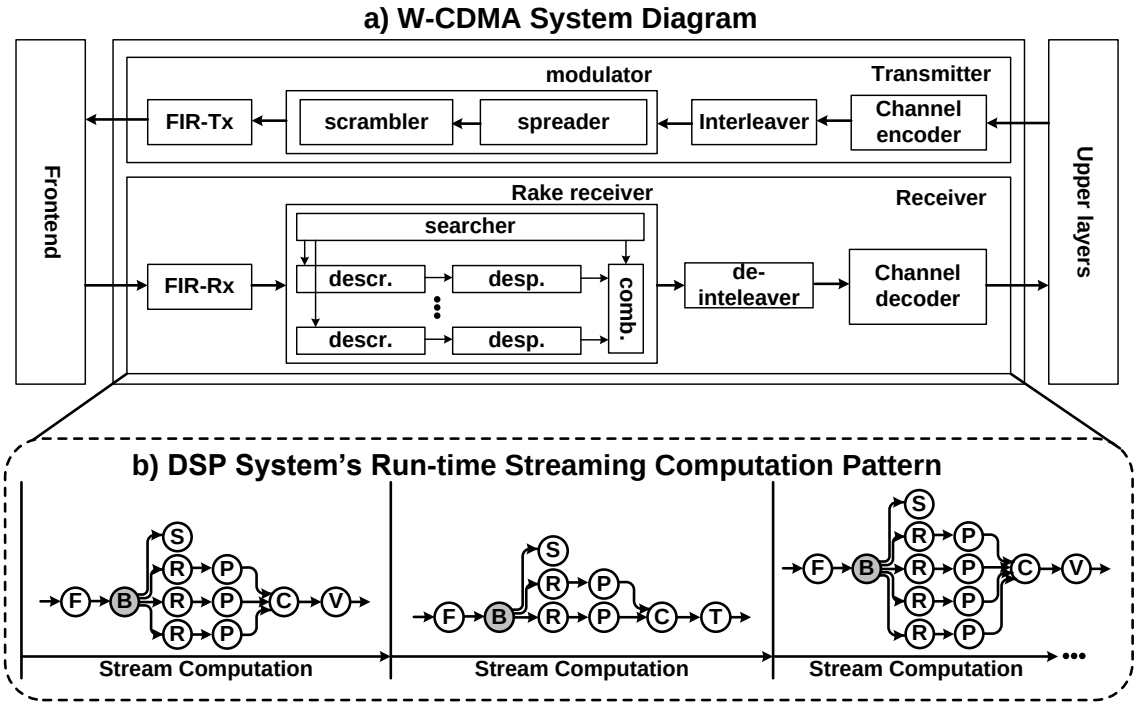


Figure 6.1: Part a: W-CDMA System Level Diagram. W-CDMA is used as the on-going example for SPEX in this study. Part b: DSP system run-time streaming computation pattern. The receiver may use different number of rake fingers (denoted by the R and P nodes) and different channel decoding algorithms (denoted by the T and V nodes). Shaded B nodes are memory buffers.

dataflow model for streaming communication. We illustrate the features of SPEX through the W-CDMA wireless protocol's physical layer processing [41]. Figure 6.1a shows the system-level diagram of W-CDMA. The receiver consists of a FIR filter, rake receiver, interleaver, and channel decoder. These algorithm kernels are connected in a feed-forward pipeline.

6.2.1 Streaming Computation in Wireless Protocols

In wireless protocols, DSP algorithm kernels are organized in pipeline-like computation chains, and data is streamed through the pipeline in a sequential order. In between long episodes of streaming computation, DSP systems intermittently reconfigure their

streaming patterns to account for changes from the users, the environment, and received inputs. Most DSP systems support multiple operation modes that are optimized for different services. These include changes in streaming rates, dataflow configurations, and algorithm kernels. For example, W-CDMA supports multiple data transmission rates ranging from 15Kbps to 2Mbps. The lower data rates are used for voice communications, and the 2Mbps is used for high-speed data communications. These different data communication rates also require different DSP algorithms and different stream configurations. Figure 6.1b describes this periodic reconfiguration in the streaming computation. During run-time streaming computation, the receiver may use different numbers of rake fingers (denoted in the figures by the R and P nodes) and different channel decoding algorithms (denoted in the figure by the T and V nodes).

6.2.2 Parameterized Dataflow Model (PDF)

The concurrent dataflow model has been used to describe streaming computations. A dataflow graph consists of a set of actors (or nodes) interconnected together with edges. Each edge contains both input and output stream rates for the source and destination actors. An actor's stream rates correspond to the amount of data consumed and produced per invocation. In particular, synchronous dataflow (SDF) has received considerable attention as the computation model for compilation onto multi-core architectures [34]. SDF is a type of dataflow model where the dataflow properties are defined statically. This allows the run-time execution schedule to be generated statically during compile-time [51]. Embedded systems usually have tight performance constraints and limited run-time scheduling support. Many of these embedded systems also use scratchpad memories instead of cache, where memory management is a software problem. Compiler-generated execution schedules are favorable because they require less run-time scheduling

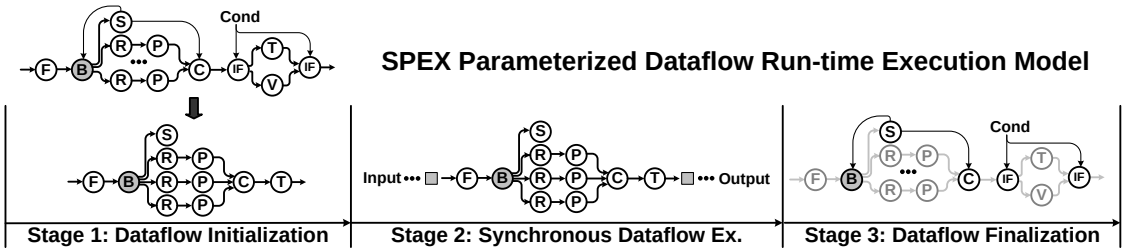


Figure 6.2: PDF execution model consists of three steps. Step 1, the parameterized dataflow graph is constrained into a synchronous dataflow graph. Step 2, the dataflow is executed following a static compile-time schedule. Step 3, PDF graph's data and states are updated with the most recent computed values.

and memory management overhead. However, because of its statically defined dataflow properties, SDF is too restrictive to describe the run-time reconfigurations of complex DSP systems. An ideal computation model for embedded systems should have the run-time efficiency of the SDF, while also providing enough flexibility to describe run-time reconfigurations.

SPEX is based on a more dynamic dataflow, namely the parameterized dataflow (PDF) computation model [15]. In PDF, dataflow attributes are described with parameters instead of constants. A parameter is a variable with a finite set of discrete values. Our choice of using the PDF is motivated by the fact that most DSP systems only have a finite set of discrete operating modes. These configurations in the dataflow can be adequately captured by a set of parameters with discrete values. We find that the following set of four dataflow properties should be parameterized to describe DSP systems' streaming computation.

- **Variable Dataflow Rates:** The input and output stream rates of dataflow actors may take on a range of values.
- **Conditional Dataflow:** Conditional dataflow is supported by using parameters to describe the branching conditions.
- **Number of Dataflow Actors:** Parameters can be used to fire a subset of the actors in a dataflow graph during run-time.

- **Streaming Size:** The number of data elements streamed per invocation should also be defined with parameters.

Run-time Execution Model and Compilation Support. Dataflow graphs are executed on hardware through run-time schedules. The schedule may be statically determined by a compiler or dynamically generated by a run-time scheduler. As mentioned before, SDF is good for embedded architectures because compiler-generated execution schedules require less run-time resources. With SPEX's PDF computation model, we propose a three stage run-time execution model: 1) dataflow initialization, 2) dataflow execution, and 3) dataflow finalization, as shown in Figure 6.2. These three stages are executed for every PDF graph invocation. During the initialization stage, the parameters are set to constant values, effectively constraining a PDF graph into a SDF graph. The dataflow execution stage follows a compiler-generated schedule for this SDF graph. The finalization stage updates the dataflow variables and states with the results of the dataflow computation. This three stage PDF execution model provides the best of both worlds; it still maintains the efficiency of SDF execution schedules, while also provides the flexibility to reconfigure the dataflow through initialization and finalization stages.

6.2.3 Modeling Streaming Communications

Even though the parameterized dataflow model is good at describing streaming computation, many DSP systems have complex streaming communication patterns that cannot be accurately described with the dataflow graph's one-dimensional FIFO communication edges. The following is a list of communication patterns that are needed.

- **Multi-dimensional Streaming Patterns:** Many DSP systems operate on vectors and matrices, which require memory buffers with multi-dimensional streaming patterns. For example, a vector FIFO buffer may have two different streaming attributes: the streaming pattern within each vector element and the streaming pattern among the buffer vector elements.

- **Non-sequential Streaming Patterns:** Many DSP algorithms do not follow strict FIFO streaming order. For example, a complex filter may access the real or the imaginary components of an array of complex numbers in strided sequential order. An interleaver may access an array in pre-computed random order.
- **Decoupled Streaming:** Many DSP systems consist of multiple decoupled dataflow computations. These decoupled dataflow computations may still be connected through buffers, but they may operate asynchronously from each other. For example, in a W-CDMA receiver, the front-end filter must operate under the periodic real-time deadline. The data gets down-converted into a lower data rate ranging from 15Kbps for voice communication up to 2Mbps for data communication. The output is then run through a backend error decoder that does not have strict real-time deadline requirements. Many decoder implementations do not operate in sync with the front-end filter.
- **Shared Memory Buffers:** DSP systems have memory buffers that are shared between multiple readers and writers. Dataflow edges are FIFO queues that support queue push and pop. Push and pop couple two separate operations for each data element: memory allocation/deallocation and memory read/write. Shared memory buffers requires decoupled operations for memory allocation/deallocation and memory read/write.

Previous works have attempted to address these different streaming patterns by proposing different dataflow computation models. For example, multi-dimensional dataflow was proposed for supporting streaming vectors and matrices [61]. Cyclo-static dataflow can be used to model strided streaming patterns [64]. However, these are point solutions that only address a specific streaming pattern. In SPEX, we propose a different design approach: relax the dataflow computation model to let the programmers construct the appropriate streaming patterns. Instead of attempting to describe a streaming pattern with one dataflow actor or edge, SPEX allows the programmer to use a set of dataflow actors and non-dataflow functions. This set of actors and functions are not explicitly connected, but are allowed to share the same data. The dataflow actors are used to describe dataflow streaming patterns, non-dataflow functions are used for infrequent variable updates. In SPEX, these special dataflow actors are called memory actors, and these non-dataflow functions are called memory functions. This is different from a traditional

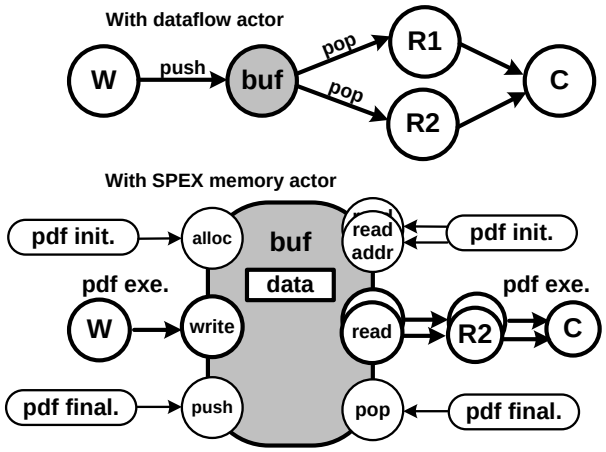


Figure 6.3: Example of a vector stream buffer with 1 writer and 2 readers. This buffer's communication pattern has all four streaming properties. This is a vector buffer, which requires multi-dimensional streaming patterns. Its has non-sequential streaming patterns because its readers must periodically reconfigure their streaming addresses. The writer and readers are decoupled because they have different real-time deadlines. This is also a shared memory buffer because the readers share the same data, but have different streaming patterns.

dataflow model where actors cannot share data. By sharing data, each memory actor or function can be used as a building block to model one aspect of the streaming pattern. The combination of a set of actors can be used to model complex stream patterns. The disadvantage for our approach is the data consistency problem. Traditional dataflow computation models do not have to deal with this problem because there is no shared data between the actors. In our PDF model, programmers must use locking mechanisms to access shared data. However, because our execution model enforces static SDF run-time execution schedules, the compiler has complete knowledge of the data access pattern for all actors and functions. Implementing locking mechanisms for a static schedule is more deterministic than for a multi-threading run-time environment.

The four streaming patterns listed previously can all be described using multiple memory actors and functions. A W-CDMA vector stream buffer is shown in Figure 6.3 with 2 readers and 1 writer. This buffer requires all four streaming patterns. 1) This is a vector

buffer, which requires multi-dimensional streaming pattern. Read and write operations access scalar data elements within a vector element. Push and pop operations manage the vector queue by accessing across vector elements. 2) This buffer has a non-sequential streaming pattern because its readers' streaming address must be periodically reconfigured. This is implemented with a memory function that sets up the reading address during the PDF initialization stage. 3) The writer and reader of this buffer are decoupled because they operate with different real-time deadlines. 4) This is also a shared buffer because there are two readers with different streaming patterns. The buffer is allowed to pop the data only after the data is read by both readers. Pop can be implemented as a memory function that runs during the PDF finalization stage. In comparison, a traditional dataflow actor can only define this complex streaming communication with vector push and pop.

6.3 SPEX Extensions for Streaming Computation

SPEX is a concurrent language extension designed for modeling streaming computation. It is based on the parameterized dataflow (PDF) model, described previously in Section 6.2. This language extension includes a set of additional variable and function types and program constructs, as well as a set of language semantic rules and restrictions. In this study, it is applied onto the C programming language, due to its popularity and large user base in the embedded DSP community. SPEX language syntax is mostly the same as the C language syntax. It contains a set of keywords for declaring stream-related variables, functions, and scopes. It also contains a set of restrictions that limits the set of C expressions that may be used under the various SPEX scopes and functions.

The rest of this section is organized as follows: Section 6.3.1 provides an overview of the

SPEX language extension, and our rationale for its language additions and restrictions. Section 6.3.2 describes the details of SPEX's variable types. Section 6.3.3 describes the details of SPEX's function types that are required for describing dataflow. Section 6.3.4 describes the languages constructs that are required for describing dataflow. In each of the section, both the SPEX keywords and restrictions are described. Section 6.3.5, section 6.3.6, and section 6.3.7 shows code example of implementing DSP algorithm kernels, memory buffers, and DSP systems in SPEX. In each section, we also discuss the implications of SPEX coding style on the overall performance of the software system.

