NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: A Hardware/Software System for Adaptive Beamforming

Author: Albert Anthony Conti III

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Eric Miller	Date
Thesis Reader: Prof. Laurie King	Date
Thesis Reader: Sarah Leeper	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: A Hardware/Software System for Adaptive Beamforming

Author: Albert Anthony Conti III

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Eric Miller	Date
Thesis Reader: Prof. Laurie King	Date
Thesis Reader: Sarah Leeper	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

Copy Deposited in Library:

Reference Librarian	Date

A Hardware/Software System for Adaptive Beamforming

A Thesis Presented

by

Albert Anthony Conti III

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

December 2006

Acknowledgement

I would like to thank my advisor Professor Miriam Leeser. Without the opportunities and guidance she has provided, none of the work presented in this thesis would have been possible. The patience, understanding, and technical advice she has given me have been invaluable in my research and personal growth.

I would also like to thank my colleagues in the Reconfigurable Computing Lab at Northeastern University. The friendly support they offer provide an enjoyable and productive work environment that I will miss.

I would like to extend my appreciation to Mercury Computer Systems for funding the research presented in this thesis as well as providing the hardware that made it possible. I would like to specifically recognize Sarah Leeper for her willingness to answer questions and help solve problems I encountered.

Abstract

Multi-computer platforms that incorporate FPGAs and other reconfigurable processors are emerging as powerful computing architectures capable of exploiting many levels of parallelism for a range of applications. The novelty of this technology, combined with the drastic differences between architectures, has resulted in a lack of tools for developing applications and maintaining portability across different platforms. This thesis presents a case study into the use of *VForce*, a framework that leverages VSIPL++ to deliver high performance for reconfigurable applications while maintaining portability across different multi-computer architectures. The case study is a hardware/software implementation of an adaptive time-domain beamformer. The computation required for adapting weights and reconstructing signals is split between software and FPGA hardware, which operate concurrently. Run-time performance is presented to show two orders of magnitude gain in performance over a software-only system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Heterogeneous cluster-style multi-computers that integrate FPGAs and other reconfigurable processing elements with microprocessors have recently emerged. While some systems serve as reconfigurable supercomputers dedicated to accelerating computationally prohibitive scientific algorithms, others exist as smaller embedded devices designed for real-time signal processing. Regardless of the form factor and architectural details that distinguish one platform from another, these systems provide opportunities for applications to exploit fine-grained and coarse-grained parallelism. These opportunities enable a level of performance that is not possible with software alone for many applications.

Reconfigurable computing architectures, though powerful, are complex to program and configure. Designing and developing an application that makes effective use of a reconfigurable multi-computer requires an understanding of the hardware as well as the computational requirements of the application. Due to the novelty of the technology and the drastic architectural differences between platforms, developing an application requires low-level programming and an in-depth knowledge of the re-

configurable devices. Compilers and other tools used to program and configure these systems are generally platform specific and lack support for application portability.

In this thesis, the *VForce* framework is presented. *VForce* is designed to maintain application portability across reconfigurable multi-computer platforms. The framework combines object oriented software that abstracts hardware details and maps functions to different platforms with a resource manager that dynamically binds these functions to available resources at run time. *VForce* sits on top of VSIPL++[22]. VSIPL++, the Vector Signal Image Processing Library, is an API to a list of commonly used signal and image processing functions. Implementations of VSIPL++ can be optimized for a specific platform to deliver high performance.

After introducing *VForce*, a case study is presented that demonstrates the effectiveness of using the framework for mapping functions of a coarser granularity than those native to the VSIPL++ specification to reconfigurable supercomputing platforms. For this case study, we chose to implement a time-domain adaptive beamformer. Beamforming is a spatial filtering operation used to reconstruct signals propagating in a direction of interest by combining signals received by an array of sensors. The type of beamformer we implemented is complex yet modular. In our implementation, computation is split between software and reconfigurable hardware to showcase *VForce*'s ability to facilitate concurrent processing. The beamformer was mapped to a Mercury Computer Systems reconfigurable platform. Run-time performance is presented to show two orders of magnitude performance gain over a software-only implementation.

The outline of the remainder of this thesis is as follows. Chapter 2 presents a background of hardware/software co-design and some of the challenges involved with integrating FPGAs and other special purpose processing elements into high performance applications. Chapter 2 also includes a background of the algorithms, applications and previous implementations of beamformers. Chapter 3 introduces *VForce*, the software library designed to maintain application portability across reconfigurable supercomputing platforms. Chapter 4 presents a case study that evaluates the effectiveness of using *VForce* for functions at a granularity higher than the functions native to the VSIPL++ specification. Chapter 5 presents the results and analysis of the run-time performance of the beamformer described in the previous chapter. Chapter 6 concludes and suggests potential future research directions.

Chapter 2

Background

The discussions that follow in this chapter and in the remainder of this thesis will be divided into two orthogonal but related topic threads. The first will address issues dealing with hardware/software co-design and making effective use of hardware other than microprocessors for computation. More specifically, this thread will focus on the challenge of designing applications which map algorithms to heterogeneous systems for high performance. The second thread will deal with beamforming as an algorithm and how it has been implemented in software, custom hardware and heterogeneous systems.

2.1 Hardware/Software Co-design

As has been the case since the inception of the modern computer, applications are limited by the functionality and speed of processors, as well as the size and speed of the memory available to them. Today, the increasing demand for processing power maintains a significant gap over what is currently available with state-of-the-art microprocessors and memory. Proof of this can be seen in the fact that billions of dollars

are spent each year toward improving chip fabrication processes and developing new processor architectures.

When an application requires more processing power than is available with state-of-the-art microprocessors, designers turn to alternatives. Some alternatives include Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), Application Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs), multi-core processors and hybrid architectures that combine microprocessors with other specialized computing resources. These specialized alternatives are each tailored for a subset of applications and are capable of outperforming microprocessors by one to three orders of magnitude. Any processing alternative to a microprocessor will be referred to as a special purpose processor (SPP) in the remainder of this thesis.

Making effective use of SPPs is a challenging problem. Architecturally, SPPs can be very different from microprocessors. These differences are why they are able to outperform microprocessors, which are designed for general purpose processing. Each SPP generally has its own set of design tools, programming paradigms and compatibilities. For this reason, there is little or no support for application portability across the different platforms.

Many issues need to be considered before moving an application to an SPP or a hybrid system, which incorporates a microprocessor and one or more SPPs. Knowledge of the underlying processing and memory access patterns in the application is required for making an educated decision as to what type of SPPs should be targeted.

It is often the case that a microprocessor is the best choice for a particular algorithm; however, algorithms can sometimes be redesigned to match a given architecture to deliver a gain in performance.

This section outlines the challenges facing application designers who employ SPPs as coprocessors that work in cooperation with a microprocessor. This configuration will be referred to as a hybrid system for the remainder of this thesis. Since the work presented in this thesis focuses on FPGA design, one of many available SPPs, the second part of this section will focus on the challenges in hardware/software co-design specific to FPGA technology. The following two sections will discuss the flows for the design and development of software and hardware for hybrid systems.

2.1.1 Software Development

Software compiled to run on a microprocessor that shares the computational workload of an application with an SPP coprocessor is generally much more complex than software compiled to run on a microprocessor that executes the entire application on its own. Software for a hybrid system needs to express the computation being executed on the microprocessor as well as the interface to the SPP. This interface usually includes considerations for synchronization, data movement and control. According to Amdahl's law[2], we can only reduce the run-time of an application to the length of time required for the slowest serial component of that application. In a hybrid system, this component is usually the software. Since high performance is the goal in using a hybrid system, it is important to make sure the software component

is designed to maximize the speedup possible with SPP coprocessors. This means that software for hybrid systems must be designed in a way that enables high SPP performance while limiting its serial contribution to the run-time of the application.

Applications implemented with a hybrid system, or entirely with an SPP, usually start as software source code written in a sequential programming language and compiled to target a microprocessor. Before the software can be redesigned to work with an SPP coprocessor, the performance of the application on a microprocessor needs to be evaluated in a way that exposes the opportunities for acceleration. Based on this, and possibly additional evaluations, decisions must be made on which partitions of the global application will be moved to SPPs, which SPPs will be targeted and how the algorithms need to be redesigned to best exploit the architectures of the different processing elements being used.

Challenges in the development of software for a hybrid system are closely related to the architectural details of the SPP and the environment in which the microprocessor and SPP are interconnected. It is often most efficient to program software at a low level in order to target the strenghts of a given SPP. However, in order to maintain portability, code should be written at a higher level of abstraction to target more than just a specific SPP. Therefore, there is a tradeoff between the performance possible with low level software design and application portability.

Hardware/Software Partitioning

The first step in developing the software component of a hybrid system is determining which parts of the application are in need of acceleration. Loops with high ranges are often the source of extended run-times on microprocessors. These blocks of code need to be responsible for a large percentage of the total run-time of the application in order to warrant moving to an SPP. Again, Amdahl's law dictates that the potential for gain in performance from accelerating a partition of an application is limited by the size of the partition with respect to the size of the overall application. For example, consider a software partition that has been identified to be compute-intensive and is accountable for 85% of the application's run-time. If this partition is to be re-implemented with an SPP, the run-time can be reduced by a maximum of 85% for a speedup factor of 6.67. Therefore, in addition to finding the candidate partitions for acceleration in a given application, a decision must also weigh the potential for gain in performance with the costs involved in using an SPP.

Identifying candidate partitions for acceleration begins by assuming that a designer will be able to achieve a speedup factor equal to the upper limit from Amdahl's law. In many instances, this is not possible. Determining what is possible requires an analysis of the application, the SPP, and how that SPP works with the microprocessor in the hybrid system. A designer choosing which SPP architecture to target needs to understand how partitions would be mapped to the different architectures, each architecture's proficiency in executing the operations required, the

communication schemes possible with the different architectures, and how the different processing elements are able to work together. Other factors must be considered as well, including the cost in terms of hardware, the necessary design tools and the design-time required.

Software Design

Software running on a microprocessor in a hybrid system needs to interface with the SPPs in order to control their operation, move data and synchronize processing. Usually, offloading portions of an application's computation to an SPP results in the addition of many lines of code.

The software component of a hybrid system differs from software written for a microprocessor alone in that the relative execution times of operations become an issue for the programmer. With a standalone microprocessor, the run-time of the application is equal to the sum of the run-times of each operation. In hybrid systems, pipelining and parallel processing techniques are often employed to overlap different stages of processing to improve performance. The run-time of a hybrid application is not a sum. Optimizing for performance requires additional analysis. It is important to make sure that operations are issued in a way that does not result in an unbalanced system. An unbalanced system is one in which one individual component is dominating the run-time of the system to the point it leaves one or more system resources underutilized. Avoiding an unbalanced system sometimes requires programming techniques like latency hiding and multi-buffering, which are not

usually employed when programming a single processor. To maximize performance, software has to synchronize the processing on the SPPs and provide access to data with enough throughput to keep up with execution on the SPP. At the same time, software is usually responsible for part of the overall computation. Designing software that maintains a balanced processing load and makes effective enough use of SPPs to warrant their costs requires in-depth knowledge of the application and platform architecture.

Software Portability

The previous section discussed the difficulties involved with exploiting the full capabilities of the available SPPs. For a given application and a given platform, the most efficient implementation will usually be the result of low-level programming and a system-level design specific to the target platform. This implementation rarely maps as efficiently to other platforms as it does to the one for which it was designed. Since performance is the primary goal in applications that make use of SPPs, effort is generally spent minimizing the run-time of the application rather than designing software that ensures portability across different platforms.

When portability is a priority, software design can be approached in two ways. The first is to encapsulate a low-level implementation for every SPP the software needs to support into one software program. This is essentially the same as designing many non-portable software implementations and choosing the correct one at compile-time. This method of designing for portability will ensure high performance

on multiple systems; however, this performance comes at the cost of a lengthy design time. A software design cycle is required for every SPP supported. Upgrades to hardware and functional changes require modification to each separate software implementation.

The second way to design software for hybrid systems and remain portable across platforms is to abstract the differences between platforms. Abstracting the platforms and interposing a middleware layer allows for the design of common interfaces for multiple SPPs. With common interfaces, a single software implementation can be designed to target multiple hybrid systems. Upgrades and changes to hardware will not affect individual application software as long as the interfaces remain unchanged. Application portability with this method of design comes at the cost of the overhead imposed by the middleware layer and whatever restrictions are created by forcing software to adhere to a fixed interface.

There will always be a tradeoff between the performance possible with low-level programming tailored to a given architecture and designing applications to be portable across multiple platforms.

2.1.2 FPGA Application Development

FPGAs are just one of many types of SPPs available to application designers seeking high performance alternatives to the microprocessor. FPGAs are generally an order of magnitude more expensive than their microprocessor counterparts in terms of chip cost and development platform cost. What makes FPGAs worth the additional

design time, money and expertise required are their fine-grained massively parallel processing architectures and their ability to be quickly reconfigured to perform different tasks.

Field Programmable Gate Arrays

FPGAs are integrated circuits that can be customized for a specific function or application within milliseconds. FPGAs are often chosen for signal and image processing applications because of their massively parallel nature. Many signal and image processing algorithms are inherently parallel and map to the parallel processing structure of FPGAs in an intuitive way. FPGAs consist of large arrays of logic blocks for data transformation and switches for routing and data movement. These arrays are configured by setting bits of configuration SRAM local to each block of logic. Current state-of-the-art FPGAs house tens of thousands of logic blocks on a chip. In addition to standard blocks of logic, FPGA vendors also include dedicated resources that vary from vendor to vendor and from chip to chip. Specialized hardware is available for adjusting clock frequencies, interfacing with high speed communication channels and reducing chip-to-chip skew. Also included are embedded block RAM, multipliers, DSP blocks and microprocessors.

FPGA Coprocessor Hardware

For high performance applications, FPGAs are typically mounted on accelerator boards. These boards serve as coprocessors that work in cooperation with a microprocessor in a hybrid system. Generally, peripheral components are mounted on

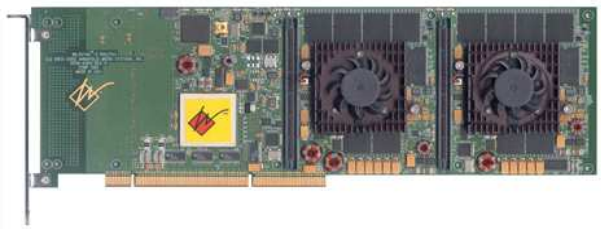

Figure 2.1: The WildstarII Pro is a PCI accelerator board housing two Xilinx VirtexII Pro FPGAs.[67]

accelerator boards with FPGAs. These peripherals often include banks of SRAM and/or DRAM, high speed serial ports and a controller for the communication bus that allows the microprocessor to control the functionality of the components on the accelerator board. A typical configuration is an FPGA mounted on a printed circuit board (PCB) with dedicated SRAM and DRAM that are accessed by means of a PCI or VME interface. One example of such a system is the WildstarII Pro[67] sold by Annapolis Microsystems. The WildstarII Pro, shown in figure 2.1, is a PCI board that houses two Xilinx VirtexII Pro[71] FPGAs and six banks of SRAM dedicated to each chip.

APIs are provided by manufacturers of accelerator boards so users can interface with the board and make use of the FPGA and other on-board devices through library calls made from software applications. For the research presented in this thesis, we target a multi-computer from Mercury Computer Systems that supports VME accelerator boards that house VirtexII Pro FPGAs.

Reconfigurable supercomputers are clusters of PCs modified to incorporate reconfigurable hardware. Offering reconfigurable components in cluster-style supercom-

puters is new to the supercomputing industry. Architectures are radically different from vendor to vendor and each continues to drastically evolve from generation to generation. While some systems still resemble cluster-style supercomputers, others have undergone a significant redesign to make the best use of the reconfigurable components available. In order to get the best performance possible, supercomputer vendors often use proprietary technology for connecting the commercial off-the-shelf processing elements. The Cray XD1[12] uses AMD's HyperTransport[25]. The SGI RASC[58] uses NUMAlink[50]. Mercury multi-computers like the one targeted in the research presented in this thesis use Race++[53].

FPGA Design Flow

This section describes the standard flow of design required for developing FPGA hardware. Once a circuit has been designed and programmed with a hardware description language, the source is compiled, simulated to test the expected run-time behavior and then run through the design tool chain in order to create a bitstream, which is used to configure the FPGA. High level compiler tools[6, 46] can help users to skip stages in the design flow; however, the process outlined here enables custom circuit design and is standard in industry.

The form of expression in code used to describe an FPGA circuit is much different than the high level source code software engineers are accustomed to using for targeting a microprocessor. The differences between the forms and levels of expression stem from the differences in the two architectures. Microprocessors have fixed

processing pipelines which implement a fixed set of functions. FPGA fabric can be configured in whichever way best matches the particular application. A hardware description language must be used to express the processing elements that will be formed within FPGA fabric, how those elements are going to be connected to one another and how the whole circuit will be controlled.

Circuits can be programmed at different and mixed levels of expression with hardware description languages. The higher level of expression is called behavioral modeling. In behavioral modeling, the programmer describes the behavior of a circuit. The compiler and other tools generate an underlying circuit that operates to produce the programmed behavior. The lower level of expression is called register transfer. When programming at the register transfer level, the user must describe the logic elements and how they connect to each other.

