RECONFIGURABLE COMPUTING WITH RapidIO FOR SPACE-BASED RADAR

By

CHRIS CONGER

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2007

To my favorite undergraduate professor, Dr. Fred O. Simons, Jr.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Alan George, for his patience, guidance, and unwavering support through my trials as a graduate student. I thank Honeywell for their technical guidance and sponsorship, as well as Xilinx for their generous donation of hardware and IP cores that made my research possible. Finally, I want to express my immeasurable gratitude towards my parents for giving me the proper motivation, wisdom, and support I needed to reach this milestone. I can only hope to repay all of the kindness and love I have been given and which has enabled me to succeed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

RECONFIGURABLE COMPUTING WITH RapidIO FOR SPACE-BASED RADAR

By

Chris Conger

August 2007

Chair: Alan D. George
Major: Electrical and Computer Engineering

Increasingly powerful radiation-hardened Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and conventional processors (along with high-performance embedded system interconnect technologies) are helping to enable the on-board processing of real-time, high-resolution radar data on satellites and other space platforms. With current processing systems for satellites based on conventional processors and non-scalable bus interconnects, it is impossible to keep up with the high throughput demands of space-based radar applications. The large datasets and real-time nature of most space-based radar algorithms call for highly-efficient data and control networks, both locally within each processing node as well as across the system interconnect. The customized architecture of FPGAs and ASICs allows for unique features, enhancements, and communication options to support such applications. Using a ground moving target indicator application as a case study, my research investigates low-level architectural design considerations on a custom-built testbed of multiple FPGAs connected over RapidIO. This work presents experimentally gathered results to provide insight into the relationship between the strenuous application demands and the underlying system architecture, as well as the practicality of using reconfigurable components for these challenging high-performance embedded computing applications.

CHAPTER 1
INTRODUCTION

Embedded processing systems operating in the harsh environments of space are subject to more stringent design constraints when compared to those imposed upon terrestrial embedded systems. Redundancy, radiation hardening, and strict power requirements are among the leading challenges presented to space platforms, and as a result the flight systems currently in space are mostly composed of lower-performance frequency-limited software processors and non-scalable bus interconnects. These current architectures may be inadequate for supporting real-time, on-board processing of sensor data of sufficient volume, and thus new components and novel architectures need to be explored in order to enable high-performance computing in space.

Radar satellites have a potentially wide field of view looking down from space, but maintaining resolution at that viewpoint results in very large data sets. Furthermore, target tracking algorithms such as Ground Moving Target Indicator (GMTI) have tight real-time processing deadlines of consecutive radar returns in order to keep up with the incoming sensor data as well as produce useful (i.e., not stale) target detections. In previous work, it has been shown using simulation [1-3] that on-board processing of high-resolution data on radar satellites requires parallel processing platforms providing high throughput to all processing nodes, and efficient processing engines to keep up with the strict real-time deadlines and large data sets.

Starting in 1997, Motorola began to develop a next-generation, high-performance, packet-switched embedded interconnect standard called RapidIO, and soon after partnered with Mercury Computers to complete the first version of the RapidIO specification. After the release of the 1.0 specification in 1999, the RapidIO Trade Association was formed, a non-profit corporation composed of a collection of industry partners for the purpose of steering the development and adoption of the RapidIO standard. Designed specifically for embedded environments, the

RapidIO standard seeks to address many of the challenges faced by embedded systems, such as providing a small footprint for lower size and power, inherent fault tolerance, high throughput, and scalability. Additionally, the RapidIO protocol includes flow control and fault isolation, features that enhance the fault tolerance of a system but are virtually non-existent in bus-based interconnects. These characteristics make the RapidIO standard an attractive option for embedded space platforms, and thus in-depth research is warranted to consider the feasibility and suitability of RapidIO for space systems and applications.

In addition to the emergence of high-performance interconnects, the embedded community has also been increasingly adopting the use of hardware co-processing engines in order to boost system performance. The main types of processing engines used aside from conventional processors are application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). While both components provide the potential for great clock-cycle efficiency through exploitation of parallelism and custom-tailored architectures for performing a particular processing task, the ASIC devices are much more expensive to design and fabricate than FPGAs. By contrast, with sufficient time and resources, ASIC devices can achieve comparable or better performance than FPGAs, as well as lower power consumption due to the removal of unnecessary logic that would be present in any FPGA design. Unfortunately, ASICs are rarely re-usable for many different applications, where FPGAs may be completely reconfigured for a wide range of end uses. FPGAs are also considered valuable prototyping components due to their flexible, re-programmable nature.