6.3.1 Overview

The SPEX extension for the C programming language consists of two main components: a set of language keywords and a set of language restrictions. It is designed to support the description of a parameterized dataflow in C. The challenge in designing a C language extension comes from the fundamental differences between the underlying computation model between C and PDF. C is an imperative language with a sequential computation model, whereas PDF is a concurrent computation model. Because dataflow computation models are different from sequential computation model, there are operations that are allowed in a sequential computation model that are not allowed in dataflow computation models.

C is still a very popular programming language for embedded applications. Compare with modern programming languages, it exposes more of the underlying hardware complexity to the programmers. Therefore, it requires higher software engineering effort, but provides more efficient and deterministic code behaviors. Because performance efficiency is still the first order constraint on embedded devices, C's approach is still a valid one in the embedded domain. SPEXC aims to provide the efficiency of C by preserving C's syntax as

much as possible. This leads SPEX to support two computation models: sequential and dataflow computation models. Dataflow actors are written in the sequential computation models, and dataflow graphs are constructed with the concurrent computation model.

SPEX's keywords provide programmers a guideline for writing C programs with valid underlying computation models. These keywords, along with the language restrictions, also provide directives for the SPEX compiler to enforce the correct C syntax for dataflow computation, and enable better C-to-dataflow translations. SPEX keywords include new variable and function types to distinguish the descriptions of different dataflow graph components. For example, parameter variables are integers with a discrete set of values. A special `param` keyword is provided to distinguish parameter variables from normal variables. Both dataflow actors and graphs are written as C functions. Special keywords `stream_kernel` and `stream_system` are provided to distinguish these two types of C functions. Due to the difference in the underlying computation model between these two types of functions, SPEX restrictions are created to restrict the allowable C syntax within each type of function.

6.3.2 SPEX Streaming Types

Parameters. Parameters, denoted by the keyword `param`, are variables that may only take on a finite set of discrete values specified by the programmer. Parameters are used to describe various parameterized dataflow properties of a PDF graph. One may declare a parameter variable with the following syntax: `param base_type var_name`. Currently, only integer-based parameter subtypes are supported in SPEX. The behavior of a parameter variable is similar to that of a C variable declared with the `static` modifier: the value of a parameterized variable are kept across function calls. This is because parameter variables are used to control dataflow configurations and represent the dataflow system states.

Therefore, the behavior of a parameter variable is similar to a static variable rather than a typical local variable. Arithmetic and comparison operations are supported for parameter variables. The range of a parameter variable can be defined using the following two functions: `void param_range(T min, T max)` and `void param_values(int num_vals, T val1, ...)`. The two functions are used to declare either a range of values or a set of discrete values for each parameter. Programmers are not required to use these two functions to define the values of a parameter. Compiler analysis can be used to deduce the possible values for these variables. If the functions are not used and the compiler analysis cannot deduce a set of values for a parameter variable, then the compiler is going to issue a compilation warning, and the subsequent compiler optimizations assume that this parameter variable can take on the full range of an integer variable.

Channels. SPEX channels, denoted by the keyword `channel`, are used to model dataflow edges. The `channel` keyword is used as a modifier for declaring an array in C. This modifier can be used with both integer and floating-point arrays. In a dataflow graph, each dataflow edge represents a FIFO buffer with associated input and output rates. Arrays declared with the `channel` keyword indicate to the compiler that the size of these arrays are dataflow rates, not absolute sizes. The compiler is allowed to optimize the dataflow execution by increase these arrays' sizes to a multiple of their original sizes. The upper bounds of these arrays' sizes are determined by the context of which these channel variables are used. If upper bounds cannot be deduced from their usage context, then the default is to assume that the declared array sizes are the only size that is allowed.

A set of language restrictions are enforced on the usage of these channel arrays to properly model the behaviors of these arrays as dataflow edges. Channel arrays must be declared as local variables. There are two types of special functions in SPEX: functions for modeling PDF actors and PDF graphs. Only PDF graph functions are allowed to

declare channel arrays as local variables. Further details with regard to these special PDF functions are explained in the following section. Channel arrays must also be used only as function arguments for functions calls. Passing a channel array either as a function argument requires the callee function to use this array either as a read-only input array or write-only output array, not both. Furthermore, within a function body, each channel array can have multiple reader callee functions, but only one writer callee function.

Shared Variables. Dataflow computation models typically do not allow shared variables between actors. Data are only exchanged through message passing communication channels between actors. In C language syntax, this implies that global variables should not be allowed. However, as mentioned in the previous section, SPEX adopts a modified dataflow model, PDF, that allows shared variables between actors. In PDF, only memory dataflow actors are allowed to access these shared variables. However, in C, all globally declared variables are shared variables. Therefore, SPEX propose a set of rules that restricts the declaration and access to these shared variables as follows. Shared variables are variables declared with `shared` keyword. Since global variables are implicitly shared variables in the C semantics, they do not have to explicitly use the `shared` keyword. Shared variables may be integer scalar or array variables. Only PDF actor functions may access and modify shared variables, and only PDF graph functions may declare shared local variables. Local variables are only visible to the functions calls within their declared PDF graph function. Global shared variables may be accessed by all PDF actor functions, and local shared variables can only be used as function call arguments.

6.3.3 SPEX Streaming Functions

SPEX PDF functions are used to construct parameterized dataflow graphs. Both PDF actors and graphs are described with the C function syntax. The `stream_kernel` key-

word must be used for functions describing PDF actors, and the `stream_system` keyword must be used for functions describing PDF graphs. Code examples of these two types of functions are given in the following sections. In addition to PDF functions, normal C functions are also supported in SPEX.

The Two Computation Models in SPEX. Unlike many other programming languages, SPEX contains two very different computation models. PDF graph functions are based on the concurrent dataflow computation model, whereas PDF actor functions are based on the sequential execution model. Furthermore, because PDF actor functions are meant to model dataflow actors, they also must take on a restricted subset of the C syntax. This the main reason why these two special function types are created in SPEX. They serve as guidelines for the programmers to write sequential C-based code that can be converted into concurrent dataflow graphs. They also serve as directives for the compile to create concurrent intermediate representations from SPEX.

Function Call Restrictions. Recursive function calls are not supported in SPEX because they cannot be properly translated into dataflow graphs. PDF graph functions may call other PDF graph functions, PDF actor functions, and C functions. Allowing PDF graph functions to call other PDF graph functions enables hierarchical dataflow descriptions, where each dataflow node may contain an entire dataflow graph. PDF actor functions may call C functions, but not PDF graph functions or PDF actor functions. This restriction is enforce because of the different computation models that exists within SPEX-C. C functions may only call other C functions.

Static Language Semantics. In SPEX, all variables and functions must be statically declared. This means that C's dynamic language features, such as run-time memory allocations and function pointers, are not supported. Because the dynamism in DSP systems is described through parameters, dynamic language features do not add any

benefits in describing these systems. Static variable and function declarations also produce more efficient code because they require less run-time management.

Function Arguments. In dataflow computation model, each dataflow edge must either be an input or an output edge. These edges are described in SPEX as C function arguments. Therefore, SPEX requires that each PDF function argument to be either read-only or write-only. The only exception is the shared variables, which can be both read and written by the same PDF actor or graph. There is also no explicit return values for these PDF functions. PDF computation model allowed dataflow edges, shared memories, and parameters as input and output dataflow edges for the actors and graphs. Therefore, PDF C function arguments must be one of the follow four variable types: channel array variables, shared array variables, shared scalar variables, and parameter scalar variables. Both read-only and write-only array variables follow the pass-by-pointer syntax in C: `func(arg_type * arg_val)`. Read-only channel variables may use type modifier `inchannel`, and write-only channel variables may use type modifier `outchannel`. These modifier keywords are not mandatory. They serve as reminders to the programmers of these variables' access restrictions. It is the compiler's responsibility to ensure that channel variables are never read and written within the same function. Shared variables must use type modifier `shared`. For scalar variables, read-only arguments follow the pass-by-value syntax in C: `func(arg_type arg_val)`, and write-only variables follow the pass-by-pointer syntax: `func(arg_type * arg_val)`.

Accessing Shared Variables. In SPEX, shared variables may be declared globally or locally within PDF graph functions. Global shared variables can be used directly by any PDF actor function, whereas local shared variables must be passed explicitly by their caller PDF graph function. Because SPEX is based on a concurrent dataflow model, multiple PDF actor functions and PDF graph functions may be scheduled to execute in

SPEX concurrent language constructs	Explanations

`stream {`
`  ... // code for PDF initialization`
`  for (...) {…} // PDF dataflow`
`  ... // code for PDF finalization`
`}`
	A PDF graph construct. It must contain only one for-loop construct for describing the dataflow. While and do-while loops are not allowed in pdf construct.

`ll_for (init;cond;incr)`
`{`
`  ... // loop body`
`}`
	Parallel for-loop construct. The different iterations of the loop body are executed in parallel. There can not be data dependencies across loop iterations.

Figure 6.4: SPEX language constructs for describing dataflow operations.

parallel. To enforce data consistency, SPEX requires that any PDF actor function that accesses a certain shared variable cannot be scheduled to execute in parallel with any other PDF actor function that also accesses the same shared variable. This is similar to transactional memory's transactional region – a PDF actor's function body is atomic with respect to other PDF functions that access the same shared variable. PDF actor functions that operates on different shared variables are allowed to execute in parallel. This responsibility is left to the compiler to determine the order of execution, and preserve the data consistency of all of the shared variables.

6.3.4 SPEX Streaming Constructs

A special language construct, stream scope, is provided in SPEX for describing parameterized dataflow (PDF) computations. Since SPEX supports both sequential and concurrent dataflow computation models, it must provide language syntax for distinguishing code written under these two different computation models. SPEX assumes that all code follows C's sequential execution model unless it is written within a stream scope. Code written within a stream scope assumes the PDF execution model, as outlined in Section 6.2.2. Figure 6.4 lists the syntax for this stream construct, a stream scope is declared with the keyword `stream`. Each stream scope contains three sections: dataflow initialization, static dataflow computation, and dataflow finalization. Static dataflow computation

requires that the control flow of the computation does not change during run-time. In order to support run-time reconfigurations of DSP system, the PDF execution model was proposed. In PDF execution model, the initialization and finalization stages are responsible for setting up the control flow before and after dataflow computation. The basic concept of the PDF execution model is similar to the constructor and destructor in a C++ object class, where these two functions are used to initialize and finalize the run-time behavior of objects.

Stream Scope Syntax. Stream scope can be viewed as a special for-loop construct with additional sections for loop initialization and finalization. It must contain only one for-loop construct within its scope, and may also contain optional code before or after the for-loop. The for-loop corresponds to the static dataflow computation, and the optional code before and after the for-loop correspond to the dataflow initialization and finalization. While and do-while loops are not allowed within the stream scope. Dataflow initialization is performed to setup the dataflow before computation, and finalization is performed after computation to update the dataflow computation results. In a stream scope, these two sections must be sequential code that does not contain any loop structures. The initialization section is the sequence of code before the static dataflow's for-loop structure, and the finalization section is the sequence of code after the static dataflow's for-loop structure.

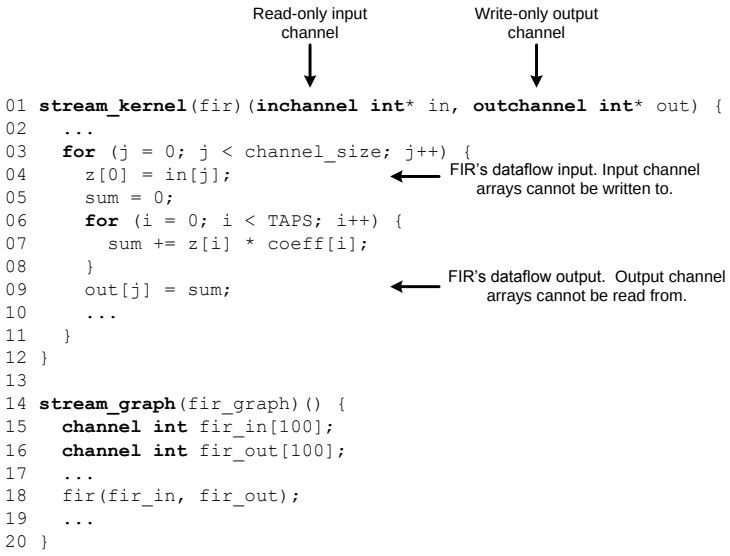
Stream Scope's For-loop Construct. A set of syntax restrictions are enforced within stream scope's for-loop construct. If and switch statements are allowed within the for-loop constructs. Break and continue are not allowed, unless they are used within the context of a switch statement. Nested stream descriptions are allowed. Loop constructs are not allowed, unless they are defined as a parallel loop construct or as part of a nested stream scope. The description of the parallel loop construct is listed in the following

paragraph. C function calls are not supported, unless they are PDF functions calls. In last sections, we explained that PDF functions are used to describe dataflow actors and graphs. In SPEX, PDF function calls take on a different language semantic than in traditional imperative languages: each PDF function call creates an explicit PDF actor or graph. For example, multiple function calls to the same `stream_kernel` function create multiple copies of the same PDF actor. All of the actors are created during the initialization stage, and they are destroyed during the finalization stage. This is very different from the C language semantics, where multiple function calls indicate that the function is executed sequentially multiple times. This is one of the key differences between the sequential computation model versus the concurrent dataflow computation model.

Parallel Loop Language Construct. PDF function calls create explicit actors or graphs. Parallel loop construct is used as a method to create multiple actors or graphs from the same PDF function. The syntax of this construct is summarized in Figure 6.4. The construct follows the for-loop syntax, but uses keyword `ll_for` keyword instead of the `for` keyword. This loop construct can only be used within a stream scope's for-loop construct. The syntax is also similar to the for-loop syntax with a few restrictions. The loop-body must only contain PDF function calls. There must be no inter-loop data dependencies. Furthermore, this loop's upper bound, lower bound and increment must all be either constants or variables that are defined in its stream scope's initialization section. Programmers must understand that this is not a typical software loop, but a shorthand for creating multiple dataflow actors and graphs.

6.3.5 Implementing DSP Algorithm Kernels

In this section, we demonstrate code example of implementing a DSP algorithm kernel in SPEX with the `stream_kernel` function. The code in Figure 6.5 shows a SPEX imple-




```
01 stream_kernel(fir)(inchannel int* in, outchannel int* out) {
02   ...
03   for (j = 0; j < channel_size; j++) {
04     z[0] = in[j];
05     sum = 0;
06     for (i = 0; i < TAPS; i++) {
07       sum += z[i] * coeff[i];
08     }
09     out[j] = sum;
10     ...
11   }
12 }
13
14 stream_graph(fir_graph)() {
15   channel int fir_in[100];
16   channel int fir_out[100];
17   ...
18   fir(fir_in, fir_out);
19   ...
20 }
```

Figure 6.5: DSP algorithm kernel example – FIR filter. The keyword `stream_kernel` is on line 1 to indicate that this is a PDF actor function.

mentation of FIR filter. In this example, there is one input channel array variable and one output array variable, denoted with keyword `inchannel` and `outchannel`. As shown on line 4 and 9, input channel variable can only be read from, and output channel variable can only be written to. If this restriction is violated, then the SPEX compiler should return an error message. This restriction of the read-only and write-only nature of function arguments is the biggest syntactical difference between a `stream_kernel` function and a normal C function. The function body of this function follows the same language syntax as C.