Describing a circuit is just one part of the programming stage of the design process. In addition, synchronization, control, memory management and communication have to be considered. Often, drivers for the off-chip peripherals such as SRAM, DRAM and communication channels are provided by the board vendor in the form of hardware description language source code or pre-built IP-blocks with which user logic can interface. Even with pre-built drivers, incorporating state machines to handle synchronization, data movement and communication can be difficult. These aspects of an FPGA application are often thought of as overhead, but for many applications, how these aspects are designed can be on the critical path of the performance of the entire system.

After compilation and successful behavioral simulation of a circuit, the first step in transforming the compiled description into an FPGA bitstream is synthesis. During the synthesis stage of the design cycle, the logic derived from the compilation of the HDL source code is transformed into an implementation of a circuit using blocks of logic and the other resources native to a particular FPGA technology. What makes synthesis complex is there are always many ways to implement a particular circuit with a given FPGA technology. The synthesis tool needs to deliver an implementation that is efficient in terms of speed and space. Most synthesis tools have the option to constrain the circuit the tool produces with a minimum clock frequency. Other options are sometimes available to adjust the levels of effort for optimizing the speed and area of the circuit. The result of the synthesis processes is a netlist. A netlist is a list of the FPGA structures used to implement the circuit, how those structures are configured and how they are connected with one another.

Once synthesis is complete, the structures used to implement the circuit as described in the netlist need to be mapped to the actual structures on a given target FPGA. Each structure might be able to be placed in many different positions on the chip. During the placement stage of the design process, it is the placement tool's responsibility to map each structure to a position on the FPGA. A routing tool is responsible for connecting the FPGA structures as placed by the placement tool. The placement of structures and the routing of the nets necessary to connect the structures is an iterative task. Once a circuit has been completely routed, it is checked to see if the timing constraints were met. If the circuit is too slow or skewed in a way

that would prevent correct operation, portions of the circuit need to be re-placed and re-routed. This process continues until a circuit that meets all timing constraints is discovered. Unfortunately, determining if there is a mapping of a given netlist to a particular FPGA that meets a certain timing constraint is an NP-Complete problem. Tools use heuristics and other approximation algorithms in an effort to find an efficient circuit.

Once a circuit that meets the necessary timing constraints is placed and routed, a bitstream, which encodes the status of each structure on the FPGA is generated and can be used to configure the chip.

For the design and development of the circuit presented in this thesis, we used Mentor Graphics ModelSim 6.0c[47] for behavioral simulation, Synplicity Synplify 8.0 Pro[60] for synthesis and Xilinx ISE 8.0[70] for place-and-route.

FPGA Application Design

FPGA application designers have a multitude of responsibilities. They need to determine which instructions they want to implement, how many instructions can execute in parallel, how many clock cycles each instruction should take to complete and where the data is going to reside before and after computation. In FPGA design, the time/space tradeoff is two-fold. Space is limited in terms of the amount of memory available to an FPGA and the amount of area on chip to place and route logic. FPGA application designers need to maximize performance with limited chip area, memory on-chip, memory off-chip, bandwidth to memory and communication bandwidth.

Efficient FPGA application design addresses the problem of how an FPGA can make the best use of its bandwidth to memory and communication to maximize the performance of the functions it has been allocated to execute. Maximizing the parallel processing capabilities for a given function on a given FPGA is not enough. The goal should be to maximize the data throughput given the parallel processing capability of the FPGA, the bandwidth to and from memory and the communication bandwidth.

FPGA applications usually begin as software source code that has been profiled to identify a partition responsible for a significant portion of the overall run-time. Generally, functions that are likely candidates for acceleration have processing and data access patterns that are uniform and without data and control dependencies. Determining the best way for a partition of code to be mapped to an FPGA and the platform that houses it depends on the processing and data access patterns as well as the possibilities for data movement on the given platform. The best opportunities for gains in performance often come as a result of redesigning the functions to operate in a streaming mode. Here, data is streamed on chip in an access pattern that has been predetermined, processed and then streamed off the FPGA. Designing hardware for a streaming mode of processing means placing enough logic in parallel to keep up with the rate of data moving on and off the FPGA. When a streaming mode of processing is not possible for a given application, input and intermediate data is usually stored off-chip in DRAM or SRAM. In these scenarios, FPGA hardware needs to be designed to control the off-chip banks of memory and process data at a rate

that makes the most efficient use of the bandwidth to each of the banks. On-chip memory can be used to buffer and cache data.

FPGA Application Portability

Designing an FPGA application to be portable in the same way that software is portable across microprocessors is not possible. Standards exist for hardware description languages (HDLs) so compilers and design tools are able to process the same source code. Even with the HDL standards in place, design tools provided by FPGA vendors and third parties use their own extensions to languages so programmers can make better use of the features unique to each FPGA. Even more problematic than the issues involved when portability from one FPGA vendor to another are the issues involved with moving an application from one accelerator board to another. Interfaces to off-chip peripherals will always be custom to the board for which they are designed. Since any application that makes use of an FPGA needs to interface with its pins at some level, facilitating portability means vendors would have to adhere to standard interfaces.

Individual components programmed with a HDL and free of vendor-specific language semantics are portable to the extent that they can be incorporated into applications that are built with different design tools and target different FPGAs on different accelerator board platforms. As soon as code is included that is specific to a given FPGA architecture or off-chip peripheral, the component can not be easily ported.

2.1.3 Summary

This section has outlined the challenges facing application designers who target hybrid systems that incorporate FPGAs. Developing software and FPGA hardware are two separate problems, each with their own forms of expression, programming paradigms and design tools. Designing efficient ways for software to work in cooperation with FPGA hardware is specific to the application. Portability across different hybrid system platforms is an even more difficult problem.

2.2 Beamforming

Beamforming is a term that encompasses a family of techniques and algorithms used for spatial filtering. The material in this section is taken from Van Veen's original presentation of the concept and techniques[63]. Beamforming is often useful for focusing arrays sensors on a signal or collection of signals of interest. Beamformers are advantageous because of their ability to null interference from propagating signals not of interest, as well as to increase the signal to noise ratio for signals of focus. Some algorithms also enable run-time adaptation so incoming data can improve signal quality. Algorithm parameters are adjusted to cope with changing environments and signal patterns.

2.2.1 Spatial filtering

A beamformer samples propagating wave fields in space. Signals received from sensors at different points in space are delayed and combined linearly. This linear combina-

tion produces an output equal to a weighted sum of signals that originated from a particular direction in space. The output at time k, $y(k)$, is given by a linear combination of the data received at the n sensors at times dependent upon the direction of the incoming wave of interest and the relative location of each sensor:

$$y(k) = \sum_{i=1}^{n} w_i * x_i(k - d_i) \tag{2.1}$$

where w_i is the weight applied to signal data received at sensor i, $x_i(k)$ is the signal received by sensor i at time k and d_i is the time it takes for the wave of interest to propagate from the center of the sensor array to sensor i in the direction of interest. Figure 2.2 shows how a propagating wave front affects an array of sensors. By delaying the signals received at each sensor, energy emitted by a point source at a specific moment in time can be recombined.

Equation 2.1 shows how a beamformer is used to reconstruct a signal originating from a specific direction by sampling at different points in space. One sample from each sensor is multiplied by the corresponding weight and summed to produce the output, $y(k)$ for a given step in time, k. This formula is effective for reconstructing signals with a specific frequency or a narrow band of frequencies. When a broad band of frequency signals are of interest, beamformers sample in both space and time. Since the beamformer system presented in this thesis samples in space only, the remainder of the background discussion will focus on purely spatially sampling beamformers. Figure 2.3 shows a signal processing diagram of a spatial filtering beamformer.

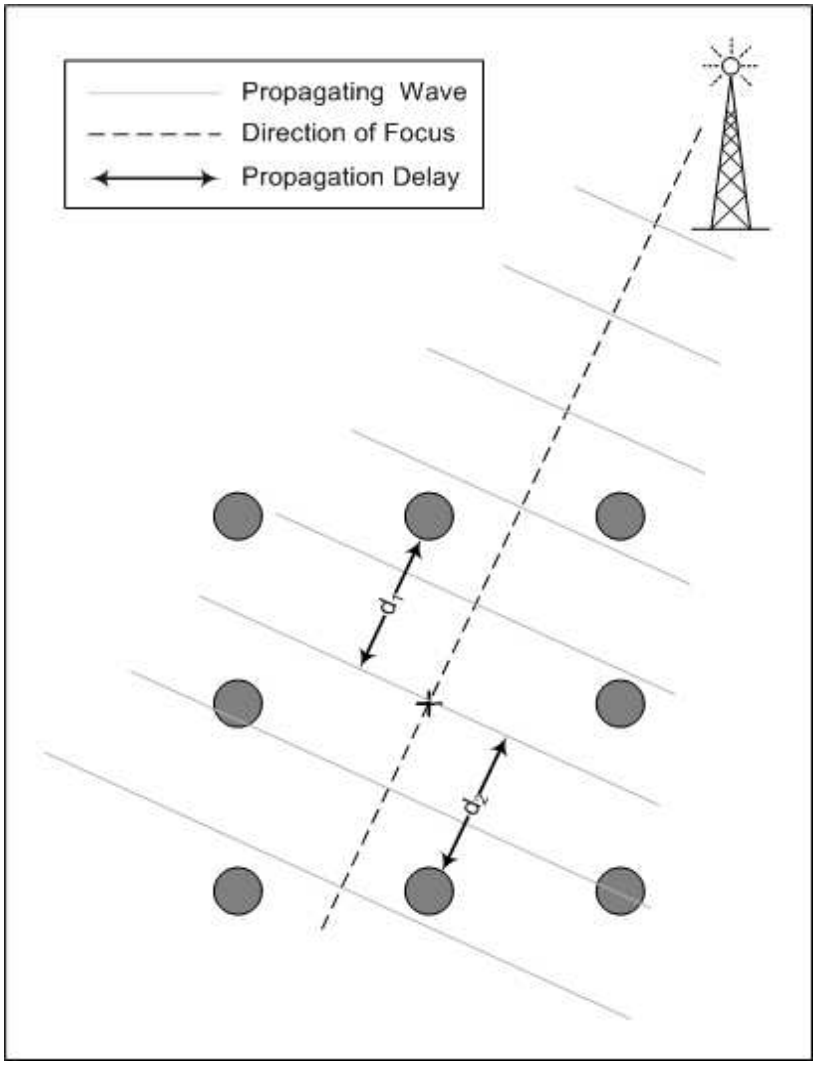

Figure 2.2: As waves propagate, signals cross each sensor in an array at different points in time. Delaying the signals received from each sensor will aim the array in a particular direction of interest.

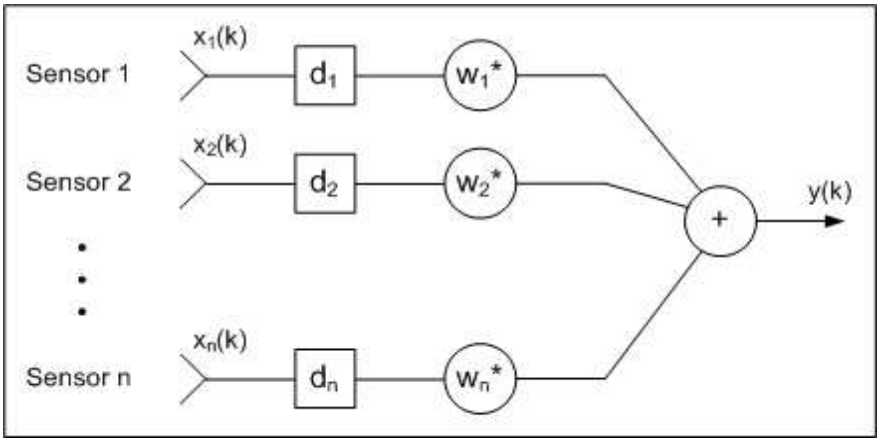

Figure 2.3: Spatial filtering beamformer. One sample from each sensor contributes to the output of the beamformer at each step in time.[63]

Both forms of beamforming can be performed in the time or frequency domain. The discussion throughout this thesis focuses on time domain beamforming that sample solely in space. Extending the discussion to examine how concepts would translate in the frequency domain and for beamformers that sample in time and space is straightforward.

2.2.2 Weights

The previous section described how beamformers are spatial filters. Adjusting the delay elements for each sensor will hone a sensor array in a particular direction of interest. After a sensor array is pointed in a desired direction, weights are chosen to form the desired beam pattern and null interference. Interference from jammers and other conflicting signals with similar frequencies will reduce the signal to noise ratio in the output of a beamformer. Weights correctly adjusted have the ability to cancel the effects of some interference and restore the ratio of the signal of interest

to noise and other forms of interference. Generating weights to form the desired beam pattern and cancel unwanted interference can be accomplished with a number of algorithms and methods discussed in books specific to beamforming and wireless communication[19, 33, 37]. For the beamformer presented in this thesis, weights are generated by solving a linear system of equations. A reference signal is used to find the initial set of weights. During operation, a feedback system is used to adapt the weights to changes in the environment.

2.2.3 Beamformer Categorization

Algorithmically, the process of forming a beam can be broken into two disjoint processes: beam formation and weight computation. Beam formation is the process of applying weights to sensor data and accumulating results specific to a certain direction in space. Weight computation is the process of updating the weights used for beam formation. Weights can be pre-computed and accessed as tables or generated on the fly from a number of algorithms. The degree to which the two processes, beam formation and weight computation, depend on one another is conditional upon the type of beamformer that is implemented.

A beamformer is data independent if the signals received by the sensors are not used to compute the weights required in beam formation. Data independent beamformers rely on sets of weights that have been previously generated. Statistically optimum beamformers adjust weights based on incoming sensor data. Within the category of statistically optimum beamformers, partially adaptive algorithms adjust

a subset of the weights at once and fully adaptive algorithms adjust all of the weights. Algorithms that update a subset of all the weights are often faster and less demanding computationally. Adaptive algorithms can either adjust continuously or periodically with block adaptation. The tradeoffs between a fast run-time and maintaining an accurate set of weights should be considered when deciding which methods to use for weight computation. Block updates and partial adaptation are efficient but may not deliver an accurate set of weights in noisy environments, whereas continuous updates and full adaptation will guarantee an optimal set of weights, but slow the rate at which a beamformer is able to process data.

2.2.4 Current Research

Beamformers are used for a multitude of applications in acoustic, optical, RF and other domains. Both new techniques and ways to use old ones are topics of current research[23, 34, 72, 8, 28, 65]. Perhaps the most common application of these techniques is for radar[15, 56] and sonar[9, 36] processing. Beamforming is also used in medical imaging applications[32, 68, 38] for tumor detection as well as in various speech recognition systems[48, 57, 18, 30] and for communication in wireless networks[7, 35, 55].

2.2.5 Summary

This section has presented a brief background on beamforming. Beamforming is a method of spatial filtering used to focus an array of sensors in a desired direction while canceling unwanted interference. Different types of beamformers can be implemented

with varying degrees of accuracy and run-time performance.

2.3 Related Work

This section summarizes the work related to the research presented in this thesis. Research into methods for hardware/software co-design is broad even when restricted to microprocessor-FPGA systems. Most techniques assign tasks to available processing elements at design time or at compile time. Our method is centered on run-time assignment of tasks to available processing elements. Thus, the discussion of related work specific to hardware/software co-design is confined to systems that use a scheme for run-time resource allocation. Beamformers implemented with reconfigurable hardware are outlined in addition to research pertaining to floating point and multiply-accumulate operations in FPGA hardware.

2.3.1 Hardware/Software Co-design

The increase in availability of architectures and systems that contain reconfigurable hardware has led to an increase in interest in run-time support for these architectures. These projects have the same goals as our approach, which are to maintain portability of application code and ease the task for programmers to make use of reconfigurable hardware. All these projects are aimed at a run-time execution model and not on compiling high level software to reconfigurable hardware, which is a related but distinct area of research.

Several researchers treat the hardware as a separate execution thread that runs

concurrently with software. Researchers at the University of Kansas have developed *hthreads* for specifying application threads running within a hybrid microprocessor-FPGA system[4, 3, 51]. Their system supports a master slave/model with one microprocessor tied to an FPGA. The support for hardware threads requires part of the system to run in hardware on the FPGA, and requires a fair amount of overhead. Elements of the operating system that handle context switching and semaphores are implemented on FPGA fabric. This reduces the time required for switching context from one thread to another and communicating from one thread to another. This comes at the cost of a distributed operating system and area on the FPGA that cannot be used for implementing a circuit to accelerate an execution thread.

A similar project[66] uses threads both in master/slave mode and in a more general network with FPGAs acting as processing elements. Their approach uses threads, and is based on an abstraction layer that uses a virtual memory model. A virtual memory handler must run in FPGA hardware to resolve accesses not in local memory. Similar to *hthreads*, this requires a fair amount of overhead. Like the *hthreads* system, operating system components are moved to FPGA fabric to create a distributed operating system, complicating the system and consuming FPGA resources. In the more general network approach, the hardware must include a communication agent that handles communication over the network as well as resolving memory accesses.

Researchers at the University of Florida have developed a system to provide runtime services for systems that include heterogeneous hardware. Their system consists of two parts, USURP (USURP's Standard for Unified Reconfigurable Platforms)[21]

and Carma (Comprehensible Approach to Reconfigurable Management Architecture)[13]. Their system supports general distributed systems where individual processors may have an attached reconfigurable hardware accelerator. USURP is built on top of MPI and is distributed, with a small manager running on every node. These researchers propose a standard interface for hardware designers to use at design time in order to support runtime portability and services. These services include performance monitoring and debugging. Their API is lower level than ours, and requires that users make calls to specify and download bitstreams, transfer data, etc. In our model, these operations are hidden inside functions and not exposed to the programmer.