However, it will be shown that it can be difficult at this point in time to construct a complete system based solely on hardware components (e.g. FPGAs). Application designers are rarely fluent in the various hardware description languages (HDL) used to develop FPGA

designs. Ideally, application engineers should still be able to develop applications in the familiar software setting, making use of the hardware processing resources of the system transparently through function calls in software. Furthermore, most flight systems have extensive middleware (software) for system monitoring and control, and porting these critical legacy software components to hardware would be fought tooth-and-nail, if it were even possible. As will be shown in my research, there is still an important role for software in these high-performance embedded systems, and gaining insight into the optimal combination of hardware and software processing for this application domain is an important objective.

My work conducted an experimental study of cutting-edge system architectures for Space-Based Radar (SBR) using FPGA processing nodes and the RapidIO high-performance interconnect for embedded systems. Hardware processing provides the clock-cycle efficiency necessary to enable high-performance computing with lower-frequency, radiation-hardened components. Compared to bus-based designs, packet-switched interconnects such as RapidIO will substantially increase the scalability, robustness, and network performance of future embedded systems, and the small footprint and fault tolerance inherent to RapidIO suggest that it may be an ideal fit for use with FPGAs in space systems. The data input and output requirements of the processing engines, as well as the data paths provided by the system interconnect and the local memory hierarchies are key design parameters that affect the ultimate system performance. By experimenting with assorted node architecture variations and application communication schedules, this research is able to identify critical design features as well as suggest enhancements or alternative design approaches to improve performance for SBR applications.

An experimental testbed was built from the ground up to provide a platform on which to perform experimentation. Based on Xilinx FPGAs, prototyping boards, and RapidIO IP cores, the testbed was assembled into a complete system for application development and fine-grained performance measurement. When necessary, additional printed circuit boards (PCBs) were designed, fabricated, assembled, and integrated with the Xilinx prototyping boards and FPGAs in order to enhance the capabilities of each testbed node. Moving down one level in the architecture, a top-level processing node architecture was needed with which to program each of the FPGAs of the system. The processing node architecture is equally important to performance as is the system topology, and the flexible, customizable nature of FPGAs was leveraged in order to propose a novel, advanced chip-level architecture connecting external memory, network fabric, and processing elements. In addition to proposing a baseline node architecture, the reprogrammability of the FPGAs allows features and design parameters to be modified in order to alter the node architecture and observe the net impact on application performance. Each of the FPGAs in the testbed contains two embedded PowerPC405 processors, providing software processors to the system that are intimately integrated with the reconfigurable fabric of the FPGAs. Finally, co-processor engines are designed in HDL to perform the processing tasks of GMTI best suited for hardware processing.

The remainder of this document will proceed as follows. Chapter 2 will present the results of a literature review of related research, and provide background information relating to this thesis. Topics covered in Chapter 2 include GMTI and space-based radar, RapidIO, reconfigurable computing, as well as commercial embedded system hardware and architectures. Chapter 3 begins the discussion of original work, by describing in detail the complete architecture of the experimental testbed. This description includes the overall testbed topology

and components, the proposed processing node architecture (i.e. the overall FPGA design), as well as individual co-processor engine architectures used for high-speed processing. Chapter 4 will introduce the experimental environment in detail, as well as outline the experimental methods used for data collection. This overview includes experimental equipment and measurement procedures, testbed setup and operation, parameter definitions, and a description of the experiments that are performed. Chapter 5 will present the experimental results, as well as offer discussion and analysis of those results. Chapter 6 provides some concluding remarks that summarize the insight that was gained from this work, as well as some suggestions for future tasks to extend the findings of this research.

CHAPTER 2
BACKGROUND AND RELATED RESEARCH

Ground Moving Target Indicator

There are a variety of specific applications within the domain of radar processing, and even while considering individual applications there are often numerous ways to implement the same algorithm using different orderings of more basic kernels and operations. Since this research is architecture-centric in nature, to reduce the application space one specific radar application known as GMTI will be selected for experimentation. GMTI is used to track moving targets on the ground from air or space, and comes recommended from Honeywell Inc., the sponsor of this research, as an application of interest for their future SBR systems. Much research exists in academic and technical literature regarding GMTI and other radar processing applications [1-18], however most deal with airborne radar platforms.