In this example, there are two function arguments – `in` and `out`. The first array is the input and the second array is the output of this stream kernel function. The input and output array each contains 100 elements. However, during the execution of this function, the SPEX kernel compiler does not make the guarantee that each time fir function is called, only 100 elements will be access from the input and output pointer locations. Because both of these variables are channel variables, the size of the array

indicates dataflow rates, not dataflow sizes. Therefore, a multiple of 100 elements maybe used as part of compiler optimizations as the actual sizes for both the input and output variables. However, SPEX requires that only a multiple of the declared array size can be used, and the ratio of the actual array sizes across all channel variables must be the same as the declared array sizes. For example, this means that an array size of 250 is not allowed in this case for either input or output array. Setting the input and output array sizes to 200 and 300 is also not allowed, as the ratio of the input and output array is 1-to-1.

Shown on line 2 in Figure 6.5, static variables are allowed as local variables. In the dataflow computation model, this means that SPEX allows both stateful and stateless dataflow actors. Although this does not make a difference in the execution, having static local variables requires the compiler to perform more memory book keeping for DSP processors with only scratch-pad memory. Each static variable must be allocated its own memory location in the local memory of the DSP processor that execute the dataflow actor. This requires extra work for the compiler if DSP processor's local memory does not have the space to fit the entire array. In addition, having stateful dataflow actors may also make other compiler optimizations more difficult. One such optimization is the dataflow actor fission, where a dataflow actor is duplicated into multiple copies in order to expose more kernel-level parallelism. Data consistency must be kept between duplicated actors that share the same data. In general, it is more efficient to avoid static local variable if possible.

6.3.6 Implementing Memory Buffers

In addition to DSP algorithm kernels, memory buffers also are essential elements in constructing wireless protocol systems. Implementing a memory buffer requires two parts

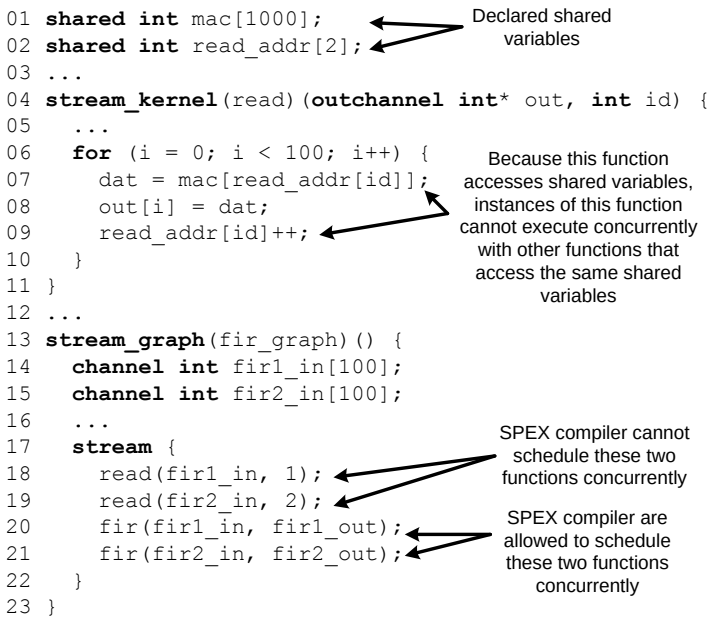


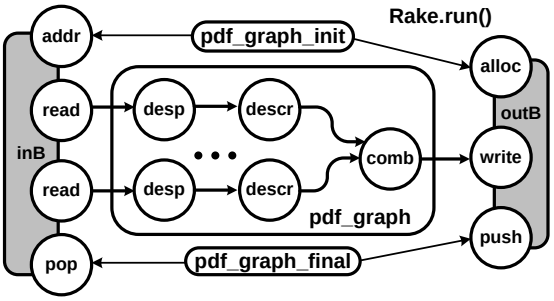
Figure 6.6: A vector stream buffer with 2 readers and 1 writer. Data objects are declared with the keyword `spex_memory` (on line 2). This example implements the same buffer shown in Figure 6.3

– declaring the memory storage space for a buffer, and defining the access functions that manipulate this memory storage space. Like C, memory storage spaces are declared as array variables in SPEX. Access functions can be implemented as PDF actor functions using the `stream_kernel` keyword. In this section, we elaborate on the implementation issues of memory buffers, and present an example memory buffer implementation. The code example is shown in Figure 6.6.

In SPEX, memory storage space for buffers can be implemented as array variables. They can be declared either as global variables or local variables declared with the `shared` keyword. In the example shown in Figure 6.6, the memory storage space is statically declared as global array variable `mac` on line 1. This particular buffer has two readers, `reader` array variable is declared to hold the reading index for the two readers. Both array variables are declared with static array sizes, this is because SPEX does not allow dynamic memory allocations. This means that `malloc` function is not supported. Arrays must be

declared with known array sizes. Pointers can only be used to pass memory addresses for these array variables. The main reason for this restriction is due to the underlying processor hardware. SDR processors such as the SODA and Ardbeg processors consist of one ARM control processor and multiple DSP processing elements (PEs). Because the PEs do not support operating systems, supporting dynamic memory allocation requires each processor to keep a set of book-keeping information on the memory usage of the processor. Because these processors have software-managed scratchpad memories, explicit DMA operations are required to transfer data between memories. Handling memory management during run-time requires each array access to check its memory address, and dynamically issue DMA operations if neccessary. This is more complex and less efficient than compile-time memory allocations. Furthermore, compile-time memory management is also able to use more sophiscated algorithms than the run-time systems. Because wireless protocols have very predictable run-time behavior, the extra flexibility of the run-time memory management also does not add any programming benefits.

For illustration purposes, only the buffer reader is shown in the code example shown in Figure 6.6. As mentioned in the previous section, the operations within a PDF function is atomic. This means that programmers do not have to use specific locks or mutex to ensure data consistency. As shown on line 7 and 9, shared variables can be access the same way as local variables. The compiler must provide this guarantee by never schedule two PDF functions that accesses the same global variable at the same time. This also means that it is generally not a good idea to write a long PDF function that access multiple global variables, as this will force the compile to generate more sequentialized schedule.



a) Rake receiver PDF diagram

```
01 shared int inb[2000];
02 shared int outb[2000];
03 shared int readers[MAX_FINGERS];
04 ...
05 stream_kernel(rake_init)(param int r, param int fingers,
06                           param int * stream_size) {
07   ...
08   if (r == 4)      stream_size = 1000;
09   else if (r == 8) stream_size = 2000;
10   for (i = 0; i < fingers; i++)
11     addr(i, peak[i]);
12   alloc(250);
13 }
14
15 stream_kernel(rake_final)(param int r) {
16   ...
17   if (r == 4)      pop(1000);
18   else if (r == 8) pop(2000);
19   push(250);
20 }
21 ...
22 stream_system(rake)(param int r, param int fingers) {
23   channel int chan1[MAX_FINGERS][100];
24   channel int chan2[MAX_FINGERS][100];
25   channel int chan3[MAX_FINGERS][100];
26   channel int chan4;
27   param int stream_size;
28   ...
29   stream {
30     rake_init(r, fingers, stream_size);
31     for (j = 0; j < stream_size; j++) {
32       ll_for(i = 0; i < fingers; i++) {
33         read(chan1[i]);
34         despreader(chan1[i], chan2[i]);
35         descrambler(chan2[i], chan3[i]);
36       }
37       combiner(chan3, chan4);
38       write(chan4);
39     }
40     rake_final(r);
41   }
42 }
```

b) Rake receiver SPEX implementation

Figure 6.7: Rake receiver implemented with PDF graph functions: `pdf_graph_init`, `pdf_graph`, and `pdf_graph_final`. These three PDF functions are used to describe the three stages in a PDF's run-time execution. `pdf_graph_init` is used to describe the PDF graph initialization; `pdf_graph` is used to describe the PDF graph execution; and `pdf_graph_final` is used to describe the PDF graph finalization.

6.3.7 Implementing DSP Systems

In this section, we illustrate examples of using implementing DSP systems with SPEX. Figure 6.7 shows the W-CDMA rake receiver implementation using a `stream_system` function. As mentioned in Section 6.2, we propose a three stage PDF run-time execution model. These three stages must be described as three sections of a `stream` scope. These three sections are described starting from line 30, 31, and 40. Both the intialization and finalization section are described as `stream_kernel` functions. Rake receiver is composed of three DSP algorithms: despreader, descrambler, and combiner. Each despreader and descrambler pair is called a rake finger. Most DSP systems have run-time reconfigurations. There are three run-time reconfigurations modeled in this simplified version of the rake receiver: 1) the number of rake fingers; 2) the number of elements streamed per function invocation; 3) the streaming read address for each rake finger. Because the number of rake fingers and the number of streaming elements both affect the dataflow configuration, they are described with parameter variables `fingers` and `stream_size`. The streaming read address is determined by initializing the input stream buffer during the PDF initialization stage, shown on line 11.

Initialization & Finalization. In SPEX, DSP systems are modelled as PDF graphs. The purpose of the initialization stage is to setup the PDF graph for execution. Because the dataflow computation must use a synchronous dataflow schedule, the initialization stage is responsible for setting all of the parameter variables to constant values. Setting up the stream communication patterns is also done in this step. In the rake receive example shown in Figure 6.7, the initialization function for the rake receiver is shown between line 5 and line 13. Line 8 and 9 set the parameter variable `stream_size` to a constant value based on the input spreading factor. The input stream buffer is initialized for each finger by setting the starting memory read address for each finger on line 10 and

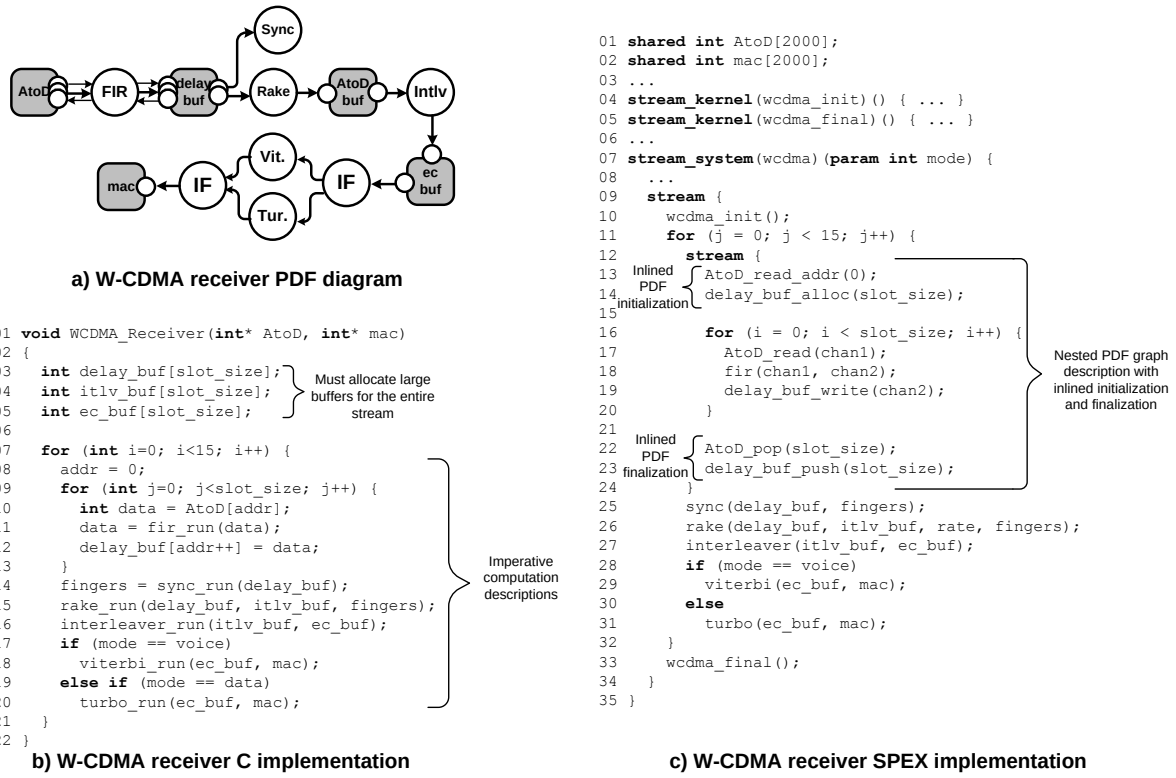
11. To setup the streaming write buffer, we allocate memory space for the output on line 12.

The purpose of the PDF finalization stage is to update the PDF graph's internal states with the computed dataflow results. It is also used to perform memory management operations for the input and output buffers, as shown in Figure 6.7 on lines 17, 18, and 19. After the data is read out of the input buffer by all of the rake fingers, its memory is deallocated. The output buffer performs a push operation to make the written output visible to other PDF actors that are reading or writing to this buffer.

Stream Construct. Figure 6.4 lists the set of parallel language constructs that are supported in SPEX for describing a concurrent dataflow graph. `stream` construct is used to describe a PDF graph. SPEX requires each dataflow to be described as a for-loop construct. Therefore, each PDF construct must contain one for-loop. Because rake receiver uses multiple despreaders and descramblers. These algorithms are declared using the `ll_for` construct, as shown from lines 32 to 36.

Parameter Variable Restrictions. Parameter variables must be declared as local variables within the PDF function. The value of parameter variables must be defined before the for-loop construct in `stream`. And they are not allowed to be redefined in the `stream` for-loop body. In addition, they are the only type of variables that are allowed to use as part of if-statements' conditional expression.