Auto-Pipe[16] is a tool developed at Washington University in St. Louis that aids in the design, evaluation and implementation of pipelined applications distributed across a set of heterogeneous devices such as microprocessors and FPGAs. Auto-Pipe compiles high-level source code, partitions computation and maps components to different processors in a pipelined fashion. The tool suite is broken into stages and applications are designed and optimized in an iterative process. In the final design stage, pipelines can be adjusted in response to run-time performance; however, this tool focuses on binding functions to resources at design time.

In her PhD Dissertation[52], Heather Quinn presents Dynamo, a run-time system designed to bind image processing components to hardware for efficient run-time execution. Dynamo uses an array of optimization algorithms to solve the problem of assigning functions to different pipeline stages. Detailed latency and overhead estimations are used at run-time to evaluate the performance of pipelines before

they are built and mapped. If Dynamo determines that a hardware pipeline is the most efficient implementation of an algorithm given the compilation and synthesis overhead required to build that pipeline, a circuit will be synthesized at run-time and applied to the executing application.

Our approach differs from the above approaches in several important ways. First, our application code does not change at all from an all software implementation to a software/hardware implementation. Second, our approach does not require any support on the reconfigurable hardware itself. This makes our approach more flexible since it can make use of any vendor's API. The vendor can specify all the details of how the hardware is programmed. We do not change the way hardware is implemented, only the way it is invoked by software. Finally, our approach is lighter weight than other approaches, introducing minimal overhead.

2.3.2 Beamforming on FPGAs

The beamformer presented in this thesis is a hardware/software implementation that uses an FPGA for the delay-and-sum multiply-accumulate operations required for beam formation and a microprocessor for weight computation. The following is a discussion of issues pertaining to floating point and multiply-accumulate trends and methods for FPGAs and previous implementations of beamformers on FPGAs.

Floating Point

Underwood[61, 62] examined the trends of peak and sustainable single and double precision floating point performance on FPGAs and showed that the performance

gap between FPGAs and CPUs will likely continue to increase. This is due to the fact that the increases in FPGA clock speeds and transistor densities will be greater than CPU increases. Additionally, increases in the bitwidths of multipliers and other dedicated resources will contribute to the viability of fully floating point pipelines on FPGAs. Bitwidths will continue to be optimized and custom data formats will be used to increase throughput and performance. Still, for applications with single and double precision floating point requirements, advances in fabrication technology will make FPGAs a viable resource. As FPGA transistor densities increase, instantiating floats and doubles will require a smaller percentage of chip area. This will lead to FPGA designs with more data types native to microprocessors. Being able to use the same data structures in software and hardware will ease hardware/software design and integration.

When designing an FPGA application, designers often try to minimize the format for the data computed and stored. Minimizing the number of bits required for fixed and floating point data types will often maximize the throughput and performance of the application. The use of reduced floating point formats is the focus of an established body of research[5, 59, 14]. In most cases, the potential gain in performance by using a reduced format for data comes at the cost of reduced accuracy in results. Today, fabrication technology is at the point where data types native to microprocessors can be implemented with FPGA fabric while still delivering significant gains in performance. The beamformer presented in this thesis uses complex single precision floating point data at every stage of the processing pipeline in software and in FPGA

hardware. Results generated by the hybrid system presented in this thesis match software results exactly. This style of design requires no analysis of accuracy and eases the transition of data between hardware and software. One goal of the case study presented in chapter 4 is to evaluate what impact the precision we chose has on the run-time performance of the system.

Multiply-Accumulate

The beam formation process implemented with FPGA hardware in the hybrid system presented in this thesis is a parallel pipeline that executes consecutive multiply-accumulate operations. Multiply-accumulate is the process of computing the summation of the products of a set of pairs of values. This is a very common function for signal and image processing applications that has been implemented in FPGA hardware. Research[73, 29] from Professor Victor Prasanna at the University of South Carolina has focused on developing efficient methods for matrix-matrix and matrix-vector multiplication with FPGA fabric. At the core of these functions are multiply-accumulate operations.

Advanced methods for implementing the multiply-accumulate operations in FPGA fabric were not required for this research. Standard floating point IP cores available from Xilinx COREgenerator[69] were sufficient to make full use of the available memory and communication bandwidth on the platform targeted.

Beamformers

Reconfigurable hardware has been used to implement beamformers in the past. Hutchings et al.[24] implemented a fixed-point frequency domain beamformer on a SLAAC1b PCI board. The SLAAC1b PCI board consists of a Xilinx 4085, two Xilinx 40150 FPGA's, and 10 SRAMs. The system runs at a 50 MHz clock rate. The FPGA implementation was compared to those running on a variety of machines including Pentium-II and Pentium-III machines, HP PA-RISC workstations, and G4 Power PC's. The fastest performing machine was a 552 MHz PA-RISC workstation and its runtime was 18 times as long as the FPGA. The slowest machine was a 400 MHz Pentium-II machine with a runtime 83 times as long as the FPGA.

Leeper et al.[31] used an Annapolis WildstarII board with a Xilinx VirtexII FPGA to implement a fixed-point block-adaptive time-domain beamformer. The hardware was composed of multiply-accumulate and an update block for the weights used to form beams. The update block used Givens rotations for the least squares solver and QR decomposition. Weights are updates at a rate of once per 1000 time steps.

Graham et al.[17] designed a fixed-point data independent time-domain sonar beamformer for a hypothetical FPGA board. Altera outlined a method[1] that uses logic and the Nios soft-core processor available for the Stratix FPGA to realize either a block-adaptive or fully-adaptive beamformer in fixed point.

Parts of the beamformer presented in this thesis are similar to these previous implementations. Grahams's implementation may be the most similar in terms of the

FPGA implementation. The hardware presented in this thesis has no functionality for computing new weights or adapting to the result data as it is produced. The FPGA relies on software for adaptation and weight computation. Since Graham's system is data-independent, the two circuits are similar in behavior and design. The hybrid Altera design that uses a combination of hardware to form beams and software running on the Nios soft-core processor is similar to the design presented here in that computation is distributed between hardware and software.

What distinguishes the beamformer presented in this thesis from these and other beamformers implemented with reconfigurable hardware is its use of single precision floating point at every stage of computation, its generic design and its use of *VForce*.

2.4 Summary

This chapter has presented background for hardware/software co-design and for beamforming. It has also summarized related research specific to run-time execution models for hardware/software co-design and FPGA implementations of beamformers. In the next chapter, the *VForce* framework is presented.

Chapter 3

VForce

This chapter presents a framework that allows users to make use of FPGAs and other SPPs with source code written in C++. The library is called *VForce* and was developed in collaboration with other students and faculty of the Reconfigurable Computing Lab at Northeastern University[54]. The material in this chapter was taken from a previously published article[49]. *VForce* stands for **V**sipl++ **FO**r **R**econfigurable **C**omputing **E**nvironments. VSIPL++[22] is a standardized specification for a C++ API to a collection of commonly used signal processing functions. The *VForce* library is designed to allow C++ programs to target SPPs while remaining portable across different reconfigurable computing platforms. It does this while preserving the VSIPL++ API as well as hiding SPP-specific details from the end-user. Application code is still written in a serial fashion without any programming paradigms specific to *VForce*. The remainder of this chapter will present a brief background on VSIPL++ and then describe the *VForce* project.

3.1 VSIPL++ Background

VSIPL++ is an acronym for Vector Signal Image Processing Library. The '++' suffix is used to indicate that it is a C++ library and to differentiate it from VSIPL[64], which is VSIPL++'s predecessor specification in C. VSIPL++ and VSIPL are standards maintained by the High Performance Embedded Computing Software Initiative (HPEC-SI)[20]. This group involves a partnership of industry, academic and government organizations centered on promoting a unified computation/communication embedded software standard. The goal of the initiative is to enable "write-once, run-everywhere" development for applications of high performance embedded computing. The specifications for VSIPL and VSIPL++ are the results of the software initiative.

The ongoing development of VSIPL++ is focused on three major aspects of the specification: high performance, code portability and end-user productivity. VSIPL++ is an open standard thT facilitates both high performance and portability. To maximize performance on a given platform, a library can implement the specification in a way that best exploits the architecture of the platform. CodeSourcery[10], an industry member of HPEC-SI, offers a reference implementation of VSIPL++ based on its VSIPL reference implementation and the C standard library. The company also offers an optimized implementation that can make use of select performance math libraries such as Intel's Math Kernel Library[27] and Mercury Computer System's Scientific Algorithm Library[45]. The reference implementation was designed to be a reference point for the correctness of future implementations. The optimized imple-

```
#include <vsip/signal.hpp>
using namespace vsip;

int main()
{
    vsipl lib;

    Vector<cscalar_f> inData(16);
    Vector<cscalar_f> outData(16);

    Fft<const_Vector,cscalar_f,cscalar_f,fft_forward>
        ftObj(Domain<1>(16),1.0);

    outData = fftObj(inData);
}
```

Figure 3.1: Function objects interact intuitively with data objects with VSIPL++

mentation is an effort to extract high performance from a small subset of platforms that CodeSourcery has decided to target. Since high performance implementations are being developed, applications that use VSIPL++ need not use low-level code to achieve performance. The "openness" of the standard enables high performance through platform-specific implementations of the library, which in turn, enable the development of platform-independent, portable source code for applications.

The object-oriented nature of VSIPL++ provides an interface to data and functions that increases application development and end-user productivity. Objects for data storage and data access interact intuitively with objects that process and transform data. Figure 3.1 shows a code snippet that contains an example of a C++ program that uses VSIPL++ data and processing objects to execute an FFT.

Including an appropriate VSIPL++ header file is all that is required to access the

underlying VSIPL++ implementation. As shown, objects are instantiated to store input and output data. An `Fft` object is instantiated with template parameters that indicate the input and output data types and the direction of the transform. Constructor arguments are used to pass the FFT's size and shape, as well as a scaling factor, which adjusts how the transform is computed internally. Output is assigned the Vector returned by the function operator, overloaded by the VSIPL++ `Fft` class.

VSIPL++ appealed to our research lab for use in the *VForce* project for multiple reasons. Most importantly, VSIPL++ is centered on high performance and portability. High performance is always important to SPP application designers and portability is one of the major focuses of the *VForce* project. VSIPL++ also has built-in support for parallel processing and mapping data across distributed computing environments. The object-oriented nature of the specification provides for implementations with modular components. The VSIPL++ specification also includes a list of functions that are proven candidates for acceleration with FPGAs and other SPPs.

3.2 *VForce* Project Goals

The primary goal of the *VForce* project is to add support for SPPs to VSIPL++ in a way that maintains code portability. This means that new and existing VSIPL++ applications must be able to run on different platforms and with different SPPs. In order to make this possible, the VSIPL++ API has to remain unchanged. Additionally, the use of *VForce* can not introduce additional programming requirements. While

add-ons are acceptable, legacy VSIPL++ application code must run with *VForce* to target SPPs. Thus, the details specific to the underlying hardware, whether it consists of microprocessors or SPPs, needs to be hidden from the user. Likewise, errors and exceptions generated by hardware and low-level APIs need to be handled in a way such that users only see VSIPL++ exceptions and error messages. *VForce* also must ease the integration of new SPP architectures and functions, which will be mapped to the set of supported SPPs. Since high performance will always be a concern, overhead introduced by *VForce* must not limit the performance of applications. Finally, *VForce* must support concurrency so hardware and software functions can execute in parallel.

3.3 *VForce* Software Framework

The *VForce* software framework is a collection of C++ classes, which enables the execution of VSIPL++ functions in SPP hardware. There are two parts to the software framework. The first is a hierarchy of classes, which abstracts the functionality of SPPs. This hierarchy is designed to provide a common interface shared among the SPPs supported by *VForce*. The way in which each SPP implements the set of virtual methods that make up the common interface is different. However, the common interface allows classes that are responsible for mapping functions to SPPs to do so in a general way. This is essentially a middleware layer of software that separates the description of functions from the hardware they will execute on. For the remainder of this thesis, this layer will be referred to as the *VForce* middleware.

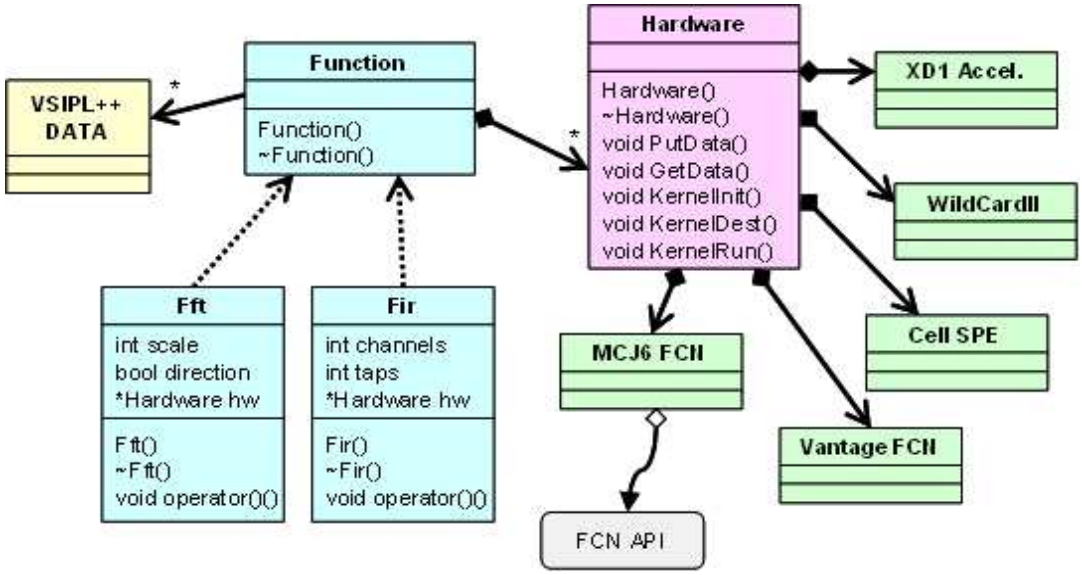

Figure 3.2: A UML diagram of the *VForce* software framework. The function classes in blue serve as the user interface to *VForce*. The class in pink serves as the middleware interface, which maps functions to SPP hardware. The green classes implement the middleware for each SPP.

The second part to the *VForce* software framework is the set of classes that replaces the existing VSIPL++ function classes. These classes are designed to intercept calls to supported functions and offload the required processing to SPP hardware. Figure 3.2 shows a UML diagram of the software framework. The classes in blue (Function, Fft and Fir) are the function class replacements. The pink class (`Hardware`) is the standard interface function classes use to map to hardware. The green classes (`MCJ6 FCN, Vantage FCN, Cell SPE, WildcardII, XD1 Accel.`) implement the middleware layer specified by the functions in the Hardware class.

3.3.1 Abstracting SPP Functionality

Each SPP supported by *VForce* is represented in the software framework by a class that implements the set of virtual methods that composes the abstract base class,

`Hardware`. Figure 3.2 shows the relationship between `HardwareBase` and the subclasses which implement the set of virtual methods. The list of functions in the `Hardware` class serves as the common interface that other classes use to access SPPs. Usually, an SPP is coupled with a vendor-specific API. This API is used to implement the interface. Essentially, these classes serve as wrappers, which insulate the details of each vendor's API. In Figure 3.2, the MCJ6 FCN class uses Mercury's FCN API to implement the interface described by `HardwareBase`. It is up to the person who creates a class to support a given SPP in a way that will maintain the goals of the *VForce* project. One of the goals is to mask underlying details specific to a given platform with the VSIPL++ interface. For example, special care must be taken to handle every exception and error a vendor's API may generate so that these errors are not exposed to end users. The list of virtual methods in the `Hardware` class includes calls to reset the device, load function kernels, send/receive data and control the operation of the function kernels. The list of methods was originally determined by the minimum requirement for microprocessor-FPGA co-processing. Since then, additional functions have been added to support higher performance run-time modes.

The list of methods used to abstract the SPP platforms limits what each SPP is capable of doing. SPP architectures can be radically different from one another. *VForce* attempts to support SPPs in a way that makes them easy to use. It is often the case that an SPP vendor's API supports higher performance methods which do not fit into the *VForce* software framework. In these cases, the run-time performance of an application designed with VSIPL++ and mapped to an SPP with *VForce* may

not be as efficient as a fully custom implementation of the application using the vendor API directly. For some applications, there will be a tradeoff between the performance possible with a fully custom software implementation and the ease of programming with *VForce*.

This common interface to SPPs is not the interface a user will see. This is an interface internal to *VForce* that function classes use to map VSIPL++ functions to SPP hardware. The hierarchy of classes can be thought of as a middleware that *VForce* interposes in between function classes and the API used to control each SPP. Users work within the context of VSIPL++. Applications target SPP hardware through regular VSIPL++ function calls.

3.3.2 Function Class Replacements

An implementation of the VSIPL++ specification will include classes corresponding to each of the functions available in VSIPL++. In order to map functions to SPP hardware, these classes need to know about the existence of SPPs. *VForce* interposes a class for each function supported on SPPs, which replaces the function class native to the implementation of the VSIPL++ specification. These replacements use the middleware described in the previous section. The original class does not go away. Instead, *VForce* uses naming conventions to intercept the instantiation of function objects. *VForce* function objects have the ability to map functions to hardware or use the original VSIPL++ function class to run in software. Figure 3.2 shows how the `Fft` and `Fir` function classes implement the interface described by class `FunctionBase`,

which interacts with VSIPL++ data objects and the *VForce* middleware.