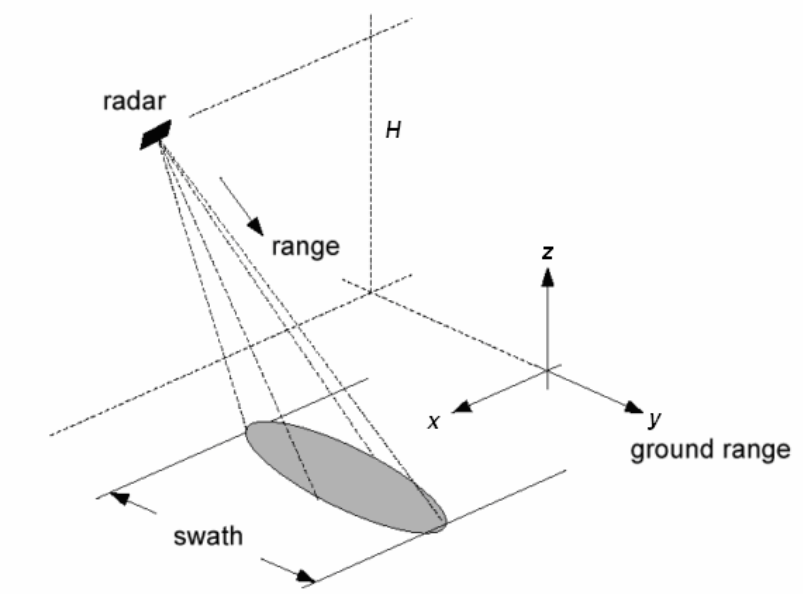


Figure 2-1. Illustration of satellite platform location, orientation, and pulse shape.

For the type of mission considered in this thesis research, the radar platform is mounted on the satellite, pointed downward towards the ground as shown in Figure 2-1. The radar sends out periodic pulses at a low duty cycle, with the long "silent" period used to listen for echoes. Each

echo is recorded over a period of time through an analog-to-digital converter (ADC) operating at a given sampling frequency, corresponding to a swath of discrete points on the ground that the transmitted pulse passes over [4]. Furthermore, each received echo is filtered into multiple frequency channels, to help overcome noise and electronic jamming. Thus, each transmitted pulse results in a 2-dimensional set of data, composed of multiple range cells as well as frequency channels. When multiple consecutive pulse returns are concatenated together, the result is a three-dimensional data-cube of raw radar returns that is passed to the high-performance system for processing (see Figure 2-2).

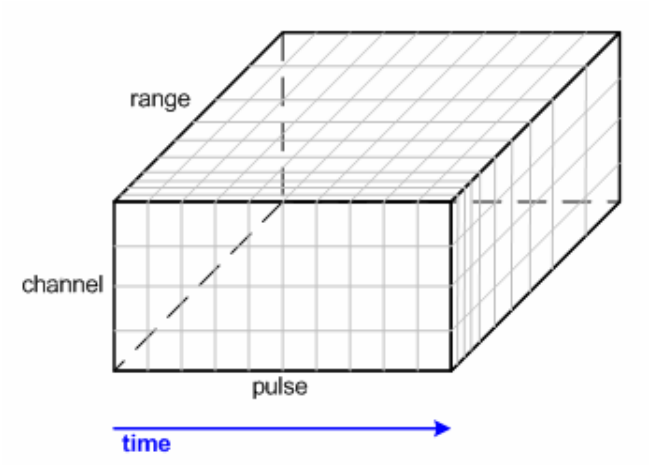


Figure 2-2. GMTI data-cube.

As mentioned above, periodically the radar will send a collection of data to the processing system. This period represents the real-time processing deadline of the system, and is referred to as the *Coherent Processing Interval* (CPI). Channel dimension lengths are typically small, in the 4-32 channel range, and typical pulse dimension lengths are also relatively small, usually between 64-256. However, the range dimension length will vary widely depending on radar height and resolution. Current GMTI implementations on aircraft have range dimensions on the order of 512-1024 cells [5-6, 12], resulting in data-cube sizes in the neighborhood of 32 MB or less. However, the high altitude and wide field of view of space-based radar platforms requires

dramatic increases in the range dimension in order to preserve resolution at that altitude. Space-based radar systems may have to deal with 64K-128K range cells, and even more in the future, which correspond to data-cubes that are 4 GB or larger.