W-CDMA Receiver Implementation. Figure 6.8 shows a simplified W-CDMA receiver implementation, in both C and SPEX. SPEX implementation requires larger code size than the C implementation. The stream computation itself requires 27 lines of SPEX code and 15 lines of C code. However, all of the streaming characteristics are lost in the C implementation. Because the algorithms are executed in sequential order, large buffers are allocated to pass entire streams of intermediate results. It is possible to reduce the




```
01 void WCDMA_Receiver(int* AtoD, int* mac)
02 {
03   int delay_buf[slot_size];
04   int itlv_buf[slot_size];
05   int ec_buf[slot_size];
06
07   for (int i=0; i<15; i++) {
08     addr = 0;
09     for (int j=0; j<slot_size; j++) {
10       int data = AtoD[addr];
11       data = fir_run(data);
12       delay_buf[addr++] = data;
13     }
14     fingers = sync_run(delay_buf);
15     rake_run(delay_buf, itlv_buf, fingers);
16     interleaver_run(itlv_buf, ec_buf);
17     if (mode == voice)
18       viterbi_run(ec_buf, mac);
19     else if (mode == data)
20       turbo_run(ec_buf, mac);
21   }
22 }
```

```
01 shared int AtoD[2000];
02 shared int mac[2000];
03 ...
04 stream_kernel(wcdma_init)() { ... }
05 stream_kernel(wcdma_final)() { ... }
06 ...
07 stream_system(wcdma)(param int mode) {
08   ...
09   stream {
10     wcdma_init();
11     for (j = 0; j < 15; j++) {
12       stream {
13         AtoD_read_addr(0);
14         delay_buf_alloc(slot_size);
15
16         for (i = 0; i < slot_size; i++) {
17           AtoD_read(chan1);
18           fir(chan1, chan2);
19           delay_buf_write(chan2);
20         }
21
22         AtoD_pop(slot_size);
23         delay_buf_push(slot_size);
24       }
25       sync(delay_buf, fingers);
26       rake(delay_buf, itlv_buf, rate, fingers);
27       interleaver(itlv_buf, ec_buf);
28       if (mode == voice)
29         viterbi(ec_buf, mac);
30       else
31         turbo(ec_buf, mac);
32     }
33     wcdma_final();
34   }
35 }
```

Figure 6.8: W-CDMA receiver implementation. The example shows both C and SPEX implementations of the receiver.

buffer size by manually rewriting the C code. However, because optimal buffer sizes are dependent on the size of hardware's physical memory, programmers are forced to write machine-dependent code. In SPEX, because the streaming patterns are exposed in the language, the compiler can automatically pick the optimal buffer size. Programmers do not have to be aware of the underlying hardware.

The W-CDMA standard divides the receiving data into TTI (Transmission Time Interval) blocks. Each TTI block contains a maximum of 5 W-CDMA frames, and each frame is further divided into 15 W-CDMA slots. Dataflow reconfigurations occur at multiple data block granularity. Because each reconfiguration requires its own PDF initialization and finalization stages, the W-CDMA receiver is implemented with nested stream scopes, as shown in Figure 6.8c from line 12 to 24.

6.4 Related Work

Dataflow Computation Models. There has been considerable work in reconfigurable dataflow models. These include less restrictive dataflow models, hybrid SDF with finite-state-machines(FSMs) [78], and parameterized SDF (PSDF) [15]. Examples of less restrictive dataflow models include the cyclo-static dataflow model (CSDF) [64], Boolean dataflow model (BDF) [20], and Synchronous piggybacked dataflow (SPDF) [63]. CSDF supports cyclic dataflow rates, where the rates are described as a periodic set of numbers. BDF includes conditional split and merge actors on top of the SDF. SPDF supports reconfigurations by coupling infrequent control updates with the synchronous dataflow. In the hybrid SDF+FSM models, the different dataflow configurations are expressed as the different states of the FSM. SPEX's PDF model is very similar to the PSDF model. One noticeable difference is that our model supports memory actors that share data. Hierarchical dataflow models have also been proposed before to model multi-rate DSP applications with constraints [21].

Dataflow Languages. There have been many dataflow languages proposed for modeling DSP systems. Some of these are frameworks that are designed for a wide range of application by supporting multiple dataflow models, such as the Ptolemy project [49], the DIF format [42], and the PeaCE design flow [36]. There also have been languages that are designed explicitly for a processor architecture. StreamIt [79] was proposed for mapping streaming computations onto tiled processor architectures. The original StreamIt was designed based on the SDF computation model. Recent updates have also introduced parameterized variables, allowing the description of variable rate dataflow. StreamIt supports stream reconfigurations and updates through teleporting messages [80], which has similar functionality to the SPDF model. Instead of SPEX's explicit memory class for streaming buffers, StreamIt couple the memory with the communication through peak op-

erations. Peaking allows the programmers to look ahead into the channel without popping the data. However, peaking cannot describe a buffer with multiple readers or writers, as each reader requires its own channel. Because peaking implicitly allocates physical memory, shared vector or matrix buffers may require significant duplication overhead.

Other Streaming Languages. There are also other streaming languages that are not based on dataflow computation models, such as Brook [19] and Sequoia [30]. Both are imperative languages with explicit constructs for streaming array structures. Sequoia is also designed to expose an application's memory hierarchy to the programmers.

6.5 Summary

In this chapter, we describe SPEX, a set of language extensions for describing wireless protocols. SPEX's streaming semantics are based on a parameterized dataflow computation model. We have modified this dataflow model to introduce special dataflow actors that are allowed to share data. This allows complex streaming communication patterns to be described with a set of dataflow actors. SPEX is applied onto the C++ programming language. It consists of a set of language constructs for describing the semantics of parameterized dataflow computations, and a set of language restrictions for helping the embedded compilation process.

CHAPTER 7

Compilation Support for the Ardbeg processor

7.1 Introduction

Some of the greatest advantages of Software Defined Radio (SDR) are based on the "software" aspect of the implementations. SDR promises greater flexibility, multi-mode operation, lower engineering efforts and costs, and shorter time-to-market. These are all based on the assumption that software development is easier than hardware development. Therefore, it is a first-order design consideration to provide the tool-support for mapping software implementations of wireless protocols onto SDR processor hardware. In Chapter 6, a programming language extension, SPEX, is proposed for describing wireless protocols. This chapter describes a compilation system, the SPIR compiler, that automatically maps SPEX-based C code to multi-core system-on-chip (MPSoC) architectures, because many of the proposed SDR processor solutions, including the SODA and Ardbeg, fall under this architecture category. The current SPIR compiler only targets the Ardbeg processor, but the proposed compilation techniques are applicable to a wide range of MPSoC processors. We choose to target the Ardbeg processor instead of the SODA processor because of its commercial tool support.

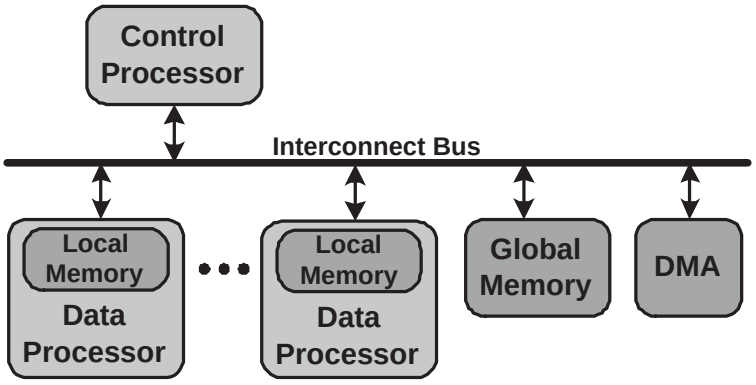


Figure 7.1: SDR control-data decoupled MPSoC architecture consisting of one general-purpose control processor, multiple data processors, and a hierarchical scratchpad memory system that are all interconnected with a bus.

MPSoC Architecture. MPSoC architectures, shown in Figure 7.1, typical have two groups of processors – the control and data processors. Control processors are general-purpose processors (e.g., ARM) that are capable of handling control-intensive code and are best suited for protocol scheduling and memory management. Conversely, data processors are specialized DSP processors that can perform heavy-duty data-intensive computations. Single-instruction multiple-data (SIMD) or vector processing is typically employed in the data processors. The system has a non-uniform memory architecture, with both a global memory and local memories on each data processor. Many systems use scratchpad memories instead of caches for local memories, which makes memory management the responsibility of the software. In many systems, one control processor is capable of supporting multiple data processors.

Two-Tier Compilation Approach. MPSoC compilers are a difference challenge due to the heterogeneous nature of the hardware. For example, VLIW processors require instruction scheduling support, and SIMD-based processors benefit from automatic vectorization support. For processors with scratchpad memories, efficient memory allocation must also be provided. Compiling for the SODA and Ardbeg architectures must deal

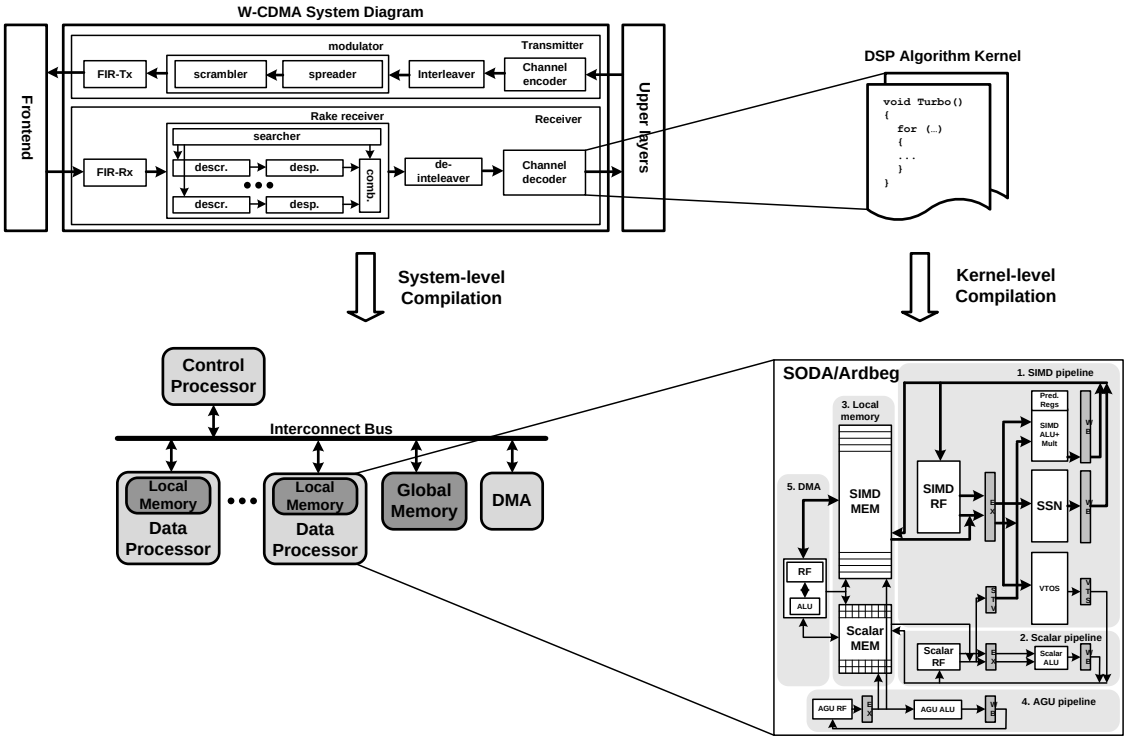


Figure 7.2: Two-tier compilation approach for SODA and Ardbeg processors. On the system-level, the compiler deal with coarse-grained compilation challenges, such as function-to-processor assignments and DMA operations. On the kernel-level, the compiler deal with fine-grained compilation challenges, such as VLIW scheduling and vectorization for SIMD processors. The SPIR compiler is a system-level compiler that only address the coarse-grained compilation challenges.

with these challenges, as well as additional challenges associated with multi-core systems. These include automatic multi-thread parallelization, synchronization, and DMA operations for communicating processors. Addressing all of these challenges at once can quickly become unmanageable. Therefore, this thesis proposes a two-tier compilation approach, as shown in Figure 7.2. The compilation process is broken into two parts: system-level and kernel-level compilation. On the system-level, the compiler deals with coarse-grain compilation challenges, such as function-to-processor assignments and DMA operations. On the kernel-level, the compiler deals with fine-grained compilation challenges, such as VLIW scheduling and vectorization for SIMD processors. The SPIR compiler is a system-

level compiler that only address the coarse-grained compilation challenges. The input of the compiler is sequential C-SPEX code. The output of the compiler is multi-threaded C code that includes processor assignments, inter-processor synchronization, and DMA communication operations. The multi-threaded C code is then feed into kernel-level compiler and linker to generate the final machine code.

Function-level Compilation. The SPIR compiler is a function-level compiler, which means that the granularity of an atomic execution unit is a function, not an instruction. A traditional compiler's intermediate representation (IR) models instruction-level interactions. A different IR is needed to model inter-function behavior. In Chapter 6, we proposed describing wireless protocols' system-level computation with a modified parameterized dataflow (PDF) model. This computation model, named the SPIR computation model, is use as the IR for this system compiler. The compiler is divided into the frontend and backend compilation. The frontend translates from C-SPEX extension into SPIR. And the backend then translates from SPIR into multi-threaded C code.

As part of this study, we also examined function-level compiler optimization. We find that some instruction-level optimizations can be adopted for function-level. In this study, we studied software pipelining. Function-level software pipelining is an adaptation of an existing instruction-level compilation technique.

Chapter Contribution and Organization. The contribution of this chapter is presenting a compilation infrastructure for automatically mapping wireless protocols onto the Ardbeg processor architecture. In Section 7.2, we give an overview of the compiler infrastructure. In Section 7.3, we describe the compiler frontend for translating C-SPEX code into SPIR format. In Section 7.4, we describe one function-level compiler optimizations for code written in SPIR format. And in Section 7.5, we describe the code generation process of translating SPIR code into multi-threading C code.

7.2 The SPIR Compiler

This section discusses the motivation and the overall infrastructure of the SPIR compiler system. It first provides our rationale for building a function-level compiler. It then goes over the overall SPIR compiler infrastructure. It then goes over the three language formats that are used in this compiler: SPEX, SPIR, and SocC. SPEX is the input language, SPIR is the compiler's intermediate representation, and SocC is the output language. Both SPEX and SocC [70] are high-level programming language extensions of C.

7.2.1 Rationales for Function-level Compilation

In modern computer systems, the task of executing a software program is divided between the compiler and the operating system. Typically, the compiler is responsible for instruction-level scheduling and management, and the operating system is responsible for thread-level scheduling and management. This division of labor makes sense for general purpose computer systems, as it combines the efficiency of the compile-time algorithms with the flexibility of the run-time systems. However, in a MPSoC system, we have a range of heterogeneous processors, each designed with a different purpose. For example, in the Ardbeg system, the ARM control processor is used for overall protocol management and interface, whereas the Ardbeg data processors are used for heavy duty DSP processing. Running an operating system on the ARM control processor makes sense, as it is designed to support many of the general purpose computing components, such as caching and virtual memory support. However, the Ardbeg PEs are designed to support DSP computations with the highest power efficiency. Without many of the general purpose computing components, they are not designed to run operating systems.

Ardbeg PEs do not support operating system, but thread-level scheduling is still required for executing software programs on Ardbeg PEs. This is one of the key challenges in providing tool support for MPSoC architectures. Currently, it is still common practice in the industry for this task to be manually performed by the programmers. The key contribution of the SPIR compiler is its attempt to address this key challenge with a compiler. The SPIR compiler raises the compilation abstraction from assembly instruction to functions by assuming each function to be an atomic operation. It is a function-level compiler that performs scheduling and management for functions on the Ardbeg PEs. It assumes that the task management of the ARM control processor and the instruction-level scheduling of each individual Ardbeg PE are handled by traditional embedded operating systems and compilers.