VForce function classes are responsible for operating a generic SPP in a way that processes VSIPL++ data to return the same results the original VSIPL++ function class would have returned. Function classes see a generic SPP because all SPPs are accessed via a common interface. *VForce* function classes work by programming or configuring an SPP with a configuration stream capable of executing the function specified by the function class. There is a one-to-many mapping of function classes to configuration streams. A single function class is designed to work for any SPP that is supported by *VForce*. In order for *VForce* to support a function for a given SPP, there must be a configuration stream for the specific function that has been implemented for that SPP. The *VForce* framework does not create SPP function kernel implementations. The binaries and configuration streams used in cooperation with their function class counterparts must exist before the function can be incorporated into *VForce*. A separate body of research addresses the automated compilation of high level software source code to FPGA circuits. This work is beyond the scope of the research presented in this thesis; however, *VForce* is able to leverage implementations generated by automated compilers [6, 46, 26].

Figure 3.3 shows how the *VForce* library works in conjunction with user code, the VSIPL++ interface and the VSIPL++ implementation. The blue "*VForce* VSIPL++" box shows how the *VForce* interface supersedes VSIPL++ for a portion of the VSIPL++ specification as well as adds additional functionality. This will be explained in the next section. This blue area corresponds to the blue UML

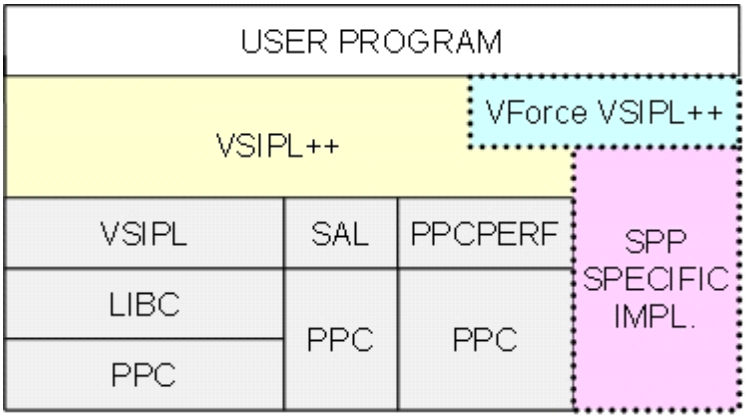

Figure 3.3: *VForce* (blue region) supersedes VSIPL++ for a subset of the VSIPL++ specification. The pink region coincides with the internal middleware and SPP implementations of VSIPL++ functions. The yellow region represents the VSIPL++ interface. The grey region is the VSIPL++ implementation.

class diagrams in Figure 3.2. Likewise, the pink "SPP Specific Implementation" box corresponds to the pink middleware class diagram in Figure 3.2. This diagram shows that *VForce* uses a combination of VSIPL++ and SPP specific libraries. The area in grey is an example of some of the libraries that can be used to implement the VSIPL++ specification.

3.3.3 Add-Ons

Offloading processing partitions of an application from software to SPP hardware can be accomplished at different levels of granularity in terms of the size and scope of the functions offloaded to an SPP. In some cases, the overhead involved with configuring a device, transmitting data and communicating with that device is too great to warrant offloading a function. Better performance often can come as a result of increasing the granularity of the function offloaded to an SPP. In some cases, this means increasing the amount of data processed on an SPP for a given function. In

other cases, it means moving a larger portion of the processing to an SPP.

A goal of the *VForce* project is to support new and existing VSIPL++ applications; however, the granularity of some VSIPL++ functions is at too low a level to improve performance by offloading to SPPs. This leaves opportunities for increases in run-time performance. *VForce* addresses this by going beyond the VSIPL++ specification to offer functions not available through VSIPL++. The case study presented in this thesis is an example of a function with a coarser level of granularity than other VSIPL++ functions. While existing VSIPL++ code will not benefit from the additional functionality that *VForce* enables, new applications can be written to take advantage of these functions.

3.4 Run-time System Model

VForce is set up so classes corresponding to each SPP's implementation of the generic SPP interface are compiled to shared objects. These shared objects are loaded dynamically at run-time to match the targeted SPP. This way, applications need not be recompiled to target new SPP architectures. Dynamically loaded shared objects (DLSOs) are added to a library that is representative of the available SPPs in a given system. Figure 3.4 shows how an application is broken down into VSIPL++, *VForce* and DLSO components. DLSOs implement the low-level functions required for the generic SPP interface specific to a particular SPP vendor API.

In addition to having a library of DLSOs to interface with the different SPPs available in a given system, *VForce* also requires that a library of function kernels

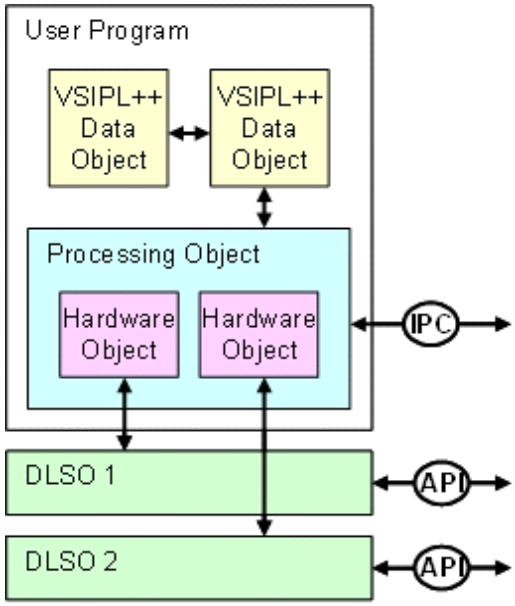

Figure 3.4: Function objects interact with VSIPL++ data and SPPs through hardware/DLSO middleware.

be present for each SPP. With systems composed of different SPPs, a mechanism for managing these libraries, as well as the physical hardware, is required. Figure 3.4 shows how function objects use inter-process communication (IPC) to interface with libraries of DLSOs and functions kernels.

3.4.1 Run-time Resource Management

VForce uses a run-time resource manager (RTRM) to supervise VSIPL++ applications and allocate hardware resources based on the availability of SPPs and their corresponding function kernels. The RTRM is a standalone application that runs on a separate processor or as a separate process on the same processor as a VSIPL++ application. There are three components that make up the RTRM. There is a library of DLSOs to support the available SPPs, a library of function kernels and a sched-

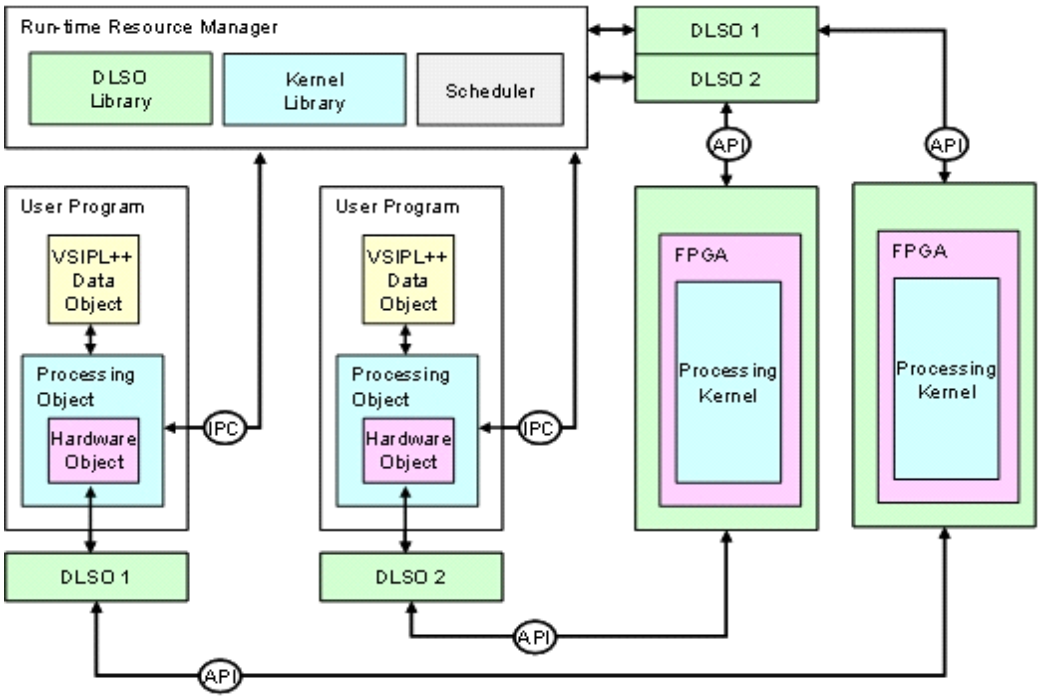

Figure 3.5: The RTRM supervises two independenet VSIPL++ applications. Each use a separate SPP allocated by the RTRM.

uler that handles requests for SPP resources from VSIPL++ applications. Figure 3.5 shows how the RTRM manages two independent VSIPL++ applications simultaneously. Note that the manager uses the same DLSOs that VSIPL++ applications use to interface with SPPs.

A VSIPL++ application that includes the *VForce* library needs to be able to communicate with the RTRM in order to use SPP resources. The RTRM is responsible for maintaining a list of available SPP resources and their status at run-time. *VForce* dynamically binds SPP resources to functions at run-time. If the execution of a VSIPL++ program reaches an instantiation of an object of a function class that has been replaced by the *VForce* software framework, the object instantiated will

be of a *VForce* class as opposed to the original VSIPL++ class. When execution reaches a call to a method of this object, the RTRM is invoked. First, the function object sends a request to the RTRM asking if there is SPP support for this function and if there is an available SPP capable of performing the function. If there is no support, or all the resources are busy, the RTRM will send a message back denying the request and the function will be executed in software. If there is support and an available resource, the RTRM uses a DLSO to configure a device with the appropriate function kernel and then sends a message back to the function object with a handle to the SPP. This SPP has been bound and allocated to the function object and will stay that way until the function object sends a message to the RTRM relinquishing the resource. Once the SPP resource has been acquired by the function object, it connects directly by loading the corresponding DLSO. At this point, there is no additional RTRM overhead. All communication and data movement is between the function object, which is local to the processor running the VSIPL++ application, and the SPP. Typically, the function object will iteratively upload input data, send messages to control SPP processing and then download result data.

The scheduling algorithm used by the RTRM is a simple first-come, first-served scheme. Requests are handled in the order they are received. Currently, there is no mechanism in place to choose the best SPP resource for a given function. The first SPP which is capable of performing the requested function is allocated. To reduce overhead, the RTRM keeps track of which function kernels were previously loaded for each SPP. In cases where the same function request is repeated, the RTRM needs only

configure the SPP the first time. This initial design of the RTRM is straightforward. In the future, more sophisticated scheduling can be incorporated.

3.5 Extending *VForce*

A goal of the *VForce* project is to facilitate the integration of new SPP architectures and the addition of new functions. Extending the *VForce* software framework to include new components is straightforward. To incorporate a new SPP, a new class must be written to implement the virtual methods in the `HardwareBase` class. This class is then compiled to a shared object and added to the library of DLSOs. In order to be useful, function kernels for this new SPP must be added to the library of function kernels. To support an additional VSIPL++ function with SPP hardware, a function kernel needs to be designed for one or many SPPs and a function class must be written to replace the function class native to the VSIPL++ implementation. To use the *VForce* framework on a new system, the RTRM must be ported to that system.

3.6 Summary

This chapter has presented *VForce*, a framework which enables users to target SPPs with serial C++ code. *VForce* sits on top of VSIPL++ and maintains application portability across reconfigurable computing platforms. *VForce*'s extensible software framework leverages existing SPP function kernels to deliver high performance in a way that hides hardware-specific details from users. With *VForce*, a run-time

resource manager supervises the execution of VSIPL++ applications and allocates SPP resources as needed. The next chapter presents a case study into the use of *VForce*.

Chapter 4

Beamforming: a case study using *VForce*

This chapter presents the design and development of a time-domain beamformer application that serves as a case study into the use of *VForce* for high performance computing. Beamforming is a function that is not included in the VSIPL++ specification. The focus of this case study is to assess the value of using *VForce* for functions at a coarser level of granularity than those native to VSIPL++. The beamformer is implemented to target a Mercury Computer Systems platform that incorporates microprocessors with FPGAs. The computation required for beamforming is split between software running on a microprocessor and FPGA hardware. The computation is split in order to showcase *VForce*'s ability to facilitate concurrent processing for hybrid applications. The case study provides a means of evaluating the effectiveness of *VForce* in terms of performance, application programmability and the methods used for implementing function kernels and the run-time resource manager.

This chapter begins by describing our approach for incorporating a new function with coarser granularity than those native to VSIPL++ and distributing computa-

tion between software and custom FPGA hardware. Next, the Mercury Computer System's reconfigurable computing platform is described and the development of both the software and FPGA hardware is presented. Finally, the experimental system used to evaluate the performance of the system at run-time is described.

4.1 Approach

The combination of many degrees of freedom in both hardware and software present a design space with many options for implementing a given algorithm. The *VForce* software framework is versatile in terms of the ways that function classes can be incorporated and implemented. FPGA fabric is able to implement essentially any type of digital circuit. Mercury's multi-computer architecture provides for many different processing configurations. There are a variety of algorithms that can be used for beamforming. Some of these algorithms can be adjusted so that a beamformer better fits the target platform and matches requirements for performance and accuracy. Therefore, when designing a system that maps the functionality of beamforming to the target Mercury platform through *VForce*, there are many options for processing, and in turn, many decisions that need to be made before we can develop a system.

At the highest level, the two major decisions that shaped our approach were which beamforming algorithm to use, and how the computations would be distributed across the Mercury multi-computer. Additional decisions having to do with interfacing were contingent upon the scheme for distributing the computation. We had to decide how the separate components would work concurrently, what the interface would look

like between the components and how user code would interface with the beamformer function. A system needed to be designed that was intuitive at the user programming level, while being capable of high performance at run-time. The following section will describe our approach in designing the software and FPGA circuits for adding beamformer functionality to *VForce*.

4.1.1 Algorithm Design

The process of forming a beam in order to focus a sensor array to receive a signal that propagates in a particular direction can be broken down into two separate sub-processes. First, weights are computed to form the desired beam pattern. Then, the weights are applied to incoming sensor data to reconstruct signals originating from a direction of interest. The goal of beamforming is to reconstruct the desired signal(s) and reduce the impact of noise and interference. Algorithms for beamforming are categorized based on how and when the weights are computed. A description of the types of beamformers and how they differ is available in Chapter 2 of this thesis. Regardless of which type of beamformer is used, each type allows for the two processes to be separated.

For this case study, we wanted the hybrid system to be capable of forming beams in different types of situations, in different domains (acoustic, RF, etc.) and for different sensor topologies. We did not use a specific sensor array or waveform as targets in the design of the system. There was no required minimum sampling frequency we needed to be able to process and no specified number of jammers that needed to be

handled at once. The goal was to design the beamformer to be generic, similar to other functions in VSIPL++. For this reason, we decided to design the system in a way that would allow a user to implement any type of time-domain beamformer. This means the user is responsible for writing the code that sets and/or updates the weights. In order to design, develop and test a functional beamformer, we had to decide on an algorithm for setting and/or updating weights. Adaptive beamformers are more computationally demanding than statistically independent beamformers. Since adaptive beamformers are slower on conventional computer systems and in higher need of run-time acceleration, we decided to implement an adaptive beamformer. The remainder of this thesis will focus on the design, development and performance of the adaptive beamforer we implemented; however, the design of the software and FPGA hardware added to support beamforming is generic and can be used for different types of beamformers.

There are many different types of adaptive beamformers. The most computationally demanding type of beamformer is the continuous, fully-adaptive beamformer. A continuous, fully-adaptive beamformer updates every weight for every sensor during each update cycle. Update cycles occur at a rate that ensures every sample from each sensor is combined with a corresponding reference signal to generate a new set of weights. On the other end of the spectrum, a block partially-adaptive beamformer updates a subset of the weights during each update cycle. Update cycles occur periodically at a rate based on characteristics of the signal of interest, the sensor array's environment and the application's tolerance for accuracy. Since

our implementation is not targeting a specific sensor array, sampling frequency or domain, we had the opportunity to base our adaptivity only on the tradeoff between accuracy of results/run-time performance. For the remainder of this thesis, the term, accuracy, will be used to express the frequency of weight update cycles when used to explain a beamformer system. A beamformer with a higher frequency of weight update cycles is more accurate than one that updates less frequently. We use a block fully-adaptive beamformer. In this scheme, all weights are updated for each update cycle, but update cycles take place periodically. If there were requirements for the run-time performance or accuracy of our system, we would have chosen a scheme that would have delivered the maximum performance given the accuracy required or delivered accuracy required given the minimum level of performance. Instead, we chose to implement a fully-adaptive beamformer in order to demonstrate the possible run-time performance for different levels of accuracy.

There are many algorithms and methods for adaptively choosing weights to form beampatterns. Evaluating the effectiveness of algorithms designed to select weights is beyond the scope of this thesis. What is important to this case study is the computation required to execute the algorithm for adapting weights is complex and intensive enough to warrant splitting the computation of the entire beamformer (adapting weights and forming beams) between hardware and software. We chose to use an algorithm that solves an overdetermined linear system of equations. Each equation represents a step in time. The products of the delayed samples from each sensor and their corresponding weights are summed to produce the result for that step in time.