GMTI is composed of a sequence of four sub-tasks, or kernels that operate on the data-cube to finally create a list of target detections from each cube [7]. These four sub-tasks are (1) pulse compression, (2) Doppler processing, (3) space-time adaptive processing, and (4) constant false-alarm rate (CFAR) detection. Figure 2-3 below illustrates the flow of GMTI.

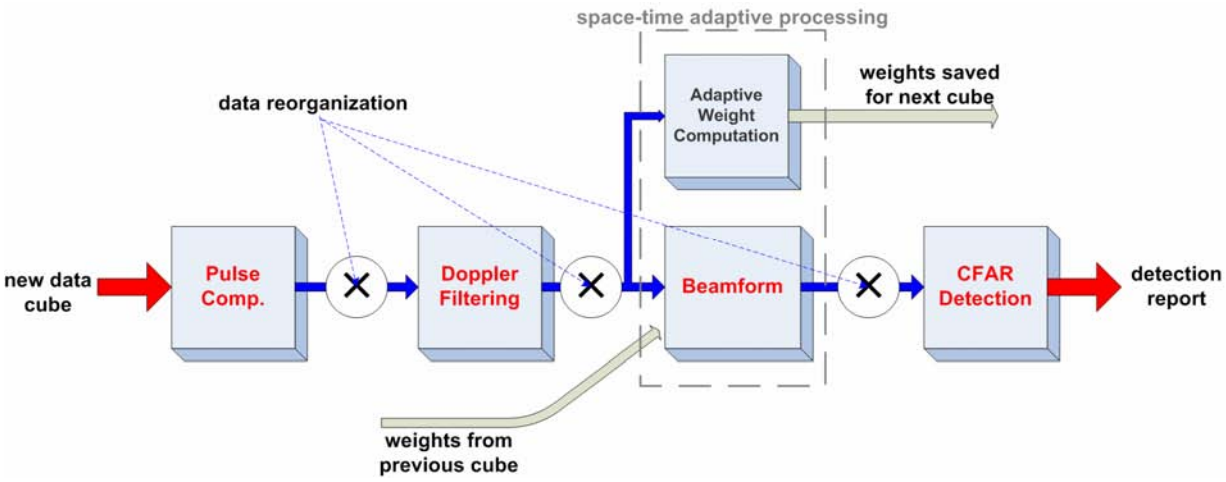


Figure 2-3. GMTI processing flow diagram.

Pulse Compression

The first stage of GMTI is a processing task associated with many radar applications, and is known as pulse compression. The reason why pulse compression is needed is based on the fact that the pulse transmitted by the radar is not instantaneous. In an ideal world, the radar could send out an instantaneous pulse, or impulse, that would impact each point on the ground instantly, and the echo received by the radar would be a perfect reflection of the ground. However, in the real world, the radar pulse is not instantaneous. As the pulse travels over the ground, the radar receives echoes from the entire length of the pulse at any point in time. Thus, what the radar actually receives as the echo is an image of the surface convolved with the shape

of the transmitted pulse. Pulse compression is used to de-convolve the shape of the transmitted pulse (which is known and constant) from the received echo, leaving a "cleaned up" image of the surface [8].

One benefit of pulse compression is that it can be used to reduce the amount of power consumed by the radar when transmitting the pulses. A tradeoff exists where shorter pulses result in less-convolved echoes, however require higher transmitting power in order to retain an acceptable signal-to-noise ratio. By transmitting longer pulses, less power is required, but of course the echo will be more "dirty" or convolved. Pulse compression thus enables the radar to operate with lower power using longer pulses [8] and still obtain clean echoes.

In addition to the raw radar data-cube (represented by the variable $\boldsymbol{a}$), pulse compression uses a constant complex vector representing a frequency-domain recreation of the original radar pulse. This vector constant, $\boldsymbol{F}$, can be pre-computed once and re-used indefinitely [9]. Since convolution can be performed by a simple multiplication in the frequency domain, pulse compression works by converting each range line of the data-cube $\boldsymbol{a}$ (see Figure 2-4, henceforth referred to as a *Doppler bin*) into the frequency domain with an FFT, performing a point-by-point complex vector multiplication of the frequency-domain Doppler bin with the complex conjugate of the vector constant $\boldsymbol{F}$ (for de-convolution), and finally converting the Doppler bin back into the time domain with an IFFT. Each Doppler bin can be processed completely independently [10].