7.2.2 Overall Compiler Infrastructure

The overall SPIR compilation flow is shown in Figure 7.3. The input is written in C with SPEX language extensions. The frontend translates the input into SPIR's PDF graph format. The focus here is to translate the sequential C semantics into the concurrent dataflow computation model. This is followed by dataflow scheduling and compiler optimization. In this step, kernel-to-processor assignments are done, execution scheduling is generated, and compiler optimizations are performed. The scheduling and optimizations annotate the results onto the SPIR graph, and make appropriate alterations to the graph itself. In the final step of the compilation flow, the compiler translates the SPIR graph back into SocC code. SocC programm language is a programming language extension of C, that is internally developed by ARM. In SocC, the code is explicitly parallelized into multiple threads. Processor assignments, memory allocations, and inter-process synchronization must be explicitly defined. The ARM SocC compiler then takes the SocC code,

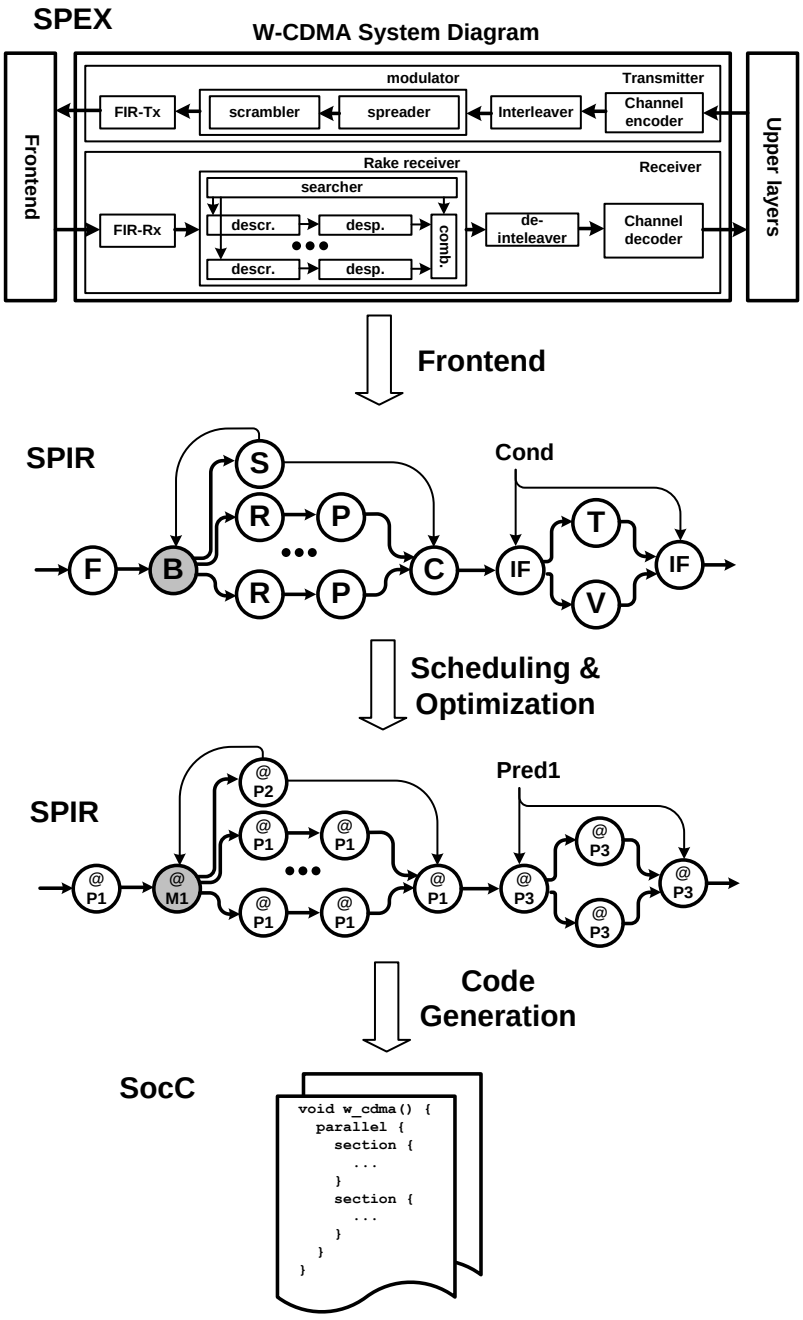


Figure 7.3: The overall SPIR compilation flow. The input is written in C with SPEX language extensions. The frontend translates the input into a SPIR dataflow graph. Dataflow scheduling and optimizations are applied to the SPIR dataflow graph by annotating the dataflow actors with processor assignments and memory allocations. The code generation then translate the SPIR graph into SocC multi-threading C code.

and compiles them into machine code for the Ardbeg MPSoC processor.

7.2.3 SPIR Intermediate Representation

The SPIR intermediate representation is one of the important contributions of this thesis. Traditional compiler IRs are designed to model instruction-level behaviors and interactions. They are ill-suited to described the inter-function interactions that are needed in a function-level compiler. The challenge is to choose a computation model that can capture the concurrent system-level operations in wireless protocols. In Chapter 6, section 6.2, we have proposed using a parameterized dataflow (PDF) computation model to describe wireless protocols. The SPIR intermediate representation is an implementation of this PDF model. The goal of the SPIR frontend is to translate the function-level parallelism in the source code into SPIR's PDF format. The goal of the SPIR backend is to exploit the function-level parallelism, and generate a multi-threaded C implementation of the dataflow graph.

7.2.4 Input and Output Language Formats

The SPIR compiler is a C-to-C compiler. It takes in SPEX code, and generates SocC code as output. Both SPEX and SocC are language extensions to C. The key difference is that SPEX is a sequential language extension, whereas SocC is a multi-threaded language extension. The key contribution of the SPIR compiler is that it automatically parallelizes sequential code into multi-threaded code for MPSoC architectures. In this section, we will describe SPEX and SocC in greater detail.

The SPEX Programming Language. SPEX is a programming language extension of C. Detail language semantics of the SPEX is described in Chapter 6. The dataflow computation model is more restrictive than the C programming language. Many of the

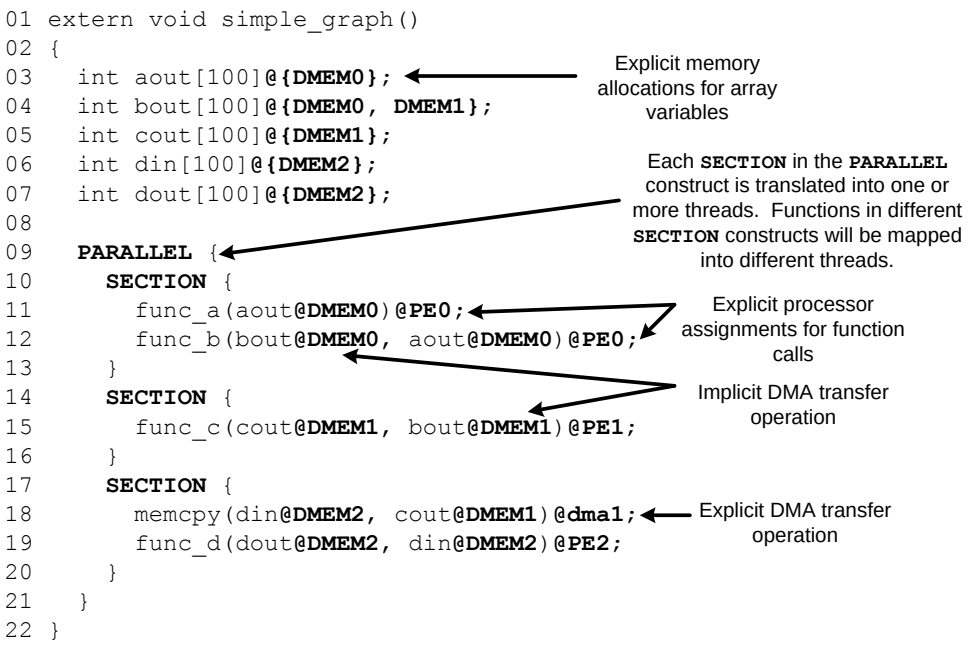


Figure 7.4: SocC programming example. SocC allows programmers to explicitly parallelize a program without the complexity of writing the code for explicit thread management. PE0-PE2 refer to the Ardbeg data processors. DMEM0-DMEM2 refer to Ardbeg data processors' local memories.

C language features cannot be translated into dataflow format. For example, variable pointers can be used as function arguments. C does not provide restrictions on the access pattern of pointers' memory locations. However, if we treat each function as a dataflow actor, then each function may only have either read-only or write-only function arguments. Therefore, only a subset of C semantics can be translated into dataflow – dataflow-safe C code. The purpose of SPEX is to provide a guidelines for the programmers to write dataflow-safe C code that can be translated into dataflow graphs.

SocC Language. The SocC programming language is a C extension developed by ARM for the Ardbeg processor [70]. It is a set of annotations that allows the programmers to explicitly parallelize a program without the complexity of writing the code for explicit thread management. It allows the programmers to define function-to-processor

assignments, assign physical memory locations for arrays, and specify synchronization buffers between communicating functions. Figure 7.4 shows an example of SocC code. On lines 3 to 7, the array variables are shown with the additional annotation `@{DMEMx}`. These annotations indicate to the compiler that these array variables should be allocated to the Ardbeg data processors' local memories. For example, line 3 has the code `int aout[100]@DMEM0;` This means that the `aout` variable is to be mapped onto the local memory of Ardbeg data processor 0. Line 9 to 21 are encapsulated in a `PARALLEL` construct, where there are multiple `SECTION` constructs. Each `SECTION` construct represents one or more threads that are executed on the Ardbeg ARM control processor. Different `SECTION` constructs are translated into different threads. Code within the same `SECTION` construct may also be broken into multiple threads, depending on the compiler implementation. Each function call is annotated with `@PEx`. For example, line 11 has the code `func_a(aout@DMEM0)@PE0`. This means that the function call `func_a` is to be executed on Ardbeg data processor 0. When the ARM control processor executes the thread that corresponds to this `SECTION`, it issues a Remote-Procedure-Call (RPC) to Ardbeg data processor 0 to execute function `func_a`. For data dependent producer-consumer functions that are mapped onto different processors, DMA operations can be either explicitly defined, as shown on line 18. DMA operations can also be implicitly defined by assigning the data dependent variable with two different memory locations, as shown on lines 4, 12, and 15. Synchronization directives are not shown in this example, but they are also supported in SocC.

The SocC compiler can automatically generate multiple threads that run on the Ardbeg ARM control processor that manages the execution of the data processors. However, it cannot automatically parallelize the code and generate the annotations. The responsibility of writing correct parallel code is still on the programmers. For wireless protocols

with complex streaming communication patterns, it becomes difficult for the programmers to determine the optimal processor assignment, and memory allocation. It is also difficult to insert the proper synchronization directives to ensure correct parallel execution. The SPIR compiler can be viewed as a frontend to the SocC compiler in the sense that it automatically generates correct parallelized SocC code based on programmers' sequential implementation. The goal of the SPIR compiler optimization and code generation is to generate correct and efficient SocC code from SPIR's dataflow representation.

7.2.5 Experimentation Infrastructure

The SPIR compiler is implemented within the SUIF compiler framework [37]. It is an instruction-level compiler that aims to parallelize sequential programs through array dependence analysis. Because the SPIR compiler is a function-level compiler, many of the SUIF optimizations are not used during the implementation. The relevant feature of the SUIF compiler is that it contains an open-source C-to-C compilation path. With our own modifications, we use the SUIF compiler to parse the SPEX input, and generate SocC output. The SUIF compiler uses its own proprietary intermediate representation – the SUIF IR. It has a translation path that converts C code into SUIF IR, as well as a translation path that converts SUIF IR into C code. The SPIR compiler interfaces with the SUIF compiler through the SUIF IR. The SPIR frontend converts SUIF code into SPIR format. And the SPIR code generation converts SPIR format back into SUIF code.

7.3 From SPEX to SPIR: Frontend Compilation

This section describes the frontend compilation process. The goal of the frontend is to translate sequential programs written in SPEX into concurrent PDF representations.

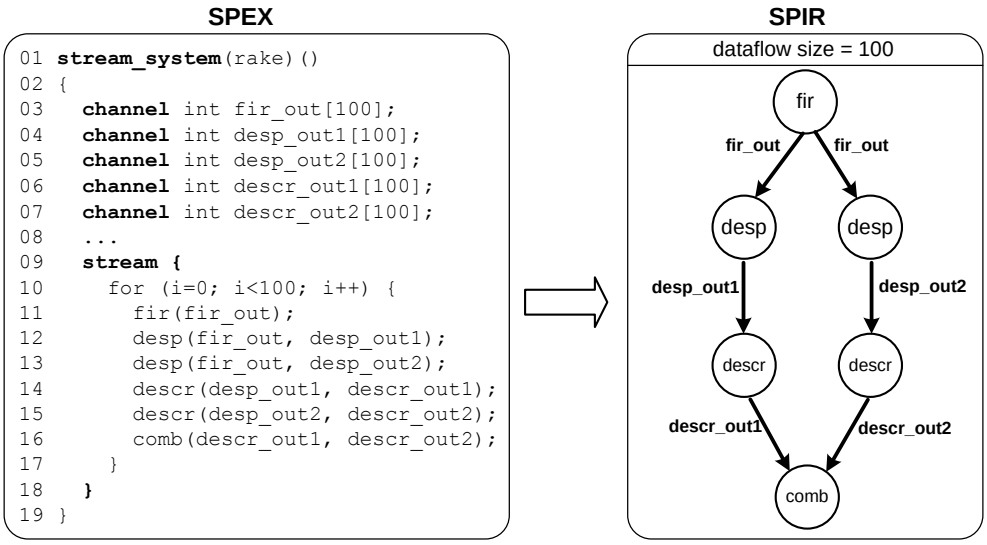


Figure 7.5: This diagram describes a simple stream construct written in SPEX, and its corresponding SPIR PDF representation.

Because many of the C language semantics cannot be translated into dataflow, SPEX can be viewed as a guideline for writing stylized C code that is dataflow-safe. This section describes the SPEX-to-SPIR translation process. It also provides rationales for the design decisions for the SPEX programming language.

The Stream Scope. The most important language feature in SPEX is the `stream` scope. This construct may only be declared within `stream_system` functions. It must contain one for-loop construct, with optional non-loop code before and after the loop construct. The for-loop construct must only contain function calls to `stream_kernel` functions. The SPIR frontend only compiles the code within the `stream` scope. It assumes the rest to be executed sequentially on the ARM control processor.