Equation 4.1 shows an example of an overdetermined linear system of equations. n unique equations are required to solve for n variables in a linear system like this. When more than n unique equations are used to solve for n variables, there can be no exact solution. The solution we chose to use is the solution that minimizes the sum of the squared error for each equation.

$$\begin{aligned}
x_1(k-d_1)*w_1 + x_2(k-d_2)*w_2 + x_3(k-d_3)*w_3 &= y(k) \\
x_1((k+1)-d_1)*w_1 + x_2((k+1)-d_2)*w_2 + x_3((k+1)-d_3)*w_3 &= y(k+1) \\
x_1((k+2)-d_1)*w_1 + x_2((k+2)-d_2)*w_2 + x_3((k+2)-d_3)*w_3 &= y(k+2) \\
x_1((k+3)-d_1)*w_1 + x_2((k+3)-d_2)*w_2 + x_3((k+3)-d_3)*w_3 &= y(k+3)
\end{aligned} \tag{4.1}$$

Equation 4.1 represents the calculation used to produce the output of the beamformer. x_i represents time-indexed sample data from the ith sensor. When computing a set of weights, a number of equations greater than the number of weights are used and w is an unknown. To begin, a reference signal can be used for y to solve for the initial set of weights. This is accomplished by solving $xw = y$ for w. During operation, a reference signal can be transmitted periodically so that weights adapt to changes in the environment. It is also possible for the weights to adapt based on noisy incoming signals[63]. This is accomplished by computing a new set of weights equal to the current set plus a scaling factor of the difference between the current set and the estimated set. This estimated set of weights can be generated in many different ways. Random sets can be used. Van Veen[63] presents a method of multiplying

the noisy data matrix for data that has already been processed by the corresponding result data to get an estimated set. He shows how this method will guarantee convergence of a weight set if the scaling factor is chosen appropriately. To test our system, we had to choose a method of updating weights. To find our estimated set, we solve the system of linear equations discussed earlier with current input data and result data previously received. The length of time in between the current data and the data previously received is based on the frequency of the desired signal and the sampling frequency of the sensors. These parameters and the value for the scaling factor are inputs to the system. The number of equations used to solve for weights is also an input.

Deciding on a format for representing data is usually a critical decision in the design of any hybrid system. Using a reduced floating or fixed point format for data can result in a significantly smaller number of bits necessary for representing data. This increases the quantity of data stored in on-chip and on-board memory, as well as the rate at which data can be transferred via communication channels. The tradeoff, of course, is that for some algorithms, using less precision for representing data can have dramatic effects in the accuracy of the results. In some cases, the accuracy of results is not as critical as throughput, and the accuracy can be sacrificed. In other cases, the accuracy of results is the most important parameter in designing a system and reductions in the format for representation are not possible. In cases where throughput is most important but accuracy needs to be maintained, using a reduced representation for data requires a statistical analysis of the algorithm and

the data that will be processed. Typically this analysis will determine the minimum representation required to guarantee a maximum percentage error aloowed in results.

Maximizing throughput and the amount of sensor data which can be stored in memory at a time is a priority for this case study. However, producing repeatable results for an application is often important for run-time operation as well as for debugging. With *VForce*, functions within applications may run locally in software one moment then on an FPGA the next. *VForce* dynamically binds functions to resources at run-time so resources that return different results for the same function could potentially cause problems. Users with no understanding of the underlying hardware may be misled as to what is happening with their application code if results can not be guaranteed with *VForce*. In some cases, differences could be interpreted as bugs.

For this case study, single precision floating point data is used at every stage of processing in software as well as in hardware. This guarantees results are the same whether the function is being executed in software or with FPGA hardware. We chose to do this in order to evaluate what performance can be achieved without sacrificing accuracy or performing an analysis to determine a tolerable level of precision.

4.1.2 Hardware/Software Partitioning

The available hardware in the target system provides different configurations for how computation can be distributed between hardware and software. Decisions also had to be made as to how distributed components would work together and interface

with one another. Furthermore, the granularity at which the beamformer function is made available to the user is another degree of freedom in the design of this system. This section will present our approach in mapping the block-adaptive beamformer algorithm to the target system.

The type of beamformer and the algorithm for adapting weights were chosen in part because they allow the subprocess of adapting weights and the subprocess of forming beams to execute concurrently with limited synchronization. Our goal was to configure the hybrid system in a way that allows these two subprocesses to execute concurrently in order to make the most efficient use of the hardware resources available in the target platform. For the remainder of this thesis, the subprocess of adapting weights will be referred to as weight adaptation and the subprocess of applying those weights to form a beam and reconstruct a signal will be referred to as signal reconstruction.

Signal reconstruction is frequently limited in performance by memory. Sensor data is fetched, multiplied by weight data corresponding to a specific sensor and then accumulated. This subprocess requires random access into arrays of sensor and weight data, complex multiplication, accumulation and sequential access into an array to store results. Signal reconstruction has no data dependencies and can be fully pipelined. Data corresponding to each sensor can be processed in parallel. The computation is highly regular and requires floating point multipliers and adders. Due to the opportunities for pipelining and parallel processing, our approach was to map this component of the computation to FPGA hardware. FPGA fabric is capable

of supporting floating point math as well as the parallelism and pipelining possible for signal reconstruction. The target hardware also provides the FPGAs with high bandwidth to memory.

Weight adaptation can differ from application to application. For example, a beamformer operating in a noisy and changing environment will probably use a very short period of time between update cycles to maintain the signal to noise ratio. Another application may not need to update as often. Similarly, one application may need to use more equations to solve for a new set of weights while another can use a number of equations equal to one plus the number of sensors in the array. Due to the complexity of the subprocess, and that it will most likely vary significantly from application to application, our approach was to use software to execute weight adaptation.

By using an FPGA for signal reconstruction and software running on a microprocessor for the weight adaptation, the two processes have the opportunity to execute concurrently if the application permits. While some applications will allow undeterministic latency between the update cycles and the sensor data used to solve for the current weights, others will force the system to stall. Splitting the computation in this manner also provides a solution that requires limited inter-process communication. The signal reconstruction subprocess produces results based on sensor, weight, and parameter data that is indicative of the direction of the beam. Results can be produced at a rate equal the speed at which sensor data is received and processed. The weight adaptation subprocess produces weights based on sensor and result data.

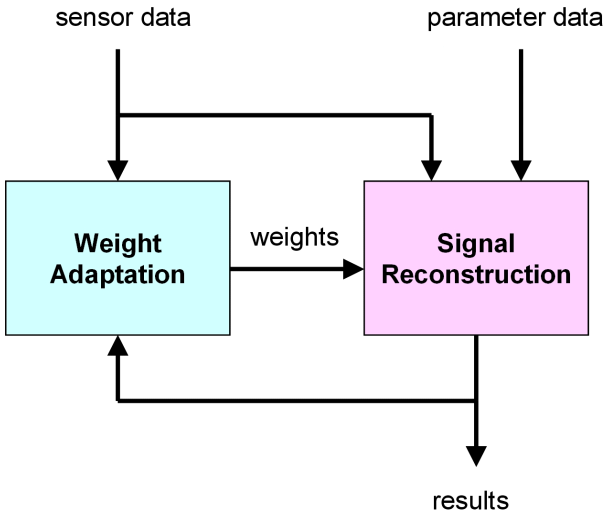

Figure 4.1: The weight adaptation and signal reconstruction subprocesses work together in a beamformer.

Thus, sensor data can be distributed to both subprocesses that can execute concurrently. Periodically, result data can be moved from the signal reconstruction subprocess to the weight adaptation subprocess in exchange for new weights. By distributing the computation at the subprocess level, the complexity, size and frequency of communication between components is minimized. Figure 4.1 shows the two subprocesses that make up the beamformer.

4.1.3 Software Interface

The level of granularity of the interface to the beamformer impacts the versatility of the function. Exposing the functionality at a low level makes the system more versatile, but less intuitive to program and integrate with VSIPL++ applications. Exposing the function at a higher level restricts how the system can be used. Our approach was to expose the functionality at the lowest level that hides the generic

SPP-specific implementation details. The signal reconstruction subprocess is encapsulated by a new function class. Member functions are provided to control the functionality of the underlying processing. The weight adaptation subprocess and the code that controls the movement of data between the two subprocesses is at the user level.

4.2 Target Platform

VForce is a framework designed to maintain application portability across different reconfigurable supercomputing platforms. The software added to the *VForce* software framework to support beamforming is platform independent. Only the function kernel that implements the signal reconstruction subprocess in hardware is platform specific. The Mercury multi-computer described in this section is the target platform in that the signal reconstruction function kernel has been implemented for the FPGA daughtercard from this system, and we use this system to benchmark the run-time performance for this case study.

The Mercury 6U VME[41] is a VME chassis in an embedded form factor with a backplane that supports Mercury's proprietary Race++[53] switch fabric. Race++ is an all-to-all switch fabric that operates at 66MHz. Point-to-point transfers are capable of reaching up to 266 MB/s for bursts. The 6U VME can be configured with any combination of PowerPC[40] and FPGA Compute Node (FCN) [39] daughtercards. Both daughtercards are compatible with the VME interface and support Race++. The PowerPC daughtercard houses two PowerPC 7447A microprocessors with Al-

Figure 4.2: The 6U VME chassis supports Mercury's Race++ backplane and accommodates ad-hoc configurations of PowerPC and FCN daughtercards.[42]

tiVec technology running at 800 MHz. Each microprocessor has access to memory and the Race++ switch fabric via its own interface chip. The MCJ6 FCN daughtercard houses two Xilinx Virtex II Pro 70 FPGAs. In addition, banks of SRAM and DRAM are dedicated to each FPGA. Figure 4.2 shows the 6U VME chassis. Figure 4.3 and figure 4.4 show the PowerPC and FCN daughtercards respectively.

Mercury provides an FCN Development Kit (FDK) for application development for their FPGA Compute Nodes. FDK extends Race++ into the FPGA fabric of the Virtex II Pros that are housed on the FCN daughtercards. FDK also provides VHDL IP so that user logic can interface with the memory and peripherals on the board. Figure 4.5 shows the interface FDK provides user logic. User logic is instantiated within User Application Space, while Mercury's intellectual property is implemented

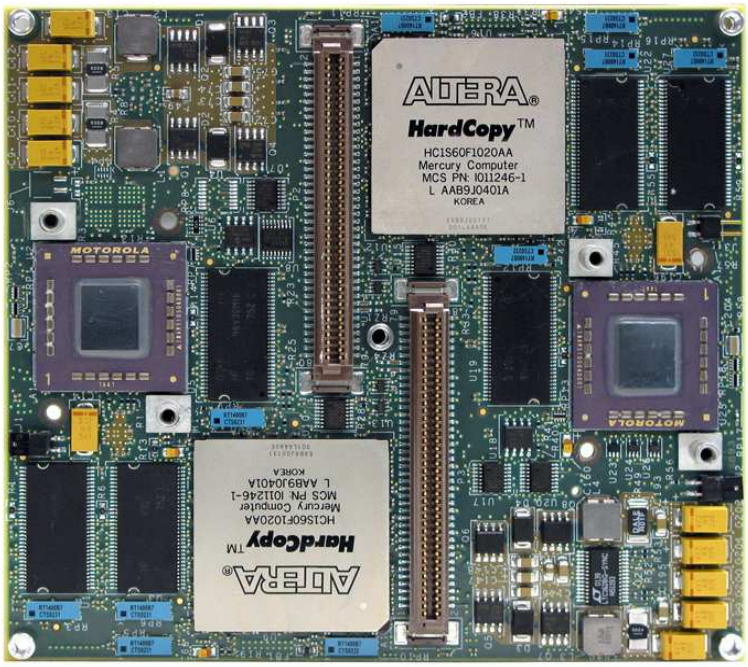

Figure 4.3: Mercury's PowerPC daughtercard houses dual PowerPC 7447As with AltiVec technology.[44]

Figure 4.4: The FCN module houses two FPGA compute nodes based on the Xilinx Virtex II Pro 70 FPGA.[43]

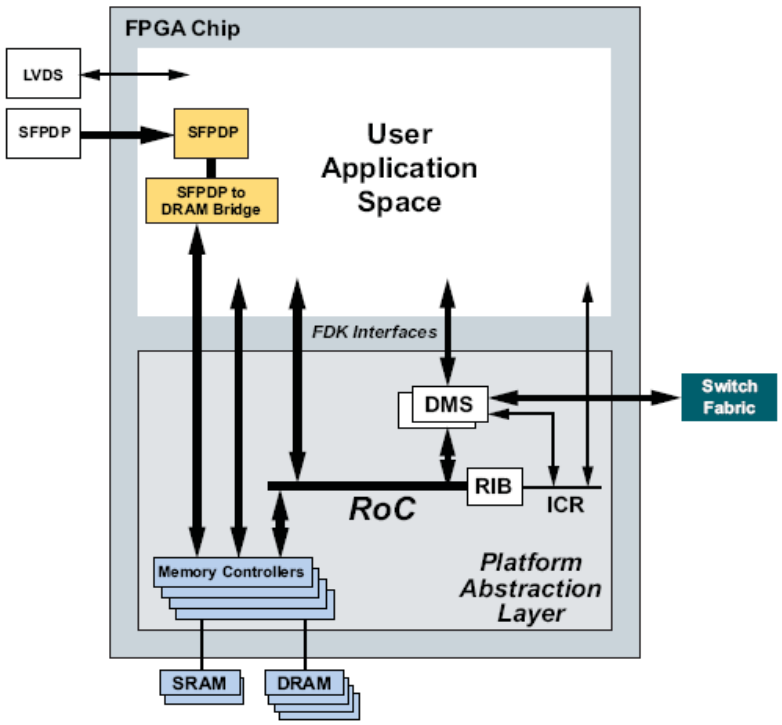

Figure 4.5: The FPGA Compute Node Development Kit provides an infrastructure that extends Race++ to RaceOnChip and provides user logic with an interface to FCN peripherals.[43]

within the Platform Abstraction Layer. The separation of application based logic and the infrastructure supplied by Mercury provides a degree of circuit design portability between Mercury platforms.

4.3 Beamformer Software

Two levels of software were added to the *VForce* framework in order to support beamforming on the Mercury platform. The lower level is the *VForce* middleware implemented in a generic fashion to map function classes to the FCN. The second layer is the function class for the signal reconstruction subprocess of the application. User level code was also developed outside of the *VForce* software framework to test

the system and to implement weight adaptation. This section will present the two levels of software added to the *VForce* framework and the user level code that implements weight adaptation. User code developed to test the system and benchmark the run-time performance of the hybrid system is presented in the final section of this chapter.

4.3.1 *VForce* Middleware

The middleware class for the Mercury FCN and the run-time resource manager are responsible for mapping function classes to the FCNs available in the 6U VME platform. Mercury provides an API for communicating between processes running on separate microprocessors. This API, which is part of Mercury's MCOE (Multi-Computer Operating Environment), is based on shared memory buffers and DX. DX is a high level, high speed, low latency communication pathway between microprocessors. Another API is provided in order for microprocessors to interface with the FCNs. The FCN middleware class added to the *VForce* software framework is a wrapper around the DX and FCN APIs. The virtual methods of the generic SPP interface in *VForce* map directly to function calls in the APIs provided by Mercury. DX function calls expect a pointer to a contiguous array of memory that has been allocated for DX transfers. Some methods, specifically the methods for moving data to and from memory local to FCNs, require data reorganization in order to transfer data from VSIPL++ types to the contiguous arrays which the DX function calls expect.

The FCN middleware class implements the *VForce* middleware layer for the tar-

get platform of this thesis. This class is essentially a wrapper for Mercury's FCN and MCOE APIs. The middleware methods function classes use to access hardware include `kernel_initialize`, `kernel_destroy`, `put_data`, `get_data`, `put_constant` and `get_constant`. `kernel_initialize` and `kernel_destroy` communicate with the RTRM to acquire and configure an FCN with a requested function kernel bitstream. These methods also create and destroy the shared memory buffers local to both the microprocessor and FCN that are used for communication. The `put_data` and `get_data` methods use the DX function calls in order to DMA data to and from the shared memory buffers established by `kernel_initialize`. The buffers local to the FCN consumes every word of SRAM and DRAM dedicated to an FPGA on an FCN, as well as the blockRAM that has a Mercury RaceOnChip interface. `put_constant` and `get_constant` methods are used for reading and writing small control words to and from the FCN.

The *VForce* run-time resource manager is a standalone application written in C++ that works in cooperation with the FCN middleware class to bind function classes to FCNs available in a Mercury 6U VME system. The manager is a lightweight application that handles two requests in a "first-come, first-served" order. The first request is `kernel_initialize`. When an FCN middleware object sends a `kernel_initialize` request to the manager, the manager receives the message, checks its internal listings of FCNs and function kernels to see if there is an available FCN and if there is a function kernel which matches the request. For this case study, the function kernel request is always an encoding that represents the signal

reconstruction function kernel. If an FCN is available and the function kernel exists, the manager configures the FCN with the bitstream that corresponds to that function kernel. Then the manager sends a message back to the FCN object that made the request that includes a handle indicating which FCN was configured. This FCN becomes locked and owned by the FCN object that requested it until that object releases the resource by sending a `kernel_destroy` request to the manager. To handle the `kernel_destroy` request, the manager returns the FCN which was allocated to the object to the list of available FCNs. If another `kernel_initialize` request is made for the same function kernel, the manager is able to immediately send a handle back to the requesting function object without reconfiguring the device. Overhead latency for acquiring and configuring an FPGA can take seconds of run-time when the bitstreams are remote. Any mechanisms used to reduce this latency have the potential to improve the overall performance of a system.