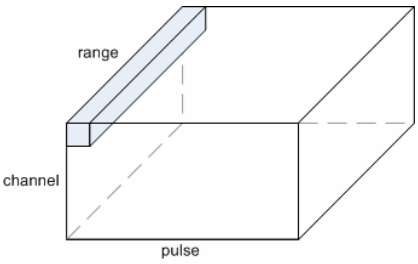


Figure 2-4. Processing dimension of pulse compression, highlighting one Doppler bin.

The operations of pulse compression are defined mathematically by the following expressions, where *R*, *P*, and *C* represent the lengths of the range, pulse, and channel dimensions, respectively:

$$A_{c,p,k} = \sum_{r=0}^{R-1} a_{c,p,r} \cdot e^{-\frac{2\pi \cdot i}{R} rk} \qquad c = 0,...,C-1 \quad p = 0,...,P-1 \quad k = 0,...,R-1 \qquad (2.1)$$

$$A'_{c,p,k} = A_{c,p,k} \cdot F_k \qquad c = 0,...,C-1 \quad p = 0,...,P-1 \quad k = 0,...,R-1 \qquad (2.2)$$

$$a'_{c,p,r} = \sum_{k=0}^{R-1} A'_{c,p,k} \cdot e^{\frac{2\pi \cdot i}{R} rk} \qquad c = 0,...,C-1 \quad p = 0,...,P-1 \quad r = 0,...,R-1 \qquad (2.3)$$

The output, ***a'***, represents the pulse-compressed data-cube and is now passed to Doppler processing after a re-organization of the data in system memory (the data reorganization operation will be addressed later).

Doppler Processing

Doppler processing performs operations similar to pulse compression, although for a different reason. Doppler processing is performed along the pulse dimension, and performs *apodization* on the data-cube before converting it to the frequency domain along the pulse dimension. Apodization is done by a complex vector multiplication of each pulse line of the data-cube (henceforth referred to as a *range bin*, see Figure 2-5) by a constant, pre-computed complex time-domain vector [10]. There are several standard apodization functions used for Doppler processing, a few examples of which are given in Figure 2-6. The choice of apodization function depends on the properties of the radar, and has no impact on the performance of the application.

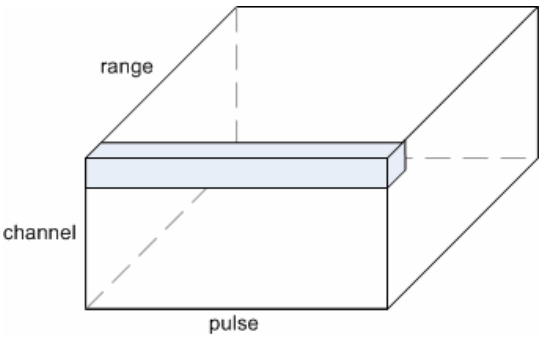


Figure 2-5. Processing dimension of Doppler processing, highlighting one range bin.

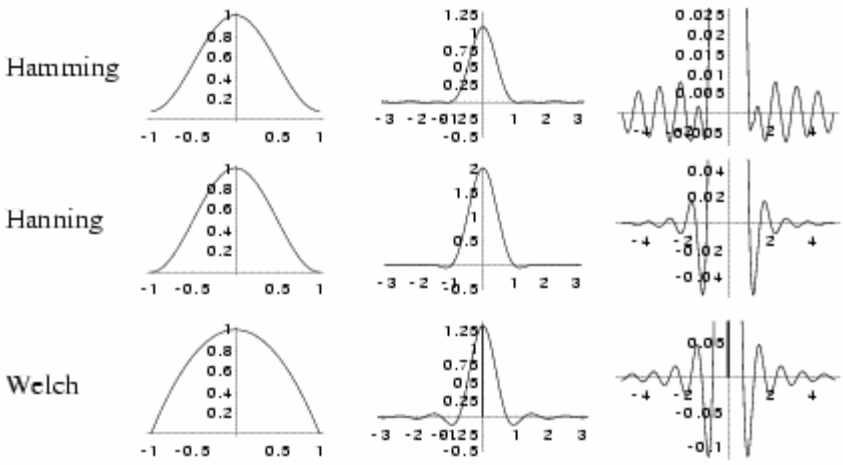


Figure 2-6. Examples of common apodization functions, time-domain (left) and frequency-domain (middle and right).