7.3.1 Basic Dataflow

Function Calls. In the simplest `stream` construct, it contains a counted for-loop on top of a list of `stream_kernel` function calls. Figure 7.5 shows an example of a simple

SPEX `stream` construct, and its corresponding SPIR dataflow representation. Unlike traditional C code, each function call within the `stream` scope creates its own dataflow actor. This is shown in the example with the `desp` and `descr` functions. Multiple function calls to the same function will result in multiple copies of that function. The frontend will not only duplicate the instruction code for each of the function calls, but also the data memory spaces as well. This is the reason why recursion for `stream_system` functions are not allowed.

Function Call Arguments. Only array variables that are declared with the `channel` keyword can be used as function arguments for these `stream_kernel` functions. As mentioned in Chapter 6, `stream_kernel`'s function arguments must be either read-only or write-only. This must be enforced by performing array access analysis of the function arguments within each `stream_kernel` function. Because each `channel` array corresponds to one dataflow edge, the frontend compilation must ensure that each `channel` array has only one producer function call.

Loops. The for-loop in the `stream` construct must be a counted loop with no side exits. Side exits, such as a `break` statement, do not have an equivalent representation in the PDF model. The restriction for the counted for-loop is meant for the compiler to be able to generate a loop iteration count during compile time. C allows the implementation of uncounted loops where the number of loop iterations is dependent on the computation within the loop body. There is no also no equivalent for this type of computation in the dataflow model. Because wireless protocols are streaming DSP applications, restricting the programmers to only write counted-loops does not restrict the expressiveness of the programming language.

7.3.2 Parameterized Dataflow

Parameter Variables. In SPEX, parameters are variables with a finite discrete set of run-time values. They are declared in SPEX with the `param` keyword. One of the jobs of the compiler frontend is to identify the set of possible values that each parameter may have during the run-time. Some of the values can be deduced through classic data dependency analysis of value assignments, comparison operations, and arithmetic operations. Special functions are also provided for the users to list the set of values for each parameter variable. If the compiler fails to identify a discrete set of values for a parameter, then a compilation flag will be raised.

Dataflow Initiation and Finalization. In SPIR's PDF model, each dataflow graph may have an optional dataflow init and dataflow final section. They correspond to the dataflow initialization and finalization stages of the SPIR's PDF model. In the SPEX code, they are the sequential code before and after the for-loop in the `stream` scope. They are akin to the C++'s constructor and destructor functions. Every time a dataflow graph is executed, it will first go through its dataflow initiation code, and then the finalization code. Parameter variables may only be modified in the initialization and finalization sections of the `stream` scope.

If-else Constructs. If-else constructs are allowed in the `stream` scope's for-loop structure. An example of this is shown in Figure 7.6. The frontend compilation inserts a pair of dataflow split and merge actors around each if-else construct. Nested if-else constructs are also supported. `channel` array variables are not allowed to have multiple producers in the `stream` scope. The only exception are the array variables declared within the if-else constructs. They are allowed to have two producers if the producers come from separate if and else paths. The frontend compiler must also check the usage pattern of variables used in the branch conditional statements. These conditional variables can only

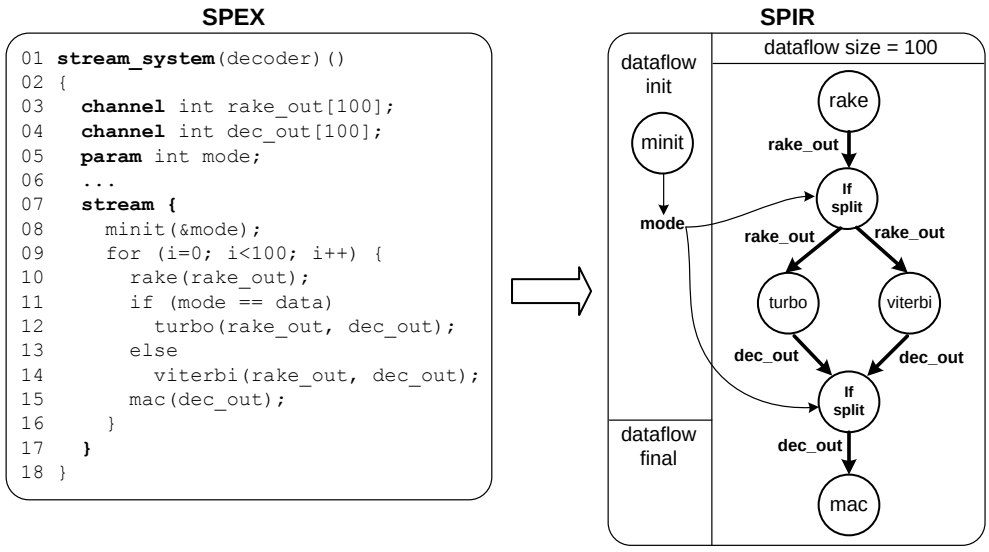


Figure 7.6: This diagram describes a stream construct with the if-else construct, and its corresponding SPIR boolean dataflow representation.

be modified in the dataflow initialization or finalization portion of the `stream` scope. In the example shown in Figure 7.6, the branch conditional variable is `mode`. It is modified on line 8 before the for-loop, but not within the for-loop. This is a restriction within SPIR's PDF model. In SPIR, the boolean dataflow's branch direction must be known before dataflow computation. This restriction has greater implication on the backend scheduling. A static dataflow schedule cannot be guaranteed if the branch condition is unknown. If we let the conditional variables to be defined anywhere within the for-loop, then the backend compilation path will have to provide a complex run-time system to propagate the conditional variables along with the data. By forcing the branch condition to be determined before or after the dataflow, the backend compilation can be greatly simplified by propagate the conditional variables only once before the dataflow computation.

ll-for Construct. As mentioned before, each function call within the `stream` scope creates a new dataflow actor. The `ll_for` construct allows the programmers to create multiple dataflow actors from the same `stream_kernel` function. An example is shown

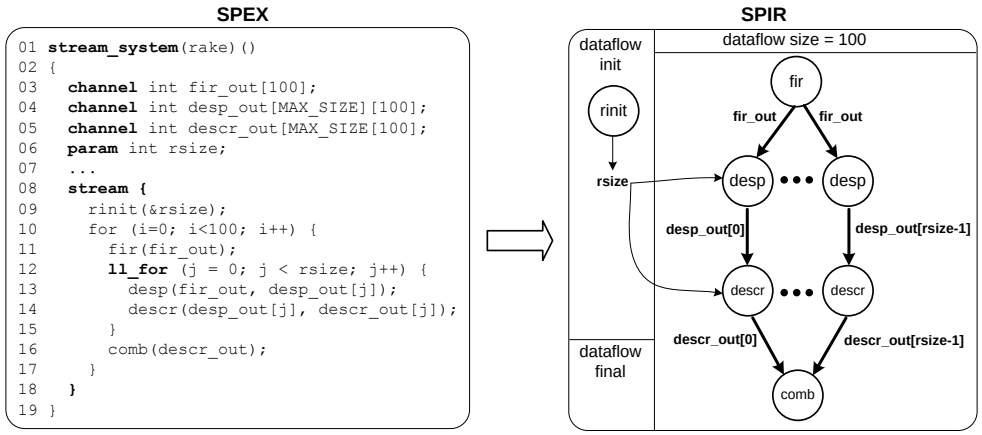


Figure 7.7: This diagram describes a stream construct with the ll-for construct, and its corresponding SPIR reconfigurable dataflow representation.

in Figure 7.7. The `ll_for` loop must be a counted loop. Its loop bounds must be either constant, or a parameter variable declared during the dataflow initiation or finalization stages. The compiler frontend must make sure that there are no data dependencies across loop iterations.

Parameterized Loop Bounds. As mentioned before, the `stream` scope must contain only counted for-loop. In SPEX, the loop bound can be also be described with a parameter variable. Like all parameter variables, loop bound variables must be defined in the dataflow initialization or finalization section. And they are not allowed to be modified within the loop.

7.4 Function-level Scheduling and Optimizations

In this section, we present the overall implementation considerations for scheduling functions. In addition, we present a function-level compiler optimization: function-level software pipelining is an adaptation of an existing instruction-level compilation technique.

7.4.1 Scheduling Overview

Previous work [50] has shown that the multi-processor scheduling problem can be divided into three major tasks: 1) processor assignment and memory allocation; 2) kernel execution ordering; and 3) kernel execution timing. All three tasks can be handled either statically by the compiler or dynamically by the run-time scheduler. In wireless communication protocols, the execution behavior is relatively static with limited run-time execution variation. The scheduling process needs to consider the inter-kernel communications, meet the real-time deadlines, and manage the scratchpad memories. This combination of factors favors a compile-time solution. Because of the run-time variations, it is impossible to completely determine the kernel execution timing during compile-time. Thus, we focus on designing a scheduler for the first two tasks. Coarse-grained function-level scheduling under strict memory constraints presents new challenges that have not been fully explored in previous compilation studies.

Kernel Profiling. Kernels form the building blocks for the SDR protocol dataflow graph. To make coarse-grained scheduling decisions, execution information about each kernel is required. Kernels are compiled and profiled individually on each of the processor types available on the MPSoC. Kernel profiles are entered in a queryable format, so that later scheduling stages can easily access the information.

Dataflow Rate Matching. The dataflow computation model allows unmatched rates on the dataflow edges. There exists a large body of previous work for dataflow rate matching algorithms [51]. However, as a side-effect of the SPEX programming language, the IR generated by the frontend has matched rates. These rates correspond to the sizes of `channel` array variables. Therefore, dataflow rate matching is currently not required in the SPIR compilation flow. This does not mean that dataflow rate matching is useless. As a part of our future work, we envision SPIR to support other high level languages that

may produce unmatched dataflow rates.

7.4.2 Coarse-grained Software Pipelining

Coarse-grained compilation requires function-level parallelism to utilize a MPSoC's resources. Wireless communication protocols do not have many kernels that execute concurrently. They are streaming applications with coarse-grained pipeline-level parallelism. Software pipelining was proposed as a method to exploit the instruction-level parallelism by overlapping consecutive loop iterations. Stream computation can be viewed as a loop that iterates through each streaming data input, where the computation for successive data inputs can also be overlapped. The coarse-grained scheduling process is similar to instruction-level software pipelining, except that kernels and bulk memory transfers are scheduled onto processors and DMA engines, instead of scheduling instructions onto ALUs and memory units. Modulo scheduling [69] is a well-known software pipelining algorithm that can achieve very good solutions. In this section, we present a coarse-grained modulo scheduling algorithm used to schedule a rate matched hierarchical dataflow graph on to a MPSoC. Similar to instruction-level modulo scheduling, coarse-grained modulo scheduling has to honor resource and dependency constraints between dataflow actors. However, coarse-grained modulo scheduling differs from traditional modulo scheduling in the following ways.

Storage assignment. In traditional modulo scheduling, allocation of storage (e.g., rotating registers) used for carrying values between operations is performed as a post-processing step. Enough storage is assumed to be available during the scheduling phase, while the register allocation phase does the actual storage allocation. In coarse-grained modulo scheduling, memory buffers must be allocated on the processors where dataflow actors are scheduled. Typically, the local memory available on processors is limited. Also,

MPSoCs can have processing elements with varying memory capacities. This limited non-uniform distribution of memories makes the storage assignment a first-class scheduling constraint. Postponing storage assignment to a later phase results in the scheduler making aggressive decisions about actor placements on processors. Consequently, storage assignment fails in many cases. Therefore, in the coarse-grained modulo scheduling method presented, scheduling and storage assignment are performed in a single phase.

Scheduling data movement. Traditional modulo scheduling assumes that the value written to the register by an operation is available to dependent operations in the very next cycle. This is because the register file is connected to all function units. However, in a MPSoC, processors have their own local memories and the data is transported between processors. DMA operations used for moving the data between processors take significant amount of time, and dependent operations must wait for the DMAs to complete before they can begin execution. Thus, unlike traditional modulo scheduling, the coarse-grained modulo scheduler must explicitly schedule the DMA operations used for moving data between processors.

II Selection. In modulo scheduling, II (initiation interval) is the interval between the start of successive iterations. The minimum initiation interval (MII) is defined as $MII = Max(ResMII, RecMII)$, where ResMII is the resource constrained MII, and RecMII is the recurrence constrained MII. In coarse-grained modulo scheduling, ResMII is defined by the total latency of all actors in the graph divided by the number of processors allocated to the graph. RecMII is defined by the maximum latency of each feedback path. Since SDR protocols are real-time applications, the scheduler must also take timing constraints into consideration. In W-CDMA, the timing constraint is defined by the overall data throughput, which is 2Mbps as the output rate of the receiving data channel. If we use W-CDMA as an example, then the maximum II must be 610K clock cycles on

a MPSoC with 400MHz data processors as an example. Like instruction-level modulo scheduling, the II selection process starts at MII, and is iteratively increased until all of the actors are scheduled or the maximum II is reached. If the maximum II is reached and no valid schedule is found, then a failure message is returned.

Kernel-To-Processor Assignment. For each II, the modulo scheduler assigns actors to processors and allocates DMA buffers for producer/consumer actor pairs that are assigned to different processors. We use a greedy heuristic which naively assigns actors to processors based only on execution latencies and II. One of the key challenges is scheduling the conditional dataflow actors. Because the if-branch and else-branch are never executed at the same time, the scheduler should assign the same hardware resources for these two paths. In our scheduler, we first schedule the dataflow path with the longest execution latency. We then schedule other dataflow paths with the same hardware resources that we assigned for the longest path.

In scheduling the longest execution latency path, its actors are first sorted into a linear list that is ordered by the data dependencies between the actors. For any pair of producer/consumer datalow actors, the producer actor is always placed into the list in front of the consumer actor. Actors with the same producer are ordered arbitrarily in the list. The list is then scheduled onto the processors using a greedy bin-packing algorithms. In this algorithm, the list of actors is broken into N sub-lists, where N is the dataflow path's execution latency divide by II. Each sub-list is assigned sequentially to the processor with the lightest workload, with consecutive software pipeline stage numbers.