4.3.2 Beamformer Function Class

Supporting functions native to the VSIPL++ specification with *VForce* requires the addition of only one function class. This function class maps the computation to a special purpose processor via the *VForce* middleware or leverages the software-only function class native to the VSIPL++ implementation. The interface to the added class is required to match that of the function class in VSIPL++. Beamforming is a function not included in the VSIPL++ specification. To support beamforming, two function classes were added to the *VForce* software framework. One is an implemen-

tation of the signal reconstruction in software to act as the function class that would normally already exist within VSIPL++. The second is the hybrid version that encapsulates code that interfaces with the generic SPP hardware object and also has the ability to leverage the software-only signal reconstruction function class. As with functions native to VSIPL++, the interfaces to the two classes are identical. Figure 4.6 shows the interface as programmed in the class headers. Set and get methods are provided for the variables and pointers to arrays that store sensor, weight, parameter and result data. In addition to updating the pointers for the various arrays, the set methods update an internal list that keeps track of whether or not the arrays have been changed between hardware passes. The term *hardware pass* will be used to describe the computation, data transfers and synchronization involved in running a function in hardware through *VForce*. The `Start()` and `Finish()` control methods are provided so users can begin and end a hardware pass. The `Start()` method takes no arguments. The internal listings that keep track of the last time the data arrays were updated are used to determine which data need to be transmitted to the FCN in order to keep the FCN's local memory synchronized with the current VSIPL++ data.

During a `SignalReconstruct` hardware pass, the function kernel is reset, data is transferred to memory local to the FCN and then a control sequence tells the function kernel to start processing. When the function kernel is done processing, result data may or may not be transferred back to memory local to the microprocessor. The `Finish()` method takes a boolean argument that controls whether or not result data

is transferred back to the microprocessor's memory before another hardware pass is run. For hardware passes that are run immediately before a weight update cycle, the data is required to be transferred back to the microprocessor's memory. During periods when an update cycle is not pending, it is up to the application or the user to decide whether result data should be transferred or should remain local to the FCN.

4.4 Signal Reconstruction Hardware

The hardware portion of the hybrid system presented in this thesis is responsible for forming a beam to reconstruct a signal propagating in a particular direction. The computation required for forming a beam is straightforward. For each step in time, samples corresponding to a particular direction in space from each sensor are multiplied by their respective weights and accumulated. The access pattern for reading sensor data is pseudo-random. In other words, the access pattern is neither sequential nor completely random. Consecutive reads into the arrays that store the sample data for each sensor are guaranteed to be within a fixed number of addresses of each other. The fixed number of addresses is determined by the topology of the sensor array, the sampling frequency of the sensors and the speed at which the signal propagates through the medium of the sensor array. The process of combining the product of weights and delayed sample data provides opportunities for pipelining and parallelization. For each step in time, a single sample from each sensor array is used to reconstruct a propagating signal. These arrays can be accessed in parallel if they

```
class SignalReconstruct
{
  //constructors
  public:
    SignalReconstruct();
    ...
    ~SignalReconstruct();

  //set and get methods
  public:
    void SetSensorCount(int count);
   int GetSensorCount();
   void SetSampleCount(int count);
   int GetSampleCount();
   void SetWeightPointer(Vector< complex<float> > *pointer);
   Vector< complex<float> > * GetWeightPointer();
   void SetIndexPointer(Vector< int > *pointer);
   Vector< int > * GetIndexPointer();
   void SetSensorPointer(Matrix< complex<float> > *pointer);
   Matrix< complex<float> > * GetSensorPointer();
   void SetResultPointer(Vector< complex<float> > *pointer);
   Vector< complex<float> > * GetResultPointer();

  //control methods
  public:
    void Start();
    bool Finish(bool data);

  //internal data
  private:
  ...
}
```

Figure 4.6: The SignalReconstruct class header shows the interface shared between the software-only and the hybrid function classes.

are stored in separate memory banks, and parallel hardware can be used to combine data in a pipeline that allows new data to be processed continuously.

The hardware used for the signal reconstruction subprocess stores sensor array data in the four 2MB banks of SRAM dedicated to the FPGA, weight and parameter data in blockRAM on-chip and result data in one of the two DRAM banks. The pseudo-random access requirement for the sensor arrays prohibits a purely streaming circuit implementation. This means that sensor data needs to be buffered into one or more of the memory banks available on the FCN. The DRAM provides the largest amount of memory to the FPGA; however, SRAM is used because using DRAM would require large secondary caching memory and circuits dedicated for control in order to deal with the non-deterministic read latency, which is inherent to DRAM technology for random-access patterns. The four banks of SRAM are capable of delivering four individually addressable 32 bit data words per clock cycle. The size of blockRAM required to store the weight data for a sensor array with our circuit implementation is equal to 32 bits times the number of sensors in the array. The size of blockRAM required to store the parameter data for a sensor array is equal to 16 bits times the number of sensors in the array. In order to allow for changes in the topology of the sensor array during operation, an amount of blockRAM capable of supporting 1024 sensors is implemented so any number of sensors up to 1024 is supported with the circuit presented in this section. Since result data is stored in a sequential access pattern, there are no complications in using DRAM.

Mercury's FDK is used to connect the signal reconstruction IP we developed with

the memory on the FCN and the rest of the multi-computer via Race++. Mercury intellectual property modules are used to control and interface with the four banks of SRAM, one bank of DRAM and one DMA endpoint. These modules interface with RaceOnChip, which is the name of the infrastructure Mercury uses to extend the Race++ protocol onto the FPGA fabric. Dual ported blockRAMs were implemented to store weight and parameter data. One port is used to interface with the signal reconstruction module. The other port is used to interface with RaceOnChip. The four SRAMs, DRAM and the dual ported blockRAM modules implemented for this case study are all accessible via Race++ and VSIPL++ applications through *VForce*.

The signal reconstruction module is a floating point pipeline developed with VHDL that has been parameterized to make use of the available bandwidth to memory. Figure 4.7 shows the floating point pipeline and how it interfaces with the rest of the components on the chip. The IEEE single precision floating point adders and multipliers required for signal reconstruction were generated using version 1.0 of the floating point core generated by Xilinx COREgenerator[11]. Four multipliers and two adders make up each of the two complex multipliers responsible for applying the weight values to the sensor samples that flow out of the sensor arrays stored in the four SRAM banks. The pair of complex products are combined and accumulated to produce a value for each step in time. Not shown in Figure 4.7 is the control and addressing logic responsible for calculating the indices into the sensor arrays and managing the operation of the circuit.

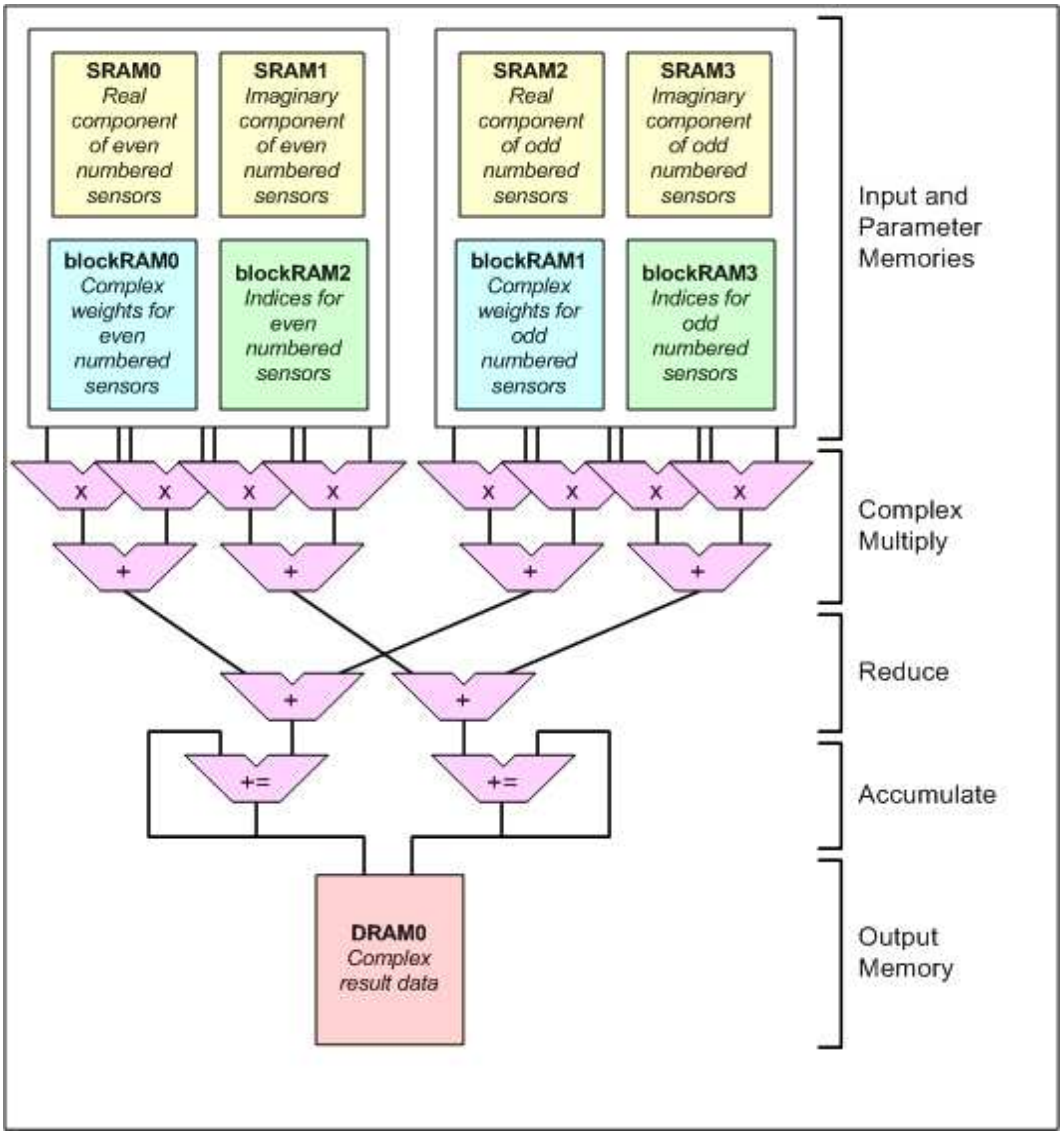

Figure 4.7: Single precision floating point multipliers and adders reconstruct signals by applying weight values to sensor data and accumulating with respect to each step in time.

The control and addressing logic consists of a lightweight state machine and a series of counters used to keep track of the circuit's progress and to generate the indices necessary for addressing the appropriate data samples in the sensor arrays. Sample data from each sensor array is partitioned into sub-sections equal to the size of the available SRAM divided by the number of sensors in the sensor array. A sub-section from each sensor array representing samples received for the same period in time is transferred to the SRAM for each hardware pass. This partitioning is accomplished in software with zero copy by using the Vector subview of the Matrix view from VSIPL++. Calculated strides are used so partitioning is accomplished within VSIPL++ data types. Partitions are interleaved in SRAM in a way that counters and a pipeline of fixed point adders can be used to determine the appropriate addresses for each step in time. For each step in time and for each sensor, the address into the SRAM housing the sensor arrays is equal to that sensor's partition's offset into SRAM plus the offset specific to the beam's direction plus the current step in time. Three pipeline stages are required to fetch the beam direction offset and add the three values in order to compute the address into SRAM.

4.5 Summary

Function classes and a function kernel for the Mercury 6U VME reconfigurable supercomputer are presented as new additions to the *VForce* software framework for supporting beamforming in the time domain. A block-adaptive beamformer is mapped to the target Mercury platform using *VForce* to distribute concurrent processing be-

tween software and FPGA hardware. In the next chapter, the run-time performance of the beamformer will be evaluated based on a testbench designed to explore the versatility of the beamformer function classes.

Chapter 5

Results

This chapter presents the run-time performance for the hybrid systems described in the previous chapter. Run-time performance is evaluated for a range of hybrid adaptive beamformer test applications and is compared to the software-only function class running on a single PowerPC microprocessor. An analysis of the performance is presented along with a discussion of the effectiveness and scalability of the hybrid system.

5.1 Experimental Setup

This section describes the hardware and software used to test the hybrid systems presented in chapter 4. In designing the test system, it was important to provide mechanisms to verify the correctness of the functions executing in software as well as in hardware for the different modes of processing. In addition to testing the beamformer for correctness, benchmarking was conducted to evaluate its run-time performance. Since the hybrid systems presented are designed to be generic and work for different domains, a single benchmark would not be an effective means to

evaluate the run-time performance. Therefore, a range of beamformer testbenches are implemented with the hybrid system.

5.1.1 Middleware Verification

FCN middleware objects make use of the MCOE APIs to communicate with the run-time resource manager and function kernels resident in FPGAs on FCNs. In order to test the functionality, middleware objects are tested with the run-time resource manager and a function kernel that included interfaces to all the peripherals made available on the FCN via the FCN middleware class. A function kernel was generated based on the Mercury Default Bitstream. This function kernel provided Race++ access to the peripheral memory banks, on-chip blockRAM and included a simple state machine to acknowledge receiving control signals from FCN middleware objects.

5.1.2 Beamformer Verification

The software-only function class written in VSIPL++ added to the *VForce* software framework was tested by comparing the results it generated against the results generated by a previously verified software implementation written in C. Both versions processed identical data sets generated randomly.

The hybrid function class and signal reconstruction function kernel were tested together with *VForce* and the run-time resource manager. A 6U VME setup with one PowerPC daughtercard and one FCN daughtercard were used for testing. One PowerPC executed the run-time resource manager while the other executed the program containing the testbenches. The results generated were verified by comparing

the results generated by the software-only function class. Both versions processed identical data sets.

5.1.3 Benchmark Suite

The software-only and hybrid function classes added to the *VForce* software framework are versatile in that they are capable of signal reconstruction for arrays of sensors of different sizes and topologies. The implementation of the block-adaptive beamformer discussed in this chapter allows for a variable number of equations used to solve for new weights during each update cycle. It also allows the user to choose when weight update cycles occur in terms of the number of sensor samples processed between updates.

Testbenches written as user-level VSIPL++ applications were designed in order to evaluate the run-time performance of block-adaptive beamformer implementations with a varying number of sensors, equations used per weight update and samples in between weight update cycles. The test application also varies the number of beams formed for a given buffer of sensor data. The last parameter the test application is capable of varying is the number of FCNs used for signal reconstruction. Since a 6U VME system can accommodate multiple FCNs, it may make sense in some applications to distribute sensor data across more than one FCN in a round-robin fashion.

5.2 Performance

This section presents the run-time performance of the following three systems for implementing beamforming: the software-only function class, the hybrid function class making use of one FCN and the hybrid function class making use of two FCNs. Each of these systems was evaluated by reconstructing signals based on synthetic data for different sensor array topologies, different parameters for weight adaptation and different sized bursts of processing. In total, 25 experiments were run on each system and benchmarked for performance. In addition to presenting the end-to-end run-times for each experiment on each system, a breakdown of the timing for each component of the overall processing for a sub-set of the experiments is presented.

5.2.1 Experiment Parameters

Four parameters of a simple test application were adjusted in order to create the 25 unique experiments run to evaluate the three systems. The parameters are: the number of sensors in the sensor array, the number of beams of focus, the number of equations for weight adaptations and the number of samples between weight update cycles. The range of values for each parameter was selected to represent the computational requirements of an assortment of beamformer applications, as well as to identify the bottlenecks in the three systems.

The first parameter used to define the performance experiments is the number of sensors in the sensor array. The signal reconstruction circuit is capable of processing sensor data in parallel and in a pipelined fashion. The larger a sensor array, the more

potential there is for gains in performance for the hybrid systems as compared to the software-only system. The number of sensors in each array ranged from 4 to 64 in the experiments presented in this chapter. The signal reconstruction circuit is capable of handling any even number of sensors up to 1024. Sensor arrays with an odd number of sensors are handled by padding an additional SRAM partition with zeroes. Sensor arrays with numbers of sensors greater than 1024 are split, reconstructed separately and then combined in software. Only arrays with even numbers of sensors were tested. The range only went as high as 64 due to limitations in the VSIPL++ least squares solver, which was used for weight adaptation. Additionally, the number of equations must be greater than or equal to the number of sensors in order for the least squares solver to work. 64 sensors with 64 equations is the largest size that would run with the optimized VSIPL++ implementation from CodeSourcery on the PowerPC daughtercard.

The second and third parameters used to define the performance experiments are the number of equations used for weight adaptation and the number of sensor samples processed between weight updates. These two parameters test the performance of the three systems for applications that use different schemes for adapting to noise and other environmental changes. The number of equations used for weight adaptation ranges from 64 to 1024. As mentioned above, there is a memory limitation in the VSIPL++ least squares solver. The size of the VSIPL++ QR object used for the least squares solver cannot exceed 64^2. Thus, for experiments with fewer sensors, more equations can be used while still not exceeding the memory limitation. The number

of samples processed between weight updates ranges from 16k to 256k. 16k was chosen as the minimum because at a period of 16k samples, the software component executing the weight adaptation subprocess accounts for a majority of the overall computation.