Each range bin of the data-cube ***a'*** is processed completely independently, and is first multiplied by the apodization vector, ***g***. The resulting range bin is then converted to the frequency domain with an FFT, in preparation for the next stage. These operations are defined mathematically by the following expressions:

$$b_{c,p,r} = a'_{c,p,r} \cdot g_p \qquad c = 0,...,C-1 \quad p = 0,...,P-1 \quad r = 0,...,R-1 \qquad (2.4)$$

$$B_{c,k,r} = \sum_{p=0}^{P-1} b_{c,p,r} \cdot e^{-\frac{2\pi \cdot i}{P} pk} \qquad c = 0,...,C-1 \quad k = 0,...,P-1 \quad r = 0,...,R-1 \qquad (2.5)$$

The output, ***B***, represents the processed data-cube that is ready for beamforming in the next stage, after yet another re-organization of the data in system memory.

Space-Time Adaptive Processing

Space-time adaptive processing (STAP) is the only part of the entire GMTI application that has any temporal dependency between different CPIs. There are two kernels that are considered to be a part of STAP: (1) adaptive weight computation (AWC), and (2) beamforming. While AWC is defined as a part of STAP, it is actually performed in parallel with the rest of the entire algorithm (recall Figure 2-3). For a given data-cube, the results of AWC will not be used until the beamforming stage of the next CPI. Beamforming, by contrast, is in the critical path of processing, and proceeds immediately following Doppler processing using the weights provided by AWC on the previous data-cube [10-12].

Adaptive Weight Computation

Adaptive weight computation is performed in parallel with the rest of the stages of GMTI, and is not considered to be a performance-critical step given the extended amount of time available to complete the processing. The output of AWC is a small set of weights known as the *steering vector*, and is passed to the beamforming stage. Since AWC does not lie in the critical path of processing, this stage is omitted in the implementation of GMTI for this thesis in order to reduce the necessary attention paid to the application for this architecture-centric research. For more information on the mathematical theory behind AWC and the operations performed, see [10, 12].

Beamforming

Beamforming takes the weights provided by the AWC step, and projects each channel line of the data-cube into one or more "beams" through matrix multiplication [10-13]. Each beam represents one target classification, and can be formed completely independently. The formation of each beam can be thought of as filtering the data-cube to identify the probability that a given cell is a given type or class of target [14]. As the final step of beamforming, the magnitude of

each complex element of the data-cube is obtained in order to pass real values to CFAR detection in the final stage. Figure 2-7 below shows the processing dimension of beamforming, along with a visualization of the reduction that occurs to the data-cube as a result of beamforming.

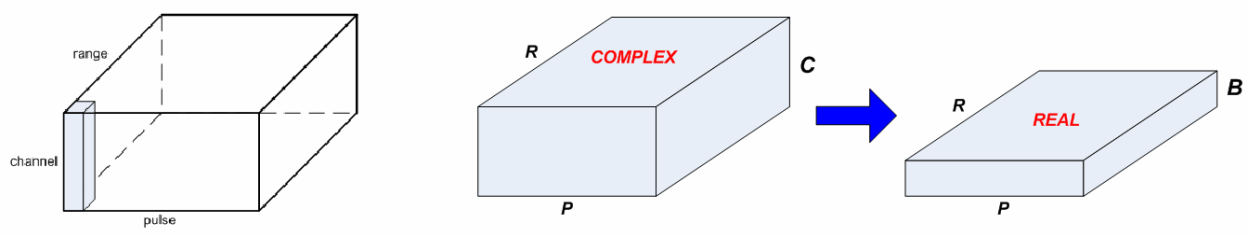


Figure 2-7. Processing dimension of beamforming (left), and diagram illustrating data-cube reduction that occurs as a result of beamforming (right).

Taking the processed, reorganized data-cube ***B*** from Doppler processing, as well as the steering vector $\boldsymbol{\omega}$ from AWC, the beams are formed with a matrix multiplication followed by a magnituding operation on each element of the resulting beams. These operations are defined mathematically by the following expressions:

$$B'_{b,p,r} = \sum_{c=0}^{C-1} \omega_{b,c} \cdot B_{c,p,r} \qquad b = 0,\ldots,B-1 \quad p = 0,\ldots,P-1 \quad r = 0,\ldots,R-1 \qquad (2.6)$$

$$C_{b,p,r} = \sqrt{B'_{b,p,r}\big|^2_{real} + B'_{b,p,r}\big|^2_{imag}} \qquad b = 0,\ldots,B-1 \quad p = 0,\ldots,P-1 \quad r = 0,\ldots,R-1 \qquad (2.7)$$

The output cube, C, is a real-valued, completely processed and beamformed data-cube. The data-cube undergoes one final reorganization in system memory before being passed to the final stage of processing, CFAR detection.