Actors that are not scheduled from the longest execution path scheduling must be part of an if-else branch. They are assigned based on the scheduling results of the longest latency dataflow path. The goal is to assign the same set of processors and software pipeline stages for both paths of the same branch. This requires that one of the two paths

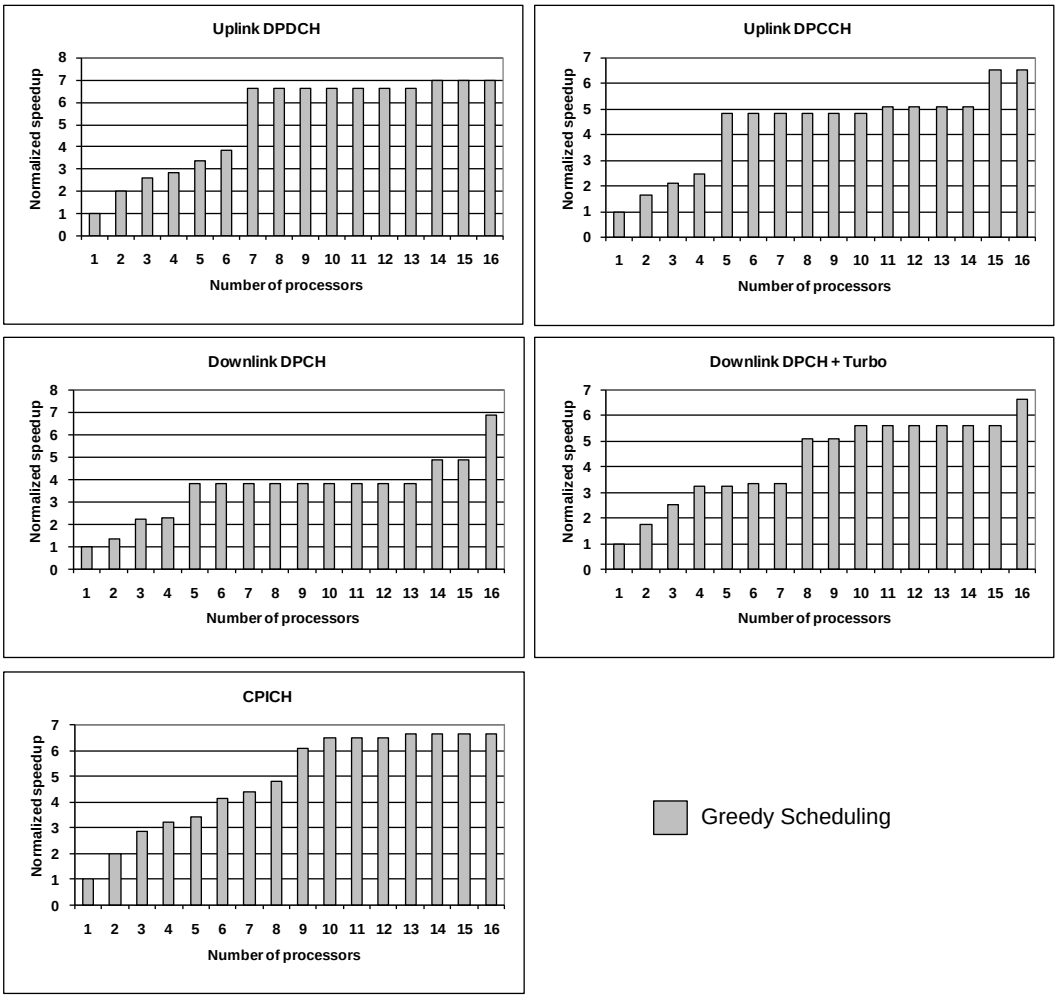


Figure 7.8: Execution speedup for W-CDMA benchmarks compiled by greedy modulo scheduler running on 1 to 16 data processors.

to be scheduled first. Therefore, the scheduling first starts from if-else branches where one of the two paths is scheduled as a part of the longest execution latency path. It then iteratively scheduled the rest of the if-else branches if there are nested branches. After all of the actors are scheduled, DMA operations are assigned to each SPIR edge where the source and destination actors have different processor assignments or software pipeline stages.

Experimental Results W-CDMA protocol specifications [41] define multiple transmission modes for different purposes, ranging from data and voice transmissions to syn-

chronization. In this study, we picked five operating modes that cover the essential W-CDMA operations, and handcoded them in SPIR. These five modes are: downlink DPCH(Dedicated Physical CHannel); uplink DPCCH(Dedicated Physical Control CHannel); uplink DPDCH (Dedicated Physical Data CHannel); and CPICH (Common Pilot CHannel). Downlink DPCH is the main data receiver channel, it is time-multiplexed between receiving protocol control and user data. We included two versions of the downlink DPCH, one with the Turbo decoder and one without. This is because many proposed SDR solutions still use Turbo ASIC accelerators [81] due to its high computation requirements. Uplink DPCCH and DPDCH are the transmitter counter-parts of the downlink DPCH. And finally, CPICH is the synchronization channel, which is used to measure signal strength and synchronize data transmission.

Figure 7.8 shows the overall execution speedup for the W-CDMA benchmarks compiled with the greedy modulo scheduler, running on 1 to 16 data processors with 1 control processor. The execution speedup is normalized to the execution time of the benchmarks running on 1 data processor. For all of the benchmarks, the modulo scheduler achieves near-linear speedup up to 4 processors. However, it cannot efficiently utilize more than 5 processors, even though there are many more kernels in the benchmarks. The reason is because there are a few bottleneck algorithms, such as filter, searcher, and Turbo decoder, that require much more computational resources then the rest of the algorithms. Therefore, even though there are many processors available, the majority of the time are spend waiting for the bottleneck algorithms to finish.

7.5 From SPIR to SocC: Code Generation

The final step in the SPIR compiler is the SocC code generation. In this step, the SPIR PDF format is converted back into C with the SocC language extensions. This is done with the aid of the SUIF compiler's C code generation path. The SPIR compiler produces code in SUIF intermediate format, which is then converted back into C using SUIF. Through the function-level scheduling and software pipelining optimizations, each dataflow actor is annotated with a Ardbeg PE number and the software pipeline stage count. The main focus of the code generation step is to translate these annotations into multi-threaded format. In the rest of the section, we are going to talk about the two major components of the translation process: predication generation for conditional dataflow and DMA operation generations.

7.5.1 Predicated Execution

SPIR's PDF model supports conditional dataflow with if-else split and merge actors. The code generation must insert proper control code to make sure the correct path executes during run-time. This is similar to instruction-level predicated execution, where predicated instructions execute only if its predicate is true during the run-time. Even though the granularity is a function in the SPIR compiler, the basic technique for generating predicates can be adopted with very little modification. In this compiler, we use an algorithm first proposed in [29]. This particular algorithm is used because it minimizes the number of predicates used. The predicates are converted into if-statements in the output SocC code.

The predicate value itself is always stored in the ARM control processor, and is sent to the PEs through explicit DMA operations. Because the SPEX language does not allow

the values of if conditions to be modified within the for-loop of the `stream` scope, the SPIR compiler does not have to handle inter-PE predicate propagation. The results of the dataflow are always send back from the data PEs to the control processor, and the predicates are updated atomically on the control processor.

7.5.2 Memory Buffers and DMA Operations

With the software pipeline scheduler, each dataflow actor in the SPIR graph is assigned a processor and a stage number. The stage number is the pipeline stage that the dataflow actor executes. If a pair of producer-consumer actors are assigned to different stages or different processors, then memory buffers must be inserted to ensure correct pipeline execution. Different processor assignments also need explicit DMA operations. For function-level software pipelining, the double-buffering technique is used between pipeline stages. In the double-buffering technique, each pipeline buffer duplicated into two buffers. While the function is writing or reading from one buffer, the content of the other buffer is accessed by the DMA for inter-processor data transfer. Both the producer and the consumer are required to have double-buffering. Therefore, four buffers are required for two functions that are separated by one pipeline stage. If the functions are separated by more than one stage, each additional stage only requires one additional buffer. Currently, we put these additional buffers in the local memory of the consumer function. Naturally, compiler optimizations can be applied for better buffer-to-memory allocation. We intend to investigate this as a part of our future work.

7.5.3 SocC Output Example

Figure 7.9 is the implementation of a simple feedforward dataflow containing two actors. This dataflow is implemented with SPEX, SPIR, and SocC. The SPEX imple-

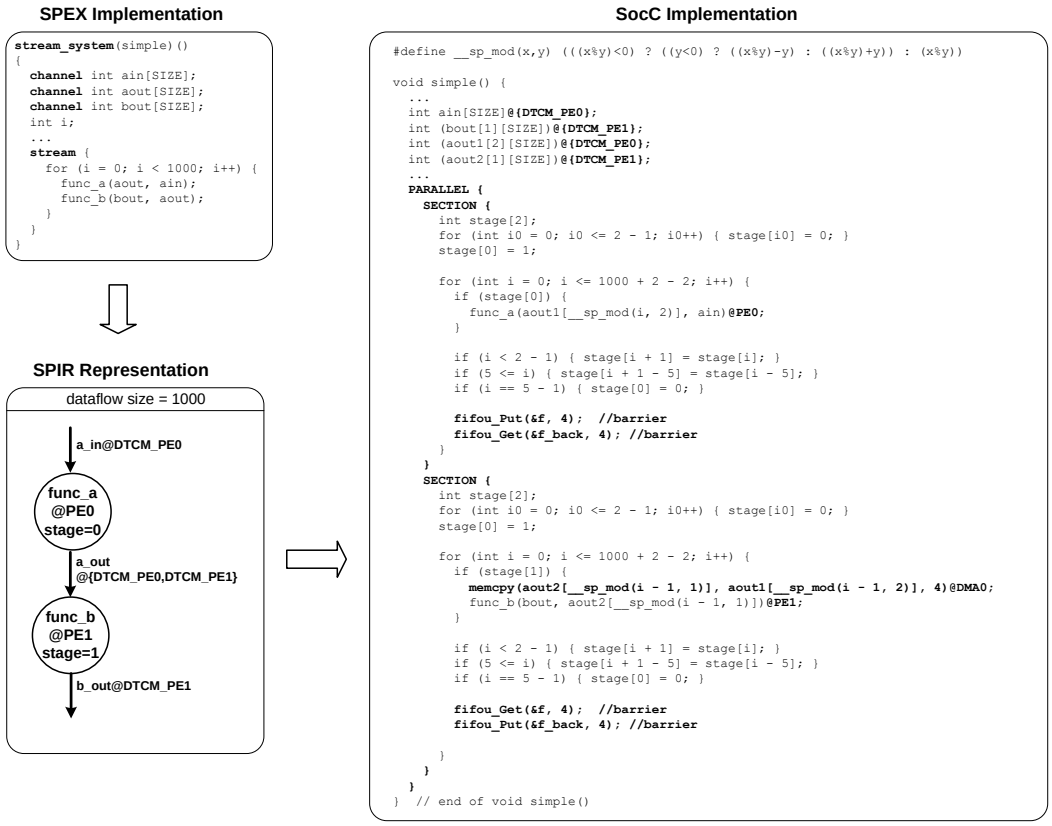


Figure 7.9: SPEX, SPIR, and SocC implementation of a simple feedforward dataflow containing two actors. The SPEX implementation is the input of the SPIR compiler, and the SocC implementation is the output of the SPIR compiler. In the SocC implementation, the dataflow is mapped onto two Ardbeg PEs, and is software pipelined into two stages.

mentation is the input of the compiler, the SPIR implementation is in the IR format, and the SocC implementation is the output of the compiler. In the SocC implementation, the dataflow is mapped onto two Ardbeg PEs, and is software pipelined into two stages. `func_a` is mapped onto PE0, executing in the first software pipeline stage. `func_b` is mapped onto PE1, executing in the second software pipeline stage. As shown in the figure, the SocC code is divided into two sections. Each section corresponds to one thread of execution. DMA operations and synchronization primitives are added to the SocC code. This example demonstrates the importance of the SPIR compiler. Even for a simple dataflow, the resulting multi-threading SocC code contains non-trivial implementation

details. For complex dataflow graphs, the SocC code can quickly become unmanageable.

7.6 Related Work

Dataflow Scheduling. There has been numerous compilation projects on mapping dataflow graphs onto multi-core processors. Depending on the underlying dataflow model, some requires run-time system support, while others can generate compile-time schedules. In the MIT StreamIt compiler [34], the underlying model is based on the Synchronous Dataflow (SDF) model. They have examined the different static dataflow scheduling algorithms in [43], and their impacts on the run-time execution. There are other projects, such as the Ptolemy [66], PeaCE [36], and DIF [67], that support multiple different dataflow models. This means that they cannot generate fully static run-time schedules, and have to provide run-time scheduler for run-time execution of dataflow actors.

Compilation Support for Multi-core DSP Processor. There has been numerous compilers for other multi-core DSP processors. The IBM Cell compiler is the most relevant to this study because its architecture [40] is the most similar to the Ardbeg processor architecture. Most of the IBM Cell compilation effort is focused on provided efficient single PE performance through various data-level parallelization techniques [27]. [74] advocates a multi-tier programming approach, which is similar to this thesis's proposed two-tiered compilation approach. However, it does not provide a compilation system that supports automatic code generations. More recent effort [47] has started to examine the function-level compilation methodology. There are other multi-core compilers that are not based on compiling dataflow models. These include the compiling the Brook streaming language onto multi-core processors [53], loop-centric parallelizing compiler for Vector-thread architecture [38], and basic-block level parallelization for the TRIP EDGE

architecture [76].

Software Pipelining. In the compiler domain, modulo scheduling is a well known software pipelining technique [69]. There has been previous work purposing constraint-based modulo scheduling, including [28], and [11]. But all of these techniques are geared toward instruction-level modulo scheduling. [72] extends the modulo scheduling to software pipeline any loop nest in a multi-dimensional loop, which conceptually is similar to coarse-grained modulo scheduling. To our knowledge, there have not been any previous work exploring coarse-grained modulo scheduling for MPSoC architectures. However, the idea of coarse-grained software pipelining has been explored before. [25] has proposed an algorithm that automatically breaks up nested loops, function calls, and control code into sets of coarse-grain filters based on a cost model. And, these sets of filters are then generated for parallel execution. [24] has proposed of using function-level software pipelining to stream data on the Imagine Stream Architecture. [35] also explored the idea of coarse-grained software pipelining on a tiled architecture.

7.7 Summary

In this section, we present the SPIR compiler infrastructure. SPIR translates sequential SPEX code into concurrent PDF format. It performs function-level software pipelining on the dataflow, and generates multi-threaded SocC code as output. This chapter demonstrates the feasibility of building a two-tier compilation approach for automatically mapping sequential programs onto a multi-core DSP architecture.

CHAPTER 8

Conclusion

From 3G wireless communications to high definition videos, digital signal processing(DSP) has already become an integral part of our everyday lives. Within the past twenty years, engineers have designed increasingly complex DSP systems in order to satisfy our growing appetites for faster and better multimedia content. The up-and-coming multimedia applications pose a new design challenge for computer engineers. They have computation requirements beyond existing desktop computers, while also requiring the power efficiency of hand-held devices. We label these applications as mobile supercomputing.

8.1 Summary

This thesis presents a case study for designing a solution for realizing one such mobile supercomputing application — Software Defined Radio (SDR). In recent years, we have seen an increase in the number of wireless protocols that are applicable to different types of communication networks. Traditionally, the physical layer of these wireless protocols is implemented with fixed function ASICs. SDR promises to deliver a cost effective and

flexible solution by implementing a wide variety of wireless protocols in software. Such solutions have many potential advantages: 1) Multiple protocols can be supported simultaneously on the same hardware, allowing users to automatically adapt to the available wireless networks; 2) Lower engineering and verification efforts are required for software solutions over hardware solutions; 3) Higher chip volumes because the same chip can be used for multiple protocols, which lowers the cost; and 4) Better support for future protocol changes. With the tremendous benefits of SDR, it is likely that many mobile communication devices are going to be supported by SDR technologies in the foreseeable future.