The last parameter used to define the performance experiments is the number of beams formed. Beamformer applications can be designed to reconstruct signals propagating in multiple directions and with different beam resolution. This parameter controls the number of beams used for signal reconstruction on a given set of sensor data. Since sensor data represents the largest arrays of data required to be transmitted to memory local to an FCN for a hardware pass, iterating over this data for several beams can provide opportunities for performance gains on the hybrid systems as compared to the software-only system. In our experiments, the number of beams processed ranges from 1 to 10,000.

5.2.2 Experiments

Twenty-five experiments were run on the three systems. Table 5.1 shows the values of each parameter for each experiment. Each experiment processes 1 megasamples per sensor. For the remainder of this chapter, experiments will be referred to by their experiment number.

5.2.3 Results

In this section, results for each of the performance experiments are presented for the three systems. Side-by-side comparisons are presented in the next section and are

Performance Experiments				
Exp.	**Sensors**	**Eqns.**	**Period**	**Beams**
1	4	1024	256k	1
2	4	1024	256k	10
3	4	1024	256k	100
4	4	1024	256k	1000
5	8	512	128k	1
6	8	512	128k	10
7	8	512	128k	100
8	8	512	128k	1000
9	16	256	64k	1
10	16	256	64k	10
11	16	256	64k	100
12	16	256	64k	1000
13	32	128	32k	1
14	32	128	32k	10
15	32	128	32k	100
16	32	128	32k	1000
17	64	64	16k	1
18	64	64	16k	10
19	64	64	16k	100
20	64	64	16k	1000
21	64	64	16k	10000
22	64	64	32k	10000
23	64	64	64k	10000
24	64	64	128k	10000
25	64	64	256k	10000

Table 5.1: 25 experiments were performed for the three systems.

accompanied by a detailed performance analysis. The run-times for each experiment are broken down into subintervals corresponding to the run-times of the components executing the two subprocesses and a subinterval for setup time. These subintervals are presented separately to show which components of each application dominate the overall computation. The first subinterval is the time required for setup. During setup time, memory is allocated, objects are constructed and data is reorganized and moved into the appropriate arrays. In table 5.2, software-only setup time is reported as zero for all 25 experiments. These run-times are actually non-zero, but less than one hundredth of a second. For the two hybrid systems, this interval represents all the time prior to the initial hardware pass. In tables 5.3 and 5.4, setup times are irregular for the two hybrid systems due to the non-deterministic interprocess communication latencies and the remote FPGA configuration. The other two intervals are for the time in the signal reconstruction subprocess and in the weight adaptation subprocess. One of the goals of this case study is to showcase *VForce*'s ability to facilitate concurrent processing. The total time interval is the sum of the time intervals for both subprocesses. The time when both subprocesses are executing concurrently is recorded as time in the weight adaptation subprocess. All run-times are presented in seconds (s). The results shown in red, are extrapolated as real results were not collected for these experiments due to the prohibitively long run-times. Since the two hybrid systems are two orders of magnitude faster than the software-only system, real run-time results were collected for all the experiments. Performance results for the software-only, single-FPGA and two-FPGA systems are

Software-only Performance				
Exp.	**Setup**	**Weights**	**Reconstruction**	**Total**
1	0.00s	0.06s	2.24s	2.30s
2	0.00s	0.55s	22.41s	22.96s
3	0.00s	5.53s	224.09s	229.63s
4	0.00s	55.98s	2240.90s	2296.88s
5	0.00s	0.12s	4.37s	4.49s
6	0.00s	1.16s	43.73s	44.88s
7	0.00s	11.58s	437.27s	448.85s
8	0.00s	115.80s	4372.68s	4488.48s
9	0.00s	0.25s	8.64s	8.88s
10	0.00s	2.47s	86.36s	88.83s
11	0.00s	24.69s	863.62s	888.31s
12	0.00s	246.88s	8636.19s	8883.07s
13	0.00s	0.56s	17.16s	17.73s
14	0.00s	5.64s	171.63s	177.27s
15	0.00s	56.37s	1716.30s	1772.66s
16	0.00s	563.72s	17162.90s	17726.60s
17	0.00s	1.42s	34.22s	35.64s
18	0.00s	14.21s	342.16s	356.36s
19	0.00s	142.06s	3421.55s	3563.62s
20	0.00s	1421.27s	34214.80s	35636.10s
21	0.00s	14206.40s	342155.00s	356362.00s
22	0.00s	7103.20s	342155.00s	349258.20s
23	0.00s	3551.60s	342155.00s	345706.60s
24	0.00s	1775.80s	342155.00s	343930.80s
25	0.00s	887.90s	342155.00s	343042.90s

Table 5.2: Run-time in seconds for each subprocess for each experiment on the software-only system. Times shown in red are extrapolated.

presented in tables 5.2, 5.3 and 5.4 respectively.

In addition to the 25 experiments run on the three systems, additional measurements were recorded in order to examine the run-time at a finer level of granularity than the three time intervals presented in tables 5.2, 5.3 and 5.4. These measurements break the time for a hardware pass for a single FCN down into four time intervals. A hardware pass is split between the time it takes to transfer sensor data to the FCN,

Single-FPGA Hybrid Performance				
Exp.	**Setup**	**Weights**	**Reconstruction**	**Total**
1	5.16s	0.06s	1.93s	1.98s
2	5.31s	0.56s	6.96s	7.52s
3	5.13s	5.59s	57.28s	62.86s
4	5.18s	56.13s	560.46s	616.59s
5	5.15s	0.12s	3.06s	3.17s
6	5.37s	1.16s	8.10s	9.25s
7	5.38s	11.56s	58.50s	70.06s
8	5.25s	115.63s	562.54s	678.17s
9	5.13s	0.25s	5.44s	5.69s
10	5.12s	2.46s	10.50s	12.96s
11	5.12s	24.58s	61.06s	85.63s
12	5.13s	245.73s	566.64s	812.37s
13	5.14s	0.56s	10.28s	10.84s
14	5.13s	5.62s	15.37s	20.99s
15	5.29s	56.12s	66.30s	122.42s
16	5.13s	561.17s	575.61s	1136.78s
17	5.13s	1.42s	19.99s	21.41s
18	5.28s	14.16s	25.16s	39.32s
19	5.14s	141.57s	76.85s	218.42s
20	5.13s	1414.97s	594.45s	2009.42s
21	5.14s	14228.40s	5690.82s	19919.30s
22	5.40s	7064.52s	2969.52s	10034.00s
23	5.30s	3543.17s	1547.62s	5090.78s
24	5.37s	1768.35s	850.85s	2619.20s
25	5.17s	884.50s	498.79s	1383.29s

Table 5.3: Run-time in seconds for each subprocess for each experiment on the hybrid-system making use of a single FPGA.

Two-FPGA Hybrid Performance				
Exp.	**Setup**	**Weights**	**Reconstruction**	**Total**
1	10.26s	0.06s	1.77s	1.83s
2	10.44s	0.56s	6.80s	7.36s
3	10.38s	5.56s	57.10s	62.67s
4	10.27s	55.62s	560.11s	615.73s
5	10.41s	0.12s	2.99s	3.10s
6	10.32s	1.15s	8.03s	9.18s
7	10.37s	11.51s	58.43s	69.94s
8	10.25s	115.11s	562.46s	677.57s
9	10.28s	0.25s	5.41s	5.66s
10	10.40s	2.45s	10.47s	12.92s
11	10.55s	24.45s	61.01s	85.46s
12	10.46s	244.49s	566.45s	810.94s
13	10.40s	0.56s	10.27s	10.83s
14	10.56s	5.59s	15.36s	20.95s
15	10.38s	55.89s	66.27s	122.16s
16	10.26s	558.84s	575.42s	1134.26s
17	10.40s	1.41s	19.99s	21.40s
18	10.38s	14.11s	25.15s	39.26s
19	10.24s	141.06s	76.81s	217.87s
20	10.43s	1410.34s	593.62s	2003.96s
21	10.27s	14130.60s	5779.12s	19909.07s
22	10.27s	7064.33s	2972.75s	10037.10s
23	10.27s	3542.98s	1551.22s	5094.20s
24	10.36s	1768.14s	854.54s	2622.68s
25	10.25s	884.35s	502.43s	1386.78s

Table 5.4: Run-time in seconds for each subprocess for each experiment on the hybrid-system making use of two FPGAs.

HW Pass Timing Breakdown	
8 MB sensor data transfer	0.30110s
768 B parameter data transfer	0.00032s
FCN processing	0.00002s
16 KB result data transfer	0.00875s

Table 5.5: Timing breakdown of a hardware pass for a single FCN for experiments 17 through 21.

to transfer parameter data and weights, to process data and to transfer result data from the FCN back to memory local to the VSIPL++ program. Table 5.5 shows the time (in seconds) for an 8 megabyte sensor data transfer, a 768 byte paramter data transfer, FCN processing and a 16 kilobyte result data transfer. Transfers of this size correspond to the functionality of experiments 17 through 21. For a single hardware pass, the time required to transfer sensor data dominates the run-time. The FCN processing time is insignificant with respect to the total time required for a hardware pass. When multiple beams are processed, sensor data is only transferred one time for all the beams.

5.3 Analysis

This section gives an analysis of the run-time performance results presented in the previous section. Table 5.6 shows the performance of the two hybrid systems compared to the software-only system. Run-time results for the software-only system are not shown. Instead, the run-times for each experiment and each interval for the software-only system are normalized to one and the run-times for the hybrid systems are presented as a factor that represents the number of times faster each system is as

Performance Comparison				
	Single-FPGA		Two-FPGA	
Exp.	Reconstruction	Total	Reconstruction	Total
1	1.16	1.16	1.26	1.26
2	3.22	3.06	3.29	3.12
3	3.91	3.65	3.92	3.66
4	4.00	3.73	4.00	3.73
5	1.43	1.42	1.46	1.45
6	5.40	4.85	5.45	4.89
7	4.47	6.41	7.48	6.42
8	7.77	6.62	7.77	6.62
9	1.59	1.56	1.59	1.57
10	8.23	6.86	8.25	6.88
11	14.14	10.37	14.15	10.39
12	15.24	10.93	15.25	10.95
13	1.67	1.63	1.67	1.64
14	11.17	8.45	11.17	8.46
15	25.89	14.48	25.90	14.51
16	29.82	15.59	29.83	15.63
17	1.71	1.66	1.71	1.67
18	13.60	9.06	13.60	9.08
19	44.52	16.32	44.55	16.36
20	57.56	17.73	57.64	17.78
21	60.12	17.89	59.21	17.90
22	115.22	34.81	115.10	34.80
23	221.08	67.91	220.57	67.86
24	402.13	131.31	400.40	131.14
25	685.97	247.99	681.00	247.37

Table 5.6: A performance comparison of the two hybrid systems tested. Values are factors of performance normalized to the software-only system.

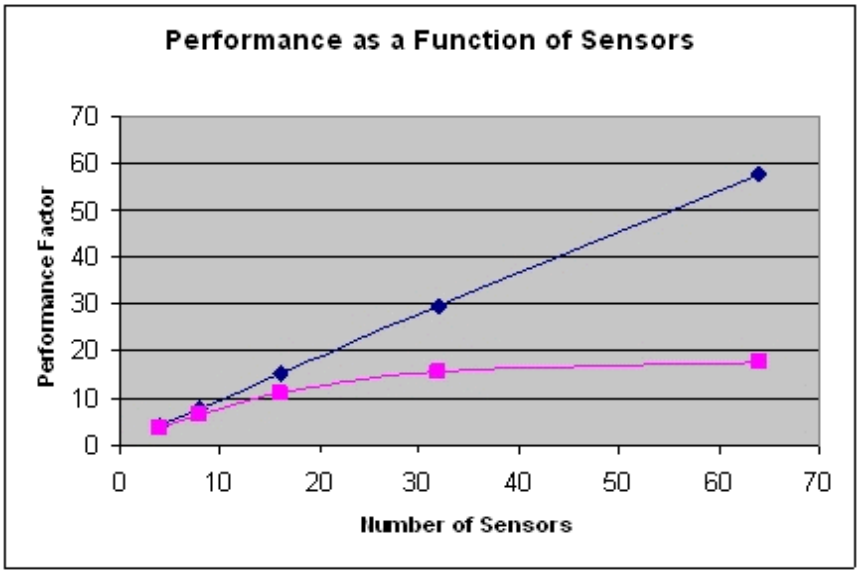

Figure 5.1: Normalized performance for experiments 4,8,12,16,20. Diamond data points represent signal reconstruction performance on a single FCN. Square data points represent the total performance.

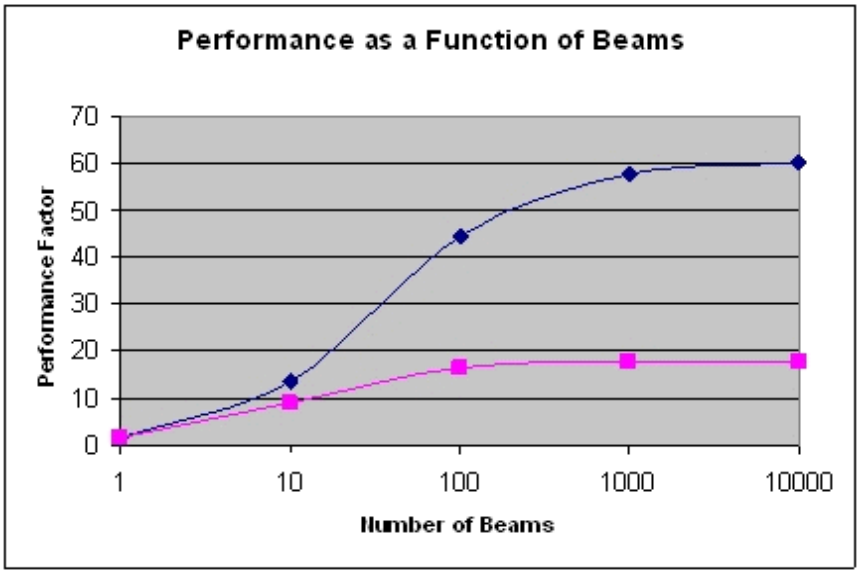

Figure 5.2: Normalized performance for experiments 17 through 21. Diamond data points represent signal reconstruction performance on a single FCN. Square data points represent the total performance.

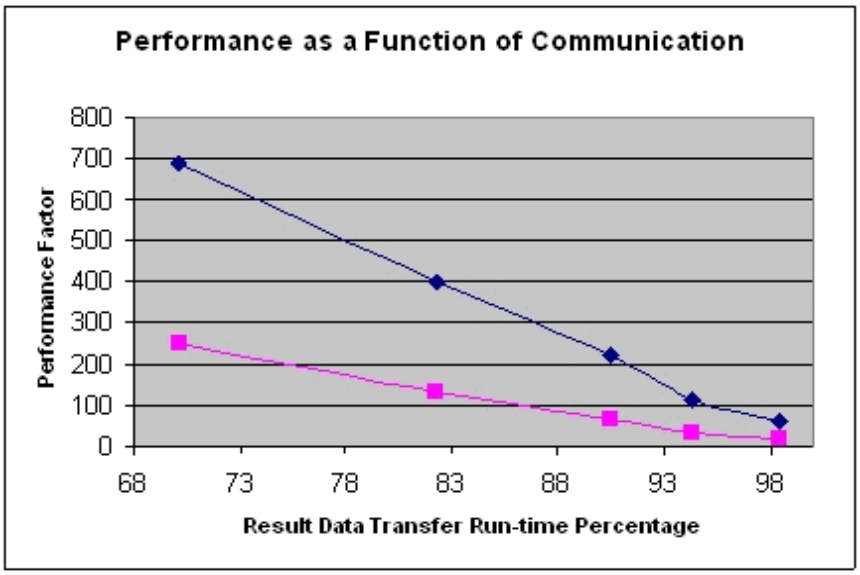

Figure 5.3: Normalized performance for experiments 21 through 25. Diamond data points represent signal reconstruction performance on a single FCN. Square data points represent the total performance. The horizonal axis represents the percentage of the signal reconstruction run-time spent transferring result data.

compared to the software-only implementation. For the remainder of this chapter, the term performance factor will be used to describe the relative performance gain of each system as compared to the software-only system for each experiment.

The hybrid systems presented in this case study are successful in delivering up to two orders of magnitude performance increase over the software-only system. In 25 different experiments designed to represent many beamformer applications, performance over the software-only implementation was improved.

The two columns immediately to the right of the experiment number in table 5.6 represent the gain in performance of the single-FPGA hybrid system as compared to the software-only system. Experiments with the same number of sensors are grouped and groups are separated by horizontal lines in the table. The performance of the

hybrid system as compared to the software-only system increases as the number of sensors in the array increases and as the number of beams increases.

Figure 5.1 shows the reconstruction and total performance factors as a function of the number of sensors for the single-FPGA system. The data points selected in order to illustrate this trend correspond to experiments 4, 8, 12, 16 and 20. These are the experiments that process 1000 beams for each size sensor array. The increase in the performance factor of the reconstruction interval of the overall computation grows close to linearly with respect to the number of sensors. This is due to the low percentage of the FCN's run-time of the overall hardware pass run-time. Table 5.5 shows that the run-time for FCN processing is orders of magnitude less than the other subintervals involved in performing a hardware pass. Doubling the number of sensors in an array doubles the amount of time required for the software-only system to reconstruct signals from the sensor data for each sensor. Since the time required for transferring result data dominates the reconstruction interval for experiments with large numbers of beams, and the amount of sensor data transferred per beam is relatively constant for experiments 1 through 21, the time required for the hybrid systems to perform reconstruction is relatively fixed. The total performance, which includes weight adaptation and signal reconstructions, does not increase linearly. It approaches a limit. This is due to the fact that the weight adaptation subprocess is executed in software and larger sized arrays require more time for weight adaptation. It takes close to double the amount of time for weight adaptation for a sensor array with double the number of sensors. The time interval for weight adaptation accounts

for more and more of the total run-time as the number of sensors in an array increases. This is the reason for the shape of the red curve in figure 5.1.

Figure 5.2 shows the performance of the single-FPGA system compared to the software-only system as a function of the number of beams processed. The data points selected in order to show this trend correspond to experiments 17 through 21. The only difference in the five experiments is the number of beams processed. The performance of the hybrid system as compared to the software-only system increases as a function of the number of beams processed but approaches a limit. As the performance approaches this limit, so does the percentage of the reconstruction time interval consumed by transferring result data. When the number of beams processed is great enough so that the time required for transferring results dominates the run-time of the reconstruction time subinterval, the performance factor for the reconstruction subinterval converges to a value. This value represents the ratio of the software-only run-time to the run-time for the result data transfers. The performance factor for the total run-time approaches a lower limit because the weight adaptation subprocess begins to account for more and more of the overall computation.

Figure 5.3 shows the factor of performance as a function of the percentage of the reconstruction time interval for which the result transfers are responsible. The data in this graph represent experiments 21 through 25. The only difference in these experiments is the period at which the weight adaptation subprocess occurs. A longer period translates to more infrequent updates and less result data being transferred.

The two rightmost columns in table 5.6 present the factors of performance gain for

the hybrid system that makes use of two FCNs. The goal of using more than one FCN was to increase the overall throughput of the system. Since concurrent processing is possible with *VForce*, distributing data to more computing resources provides an opportunity for additional performance improvements. For all the experiments conducted on the application presented in this case study, it was shown that using more than one FPGA is not worth the minimal gain in performance.

One reason why an additional FPGA does not improve the overall performance of the system is that only the processing, not communication, can be overlapped during concurrent processing through *VForce*. The version of *VForce* used for this case study did not support overlapping SPP processing with communication between the VSIPL++ program and the SPP. Another reason why the additional FPGA does not improve the overall performance is because of the relatively small amount of processing performed in a single hardware pass compared to the data transfer time. Weight and parameter data specific to a single beam are transferred and signals propagating in a single direction are reconstructed during each pass. Reconstructing signals propagating in multiple directions in a single hardware pass would provide an opportunity for additional performance gains and would extend the FCN's processing time. Increased processing time would provide more opportunity for concurrency in processing on multiple FPGAs.

5.4 Summary

This chapter presents the run-time performance results for 25 benchmarking experiments applied to the three systems presented in the previous chapter. The hybrid systems improve the run-time performance of user-level VSIPL++ application code by up to two orders of magnitude. In the next chapter, conclusions drawn from this case study are presented and future research directions are suggested.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The availability of FPGAs and other reconfigurable processing elements in cluster-style and other supercomputing platforms provides opportunities for high performance hybrid applications that make use of microprocessors and special purpose processors. These reconfigurable supercomputing platforms can exploit both fine-grained and coarse-grained parallelism in a range of domains and applications. Due to the recent emergence and rapid changes of this technology, architectures for reconfigurable computing platforms are drastically different from vendor to vendor. Even the architectures from a single vendor can undergo radical modifications from version to version. The tools and compilers used to program and configure these architectures are generally platform specific and there is little or no support for application portability. Porting an application to a new system often requires prohibitive redesign cycles.

VForce is a framework that supports application portability across reconfigurable supercomputing platforms. *VForce* leverages VSIPL++, which is an API to com-

monly used signal and image processing functions. VSIPL++ focuses on high performance, portability and end-user productivity. *VForce* makes use of an object oriented software framework that facilitates the integration of new functions and reconfigurable architectures as well as a run-time resource manager for binding functions to special purpose processors dynamically.

This thesis presents a case study on the use of *VForce* for functions at a higher granularity than the functions native to the VSIPL++ specification. An implementation of a hybrid adaptive time-domain beamformer is presented and mapped to a Mercury Computer Systems reconfigurable computing platform with *VForce*. A performance evaluation was conducted on a range of experiments. Through the performance evaluation, we show that *VForce* is effective. The hybrid systems presented achieve two orders of magnitude performance increase over the software-only system. The performance trends also indicate that functions of even coarser granularity may provide higher performance while remaining at a level low enough to be seamlessly integrated into the list of functions available with VSIPL++.

6.2 Future Work

One of the limitations the *VForce* framework imposed on the hybrid systems presented in this thesis was its inability to overlap FPGA processing with communication. Many applications could benefit from the opportunity to increase the number of concurrent operations. Additionally, most reconfigurable platforms can support concurrent communication operations. *VForce* should be designed to take advantage

of all the communication bandwidth possible. Many applications are like the beamformer in that there is a significant amount of communication and data movement necessary. For these applications, the inability to overlap processing with communication can result in systems that underutilize computing resources. Due to these reasons, the *VForce* software framework could benefit from the addition of a mechanism to overlap and issue concurrent communication operations.

The circuit for signal reconstruction added to *VForce* stores weight and parameter data for one beam at a time. For applications that reconstruct signals that propagate in more than one direction, it makes sense to store weight and parameter data for multiple beams at once. This would reduce the run-time required for context switching between beams in different directions. Since weight and parameter data is relatively small with respect to the rest of the data arrays required for beamforming and there is ample memory in blockRAM and DRAM available on the FCN, adjusting the circuit to handle multiple beams at once would not add complexity to the existing circuit. If this change was made to the signal reconstruction circuit, the hybrid function class could be altered to transfer fewer, larger blocks of data, which would make data transfer more efficient.

Currently, computation for the beamformer application is split between software and FPGA hardware. It is possible to implement the entire application with FPGA hardware. This would potentially improve performance, and possibly more importantly, increase the effective minimum period for weight updates.

Bibliography

[1] Altera. `http://www.altera.com/end-markets/wireless/advanced-dsp/beamforming/wir-beamforming.html`. Last accessed 28 August 2006.

[2] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Reston, VA, USA, April 1967. AFIPS Press.

[3] E. Anderson, J. Argon, W. Peck, J. Stevens, F. Baijot, E. Komp, D. Andrews, and R. Sass. Enabling a uniform programming model across the software/hardware boundary. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 2006.

[4] D. Andrews, R. Sass, E. Anderson, J. Argon, W. Peck, J. Stevens, F. Baijot, and E. Komp. The case for high level programming models for reconfigurable computing. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2006.

[5] P. Belanovic and M. Leeser. A library of parameterized floating-point modules and their use. In *International Conference on Field Programable Logic and Applications*, August 2002.

[6] Celoxica. `http://www.celoxica.com/products/dk/`. Last accessed 28 August 2006.

[7] J. C. Chen, L. Yip, J. Elson, H. Wang, D. Maniezzo, R. E. Hudson, K. Yao, and D. Estrin. Coherent acoustic array processing and localization on wireless sensor networks. *Proceedings of the IEEE*, 91(8), August 2003.

[8] S. Chen, N. N. Ahmad, and L. Hanzo. Adaptive minimum bit-error rate beamforming. *IEEE Transactions on Wireless Communications*, 4(2):341–348, March 2005.

[9] N. A. Cochrance, Y. Lee, and G. D. Melvin. Quantification of a multibeam sonar for fisheries assessment applications. *Journal of the Acoustical Society of America*, 114(2):745–758, August 2003.

[10] CodeSourcery. http://www.codesourcery.com/. Last accessed 30 October 2006.

[11] COREgen. http://www.xilinx.com/ipcenter/coregen/updates.htm. Last accessed 20 November 2006.

[12] Cray XD1. http://cray.com/products/xd1/index.html. Last accessed 29 November 2006.

[13] R. Deville, I. Troxel, and A. George. Performance monitoring for run-time management of reconfigurable devices. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2005.

[14] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based dsps. In *ACM International Symposium on Field Programable Gate Arrays, (Monterrey, CA)*, 2002.

[15] G. A. Fabrizio, A. B. Gershman, and M. D. Turley. Robust adaptive beamforming for hf surface wave over the horizon radar. *IEEE Transactions on Aerospace and Electronic Systems*, 40(2):510–525, April 2004.

[16] M. Franklin, J. Maschmeyer, E. Tyson, J. Buckley, and P. Crowley. Auto-pipe: a pipeline design and evalutation system. In *International Parallel and Distributed Processing Symposium (IPDPS06)*, 2006.

[17] P. Graham and B. E. Nelson. FPGA-based sonar processing. In *FPGA*, pages 201–208, 1998.

[18] J. E. Greenberg, J. G. Desloge, and P. M. Zurek. Evaluation of array-processing algorithms for a headband hearing aid. *Journal of the Acoustic Society of America*, 113(3):1646–1657, March 2003.

[19] S. Hayken and A. Steinhardt. *Adaptive radar detection and estimation*. Wiley, New York, 1992.

[20] High Performance Embedded Computing Software Initiative. http://www.hpec-si.org/. Last accessed 30 October 2006.

[21] B. Holland, J. Greco, I. Troxel, G. Barfield, V. Aggarwal, and A. George. Compile- and run-time services for distributed heterogeneous reconfigurable computing. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2006.

[22] HPEC-SI. VSIPL++: Vector Signal Image Processing Library. http://www.hpec-si.org, Last accessed 28 August 2006.

[23] H. M. G. Hussein, A.-B. M. Youssef, and Y. M. Kadah. Encoded multiple simultaneous aperture acquisition for improved signal-to-noise ratio in ultrasound imaging. In W. F. Walker and M. F. Insana, editors, *Medical Imaging 2003: Ultrasonic Imaging and Signal Processing. Edited by Walker, William F.; Insana, Michael F. Proceedings of the SPIE, Volume 5035, pp. 298-303 (2003).*, pages 298–303, May 2003.

[24] B. L. Hutchings and B. E. Nelson. GigaOp DSP on FPGA. *The Journal of VLSI Signal Processing*, 36(1):41–55, November 2004.

[25] HyperTransport. `http://www.hypertransport.org/`. Last accessed 29 November 2006.

[26] Impulse-C. `http://www.impulsec.com/XilinxDatasheetMarch06.pdf`. Last accessed 30 November 2006.

[27] Intel Math Kernel Library. `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm`. Last accessed 30 October 2006.

[28] S. A. Jafar and A. Goldsmith. Transmitter optimization and optimality of beamforming for multiple antenna systems. *IEEE Transactions on Wireless Communications*, 3(4):1165–1175, July 2004.

[29] J.-W. Jang, S. B. Choi, and V. K. Prasanna. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(11):1305–1319, 2005.

[30] D. Kolossa and R. Orglmeister. Nonlinear postprocessing for blind speech seperation. *Lecture Notes in Computer Science*, 3195:832–839, October 2004.

[31] S. Leeper, R. Haney, H. Nguyen, and M. Vai. FPGA beamforming in a wideband airborne radar system. In *2003 High Performance Embedded Computing*, September 2003.

[32] X. Li, S. K. Davis, S. Hagness, D. W. V. D. Weide, and B. D. Van Veen. Microwave imaging via space-time beamforming: experimental investigation of tumor detection in multilayer breast phantoms. *IEEE Transactions on Microwave Theory and Techniques*, 52(8), August 2004.

[33] J. Litva and T. K.-Y. Lo. *Digital beamforming in wireless communications.* Artech House, Boston, 1996.

[34] R. G. Lorenz and S. P. Boyd. Robust minimum variance beamforming. *IEEE Transactions on Signal Processing*, 53(5):1684–1696, May 2005.

[35] D. J. Love, R. W. H. Jr., and T. Strohmer. Grassmannian beamforming for multiple-input multiple-output wireless systems. *IEEE Transactions on Information Theory*, 49(10):2735–2747, October 2003.

[36] F. Lu, E. Milios, S. Stergiopoulo, and A. Dhanantwari. New towed-array shape-estimation scheme for real-time sonar systems. *IEEE Journal of Oceanic Engineering*, 28(3):552–563, July 2003.

[37] R. Mammone. *Computational methods of signal recovery and recognition*. Wiley, New York, 1992.

[38] J. A. Mann and W. F. Walker. A constrained adaptive beamformer for medical ultrasound: initial results. In *2002 IEEE Ultrasonics Symposium*, October 2002.

[39] MC FCN Card. `http://mc.com/literature/literature_files/MCJ6-FCN-ds.pdf`. Last accessed 20 November 2006.

[40] MC PPC Card. `http://mc.com/literature/literature_files/PPC7447A-dc-ds.pdf`. Last accessed 20 November 2006.

[41] Mercury 6U VME. `http://mc.com/products/view/index.cfm?id=10&type=systems`. Last accessed 20 November 2006.

[42] Mercury 6U VME. `http://mc.com/products/view/index.cfm?id=10&type=systems`. Last accessed 30 November 2006.

[43] Mercury FCN Daughtercard. `http://mc.com/literature/literature_files/MCJ6-FCN-ds.pdf`. Last accessed 30 November 2006.

[44] Mercury PPC Daughtercard. `http://mc.com/literature/literature_files/PPC7447A-dc-ds.pdf`. Last accessed 30 November 2006.

[45] Mercury Scientific Algorithm Library. `http://mc.com/products/view/index.cfm?id=5&type=software`. Last accessed 30 October 2006.

[46] Mitrion. `http://www.mitrion.com/products.shtml`. Last accessed 28 August 2006.

[47] ModelSim. `http://www.model.com/products/products_se.asp`. Last accessed 29 November 2006.

[48] D. C. Moore and I. A. McCowan. Microphone array speech recognition: experiments on overlapping speech in meetings. In *2003 IEEE International Conference on Acoustics, Speech and Signal Processing*, April 2003.

[49] N. Moore, A. Conti, L. Smith King, and M. Leeser. An extensible framework for application portability between reconfigurable supercomputing architectures. *To appear in Computer Magazine*, 2007.

[50] NUMAlink. http://www.sgi.com/pdfs/3771.pdf. Last accessed 29 November 2006.

[51] W. Peck, E. Anderson, J. Argon, J. Stevens, F. Baijot, E. Komp, D. Andrews, and R. Sass. Hthreads: a computational model for reconfigurable devices. In *Field Programmable Logic and Applications*, 2006.

[52] H. Quinn. *Runtime tools for hardware/software systems with reconfigurable hardware*. PhD thesis, Northeastern University, October 2004.

[53] Race++. http://mc.com/technologies/standards.cfm. Last accessed 20 November 2006.

[54] Reconfigurable Computing Lab at Northeastern University. http://www.ece.neu.edu/groups/rcl/index.html. Last accessed 30 November 2006.

[55] W. Rhee, W. Yu, and J. M. Cioffi. The optimality of beamforming in uplink multiuser wireless systems. *IEEE Transactions on Wireless Communication*, 3(11):86–96, January 2004.

[56] K. Schuler, M. Younis, R. Lenz, and W. Wiesbeck. Array design for automotive digital beamforming radar system. In *2005 IEEE International Radar Conference*, May 2005.

[57] M. L. Seltzer, B. Raj, and R. M. Stern. Liklelihood-maximizing beamforming for robust hands-free speech recognition. *IEEE Transactions on Speech and Audio Processing*, 12(5):489–498, September 2004.

[58] SGI RASC. http://www.sgi.com/products/rasc/. Last accessed 29 November 2006.

[59] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, 1995.

[60] Synplify Pro. http://www.synplicity.com/products/synplifypro/index.html. Last accessed 24 November 2006.

[61] K. Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM Press.

[62] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point blas performance. *IEEE Symposium on FPGAs for Custom Computing Machines*, 00:219–228, 2004.

[63] B. D. Van Veen and K. M. Buckley. Beamforming: a versatile approach to spatial filtering. *IEEE ASSP Magazine*, 5(2):4–24, April 1988.

[64] Vector Signal Image Processing Library. `http://www.vsipl.org`. Last accessed 28 August 2006.

[65] S. A. Vorobyov, A. B. Gershman, and L. Zhi-Quan. Robust adaptive beamforming using worst-case performance optimization: a solution to the signal mismatch problem. *IEEE Transactions on Signal Processing*, 51(2):313–324, February 2003.

[66] M. Vuletic, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers*, 22(2):102–113, 2005.

[67] WildstarII. `http://www.annapmicro.com/wsiippci.html`. Last accessed 29 November 2006.

[68] Y. Xie, B. Guo, J. Li, and P. Stocia. Novel multistatic adaptive microwave imaging methods for early breast cancer detection. *EURASIP Journal on Applied Signal Processing*, pages 1–13, 2006.

[69] Xilinx. `http://www.xilinx.com`. Last accessed 8 October 2006.

[70] Xilinx ISE. `http://www.xilinx.com/ise/logic_design_prod/foundation.htm`. Last accessed 24 November 2006.

[71] Xilinx VirtexII Pro. `http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/index.htm`. Last accessed 29 November 2006.

[72] T. Yoo and A. Goldsmith. Optimality of zero-forcing beamforming with multiuser diversity. In *2005 IEEE International Conference on Communications*, May 2005.

[73] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, New York, NY, USA, 2005. ACM Press.