Because SDR is an interdisciplinary research topic, this thesis examines multiple research subjects under the overall objective of realizing SDR: computer architecture, DSP algorithm optimizations, programming language design, and compiler construction. The detailed contributions are listed as follows:

1. **Processor Design.** This thesis proposes a multi-core DSP architecture, SODA, for supporting SDR. SODA consists of one control processor, four data processors, and a shared global memory. The control processor is an embedded general purpose processor that is capable of handling the control-intensive code that is used to manage the overall baseband processing system. The data processors are specialized DSP processors that can perform data-intensive computations. A commercial SDR processor based on the SODA processor architecture has been developed by ARM Ltd. The Ardbeg processor is also a multi-core DSP processor consists of 32-lane SIMD data processors. This thesis provides an detailed comparison study between the SODA and Ardbeg processors. This study reconfirms many of the SODA architectural decisions. It also reveals many design shortcomings of SODA, and explains the subsequent design improvements in Ardbeg.

2. **Algorithm Implementations.** Each DSP algorithm in W-CDMA is hand coded and optimized for the SODA data processor. The majority of wireless protocols' algorithms operate on large vectors, and are therefore a good fit for the wide-SIMD design. This thesis validates this claim by demonstrates the implementation of key DSP algorithms on SODA. DSP algorithms usually have multiple different implementations. Not all implementations can be mapped efficiently due to the wide SIMD design. This thesis describes a set of DSP algorithm implementations that map well on the SODA architecture.

3. **Language and Compiler Support.** This thesis also proposes a programming language and compilation flow for mapping wireless protocols onto the Ardbeg processor. The programming language, SPEX, is based on the parameterized dataflow computation model. And the compiler, the SPIR compiler, is a function-level compiler, where the granularity of an atomic execution unit is a function, not an instruction. The SPIR compiler takes sequential code written in SPEX, and generates multi-threaded C code as output. The multi-threaded C code is then compiled onto the Ardbeg processor using ARM's internal compiler system.

8.2 Future Work

The following topics are potential future research directions to extend this work.

Mobile Supercomputing. This thesis has demonstrated the feasibility of supporting one mobile supercomputing application with the SODA processor. There are other mobile supercomputing applications that may also benefit from SODA processor's high computational efficiency. These include high definition video, 3D graphic processing, and physics simulation for gaming. We have studied the H264 video coding application. Our initial analysis finds that it shares many of the same computation characteristics as wireless communications. Many of its algorithms can benefit from SODA's wide SIMD design. However, there are also some key differences. The computation patterns in video coding are 2D matrix operations, as compared to wireless communication algorithms' 1D vector operations. This requires better support for complex data permutations and higher memory bandwidth. The challenge is to provide support for these operations while still operate within the power budget of a mobile device.

Function-level Compilation. The work on the SPIR compiler is just our first attempt at designing a function-level compiler for multi-core DSP processors. There are many areas of the compilation process that can be improved. The frontend compilation process can be made more intelligent to parse more generic C code instead of the stylized

SPEX code. It is also possible to provide frontend support for other popular programming languages, such as Matlab/Simulink, Verilog and SystemC. SPIR currently only provides limited backend function-level optimizations. There are many other interesting optimizations that are not implemented. Some of these optimizations include function fusion or fission for better load balancing, and an intelligent scheduler for dealing with run-time execution variations. Finally, we are also interested in providing compilation support for other multi-core processors beyond Ardbeg, such as the IBM Cell processor and programmable graphics processors.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ANSI/IEEE Std 802.11, 1999 Edition, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

[2] ARM Cortex-M3. `http://www.arm.com/products/CPUs/ARM-Cortex-M3.html`.

[3] ARM Neon Technology. `http://www.arm.com/products/CPUs/NEON.html`.

[4] DSP Developers' Village, Texas Instruments. `http://dspvillage.ti.com`.

[5] ETSI TR 101 190: Digital Video Broadcasting(DVB); Transmission System for Handheld Terminals(DVB-H).

[6] Predictive Technology Model. `http://www.eas.asu.edu/ptm/`.

[7] QuickSilver Technology. `http://www.qstech.com/`.

[8] Global Cellphone Penetration Reaches 50 Pct. *Reuter*, Nov. 2007.

[9] Samsung, NXP, and T3G Showcase World's First TD-SCDMA HSDPA/GSM Multimode Mobile Phone. Oct 2007.

[10] Jung Ho Ahn et al. Evaluating the Imagine Stream Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[11] E.R. Altman and G.A. Gao. Optimal Modulo Scheduling Through Enumeration. In *International Journal of Parallel Programming*, pages 313–344, 1998.

[12] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, C. Chakrabati, and Wayne Wolf. Mobile supercomputers. *Communications of the ACM*, May 2004.

[13] Rupert Baines and Doug Pulley. Software defined baseband processing for 3G base stations. In *4th International Conference on 3G Mobile Communication Technologies (Conf. Publ. No. 494)*, pages 123–127, June 2003.

[14] C. Berrou and A. Glavieux. Near Optimum Error Correcting Coding and Decoding: Turbo-codes. In *IEEE Transactions on Communications*, volume 44, no. 10, pages 1261–1271, Oct. 1996.

[15] B. Bhattacharya and S. S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. pages 2408–2421. IEEE Transactions on Signal Processing, Oct. 2001.

[16] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, volume 26, no. 4, pages 52–60, Jul/Aug 2006.

[17] H. Bluethgen et al. A Programmable Baseband Platform for Software Defined Radio. In *Proc. of the 2004 SDR Technical Conference*, Nov.

[18] E. Boutillon, W. Gross, and P. G. Gulak. VLSI Architectures for the MAP Algorithm. In *IEEE Trans. on Communications*, volume 51, no. 2, pages 175–185, Feb. 2003.

[19] I. Buck. Brook Language Specification. In *http://merrimac.stanford.edu/brook*, Oct. 2003.

[20] J. T. Buck and E. A. Lee. Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model. Proc. Int. Conf. Acoust., Speech, Signal Processing, April 1993.

[21] N. Chandrachoodan and S. S. Bhattacharyya. The Hierarchical Timing Pair Model for Multirate DSP Applications. In *IEEE Transactions on Signal Processing*, volume 52, no. 5, May 2004.

[22] Inching Chen, Anthony Chun, Ernest Tsui, Hooman Honary, and Vicki Tsai. Overview of Intel's Reconfigurable Communication Architecture. In *3rd Workshop on Application Specific Processors*, pages 95–102, Sept. 2004.

[23] N. Clark et al. OptimoDE: Programmable Accelerator Engines Through Retargetable Customization. In *Proc. Hot Chips 6*, Aug 2004.

[24] A. Das, W. Dally, and P. Mattson. Compiling for Stream Processing. In *Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2006.

[25] W. Du, R. Ferreira, and G. Agrawal. Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism. In *Supercomputing Conference (SC)*, Nov. 2003.

[26] W.J. Ebel. Turbo-Code Implementation on C6x. In *Tech. Report, Alexandria Research Inst., Virginia Polytechnic Inst. State Univ.*, 1999.

[27] A. E. Eichenberger et al. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. In *IBM System Journal*, volume 45, No. 1, pages 59–84, 2006.

[28] A.E. Eichenberger and E. Davidson. Efficient Formulation For Optimal Modulo Schedulers. In *Proc. of Programming Language Design and Implementation*, pages 194–205, June 1997.

[29] J. Z. Fang. Compiler Algorithms on If-Conversion, Speculative Predicates Assignment and Predicated Code Optimization.

[30] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov. 2006.

[31] J. Fridman and Z. Greenfield. The TigerSharc DSP architecture. In *IEEE Micro*, pages 66–76, Jan. 2000.

[32] F. Gilbert, M. J. Thul, and N. Wehn. Communication Centric Architectures for Turbo-Decoding on Embedded Multiprocessors. In *Proc. Design, Automation and Test Europe*, pages 356–461, Mar. 2003.

[33] John Glossner, Erdem Hokenek, and Mayan Moudgill. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[34] M. Gordon et al. A Stream Compiler for Communication-Exposed Architecture. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[35] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[36] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo. PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems. In *ACM Transactions on Design Automation of Electronic Systems*, Aug. 2007.

[37] M.W. Hall et al. Maximizing Multiprocessor Performance with the SUIF Compiler. In *IEEE Computer*, Dec. 1996.

[38] M. Hampton and K. Asanovic. Compiling for Vector-Thread Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[39] Kenichi Higuchi and Hidekazu Taoka. Field Experiments of 2.5 Gbits/s High-Speed Packet Transmission Using MIMO OFDM Broadband Packet Radio Access.

[40] Peter H Hofstee. All About the Cell Processor. In *IEEE Symposium on Low-Power and High-Speed Chips(COOL Chips VIII)*, April 2005.

[41] Harri Holma and Antti Toskala. *WCDMA for UMTS: Radio Access For Third Generation Mobile Communications*. John Wiley and Sons, LTD, New York, New York, 2001.

[42] C. Hsu, I. Corretjer, M. Ko, W. Plishker, and S. S. Bhattacharyya. Dataflow interchange format: Language reference for DIF language version 1.0, users guide for DIF package version 1.0. In *Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park*, June 2007.

[43] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased Scheduling of Stream Programs. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003.

[44] F. Kienle, H. Michel, F. Gilbert, and N. Wehn. Efficient MAP-algorithm Implementation on Programmable Architectures. In *Advances in Radio Science*, volume 1, pages 259–263, 2003.

[45] S. Knowles. Ths SoC Future is Soft: http://www.iee-cambridge.org.uk/arc/seminar05/slides/SimonKnowles.pdf.

[46] Christoforos Kozyrakis and David Patterson. Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 283–293, Nov. 2002.

[47] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[48] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[49] E. A Lee. Overview of the Ptolemy Project. In *Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley*, July 2003.

[50] E. A Lee and S. Ha. Scheduling Strategies for Multiprocessor Real-time DSP. In *Global Telecommunications Conference*, pages 1279–1283, Nov. 1989.

[51] E.A. Lee and D.G. Messerschmidt. Synchronous Data Flow. In *Proc IEEE*, 75, 1235-1245 1987.

[52] Hyunseok Lee et al. Software Defined Radio - A High Performance Embedded Challenge. In *Proc. 2005 Intl. Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, Nov. 2005.

[53] S. Liao, Z. Du, G. Wu, and G.Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[54] Y. Lin, Yoonseo Choi, Scott Mahlke, Trevor Mudge, and Chaitali Chakrabarti. A Parameterized Dataflow Language Extension for Embedded Streaming Systems. In *Proceedings of IC-SAMOS VIII*, July 2008.

[55] Y. Lin et al. Design and Implementation of Turbo Decoders for Software Defined Radio. In *Proc. IEEE 2006 Workshop on Signal Processing Systems (SiPS).*

[56] Y. Lin et al. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[57] A. Lodi et al. XiSystem: A XiRisc-Based SoC With Reconfigurable IO Module. In *IEEE Journal of Solid-State Circuits*, volume 41, No. 1, pages 85–96, Jan. 2006.

[58] M. Mansour and N. Shanbhag. VLSI Architectures for SISO-APP Decoders. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 11, no. 4, pages 627–650, Aug. 2003.

[59] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable Matrix. In *13th International Conference on Field-Programmable Logic and Applications*, Jan. 2003.

[60] H. Michel, A. Worm, M. Munch, and N. Wehn. Hardware/Software Trade-offs for Advanced 3G Channel Coding. In *Proc. Design, Automation and Test Europe*, pages 396–401, Mar. 2002.

[61] P.K. Murthy and E. A. Lee. Multidimensional Synchronous Dataflow. In *IEEE Transactions on Signal Processing*, August 2002.

[62] Y. Neuvo. Cellular Phones as Embedded Systems. In *IEEE International Solid-State Circuits Conference*, 2004.

[63] C. Park, J. Chung, and S. Ha. Extended Synchronous Dataflow for Efficient DSP System Prototyping. pages 295–322. Design Automation for Embedded Systems, Kluwer Academic Publishers, March 2002.

[64] T. M Parks, J. Luis Pino, and E. A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. In *Asiolmar Conference on Signals, Systems and Computers*, October 1995.

[65] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. In *IEEE Micro*, volume 16, no. 4, Aug. 1996.

[66] J. L. Pino, S. Bhattacharyya, and E. Lee. A Hierarchical Multiprocessor Scheduling System for DSP Applications. In *Twenty-Ninth Annual Asilomar Conference on Signals, Systems, and Computers*, Oct 1995.

[67] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *In Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, July 2008.

[68] U. Ramacher. Software-Defined Radio Prospects for Multistandard Mobile Phones. *Computer*, 40(10):62–69, 2007.

[69] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelined Loops. In *Proc. of 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[70] A. Reid, Y. Lin, K. Flautner, and E. Grimley-Evans. SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Oct. 2008.

[71] Scott Rixner et al. Register Organization for Media Processing. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 375–386, Jan. 2000.

[72] H. Rong et al. Single-Dimension Software Pipelining for Multi-Dimensional Loops. In *Proc. of the International Symposium on Code Generation and Optimization*, March 2004.

[73] A. La Rosa, L. Lavagno, and C. Passerone. Implementation of a UMTS Turbo Decoder on a Dynamically Reconfigurable Platform. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, no. 1, pages 100–106, Jan. 2005.

[74] R. Sakai et al. Programming and Performance Evaluation of the Cell Processor. In *Hotchip 17*, August 2005.

[75] M. Schneider, H. Blume, and T.G. Noll. Power Estimation on Functional Level for Programmable Processors. In *Advance in Radio Science*, volume 2.

[76] A. Smith et al. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[77] R. Staszewki, K. Muhammad, and P. Balsara. A 550MSamples 8-tap FIR Digital Filter for Magnetic Recording Read Channels. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2000.

[78] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState — An Internal Representation for Codesign. Proc. International Conference on Computer Aided Design, Nov. 1999.

[79] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the 2002 International Conference on Compiler Construction*, June 2002.

[80] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport Messaging for Distributed Stream Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

[81] C.H. van Berkel et al. Vector Processing as an Enabler for Software-Defined Radio in Handsets From 3G+WLAN Onwards. In *Proc. 2004 Software Defined Radio Technical Conference*, Nov. 2004.

[82] A. Waksman. A Permutation Network. In *Journal of the ACM*, volume 15, No. 1, pages 159–163, Jan. 1968.

[83] Z. Wang, Z. Chi, and K. Parhi. Area-Efficient High-Speed Decoding Schemes for Turbo Decoders. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 10, no. 6, pages 902–912, Dec. 2002.

[84] M. Woh et al. The Next Generation Challenge for Software Defined Radio. In *International Symposium on Systems, Architecture, Modeling and Simulation (SAMOS)*, July 2007.