Constant False-Alarm Rate Detection

Constant False-Alarm Rate detection, or CFAR, is another kernel commonly associated with radar applications [15]. CFAR is used in this case as the final stage in GMTI processing, and makes the ultimate decision on what is and what is not a target. The goal of CFAR is to

minimize both the number of false-positives (things reported as targets that really are not targets) as well as false-negatives (things not reported as targets that really are targets). The output of CFAR is a detection report, containing the range, pulse, and beam cell locations of each target within the data-cube, as well as the energy of each target.

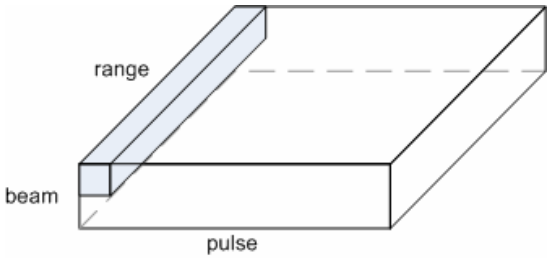


Figure 2-8. Processing dimension of CFAR detection.

CFAR operates along the range dimension (see Figure 2-8 above) by sliding a window along each Doppler bin, computing local averages and performing threshold comparisons for each cell along the range dimension. In addition to the coarse-grained, embarrassingly parallel decomposition of the data-cube across multiple processors, the computations performed by the sliding window contain a significant amount of parallelism and that may also be exploited to further reduce the amount of computations required per Doppler bin [15]. This fine-grained parallelism is based on reusing partial results from previously computed averages, and will be described in detail later in Chapter 3. The computations of CFAR on the data-cube $\boldsymbol{C}$ can be expressed mathematically as:

$$T_{b,p,r} = \frac{1}{2N_{cfar}} \cdot \sum_{i=G+1}^{G+N_{cfar}} \left[\left| C_{b,p,r+i} \right|^2 + \left| C_{b,p,r-i} \right|^2 \right] \tag{2.8}$$

$$\frac{\left| C_{b,p,r} \right|^2}{T_{b,p,r}} > \mu \quad ? \qquad b = 0,\ldots,B-1 \quad p = 0,\ldots,P-1 \quad r = 0,\ldots,R-1 \tag{2.9}$$

Where G represents the number of guard cells on each side of the target cell and N_{cfar} represents the size of the averaging window on either side of the target cell (see Figure 2-9).

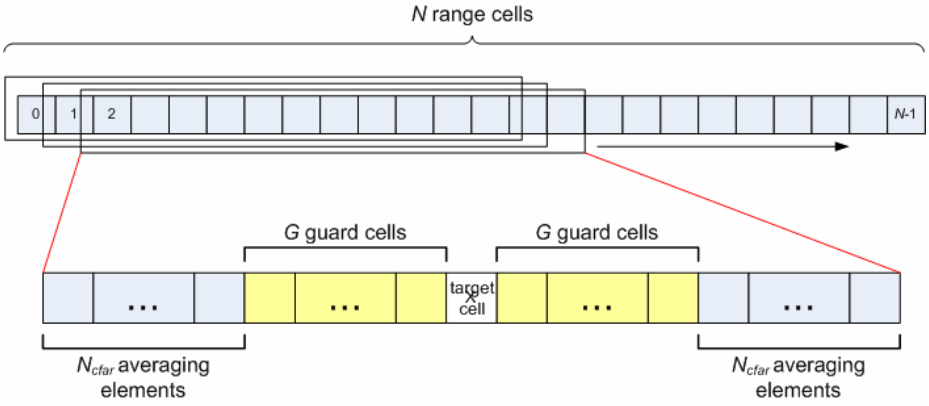


Figure 2-9. CFAR sliding window definition.

For each cell in the Doppler bin, a local average is computed of the cells surrounding the target cell (the target cell is the cell currently under consideration). A small number of cells immediately surrounding the target cell, referred to as guard cells, are ignored. Outside of the guard cells on either side of the target cell, a window of N_{cfar} cells are averaged. This average is then scaled by some constant, μ. The value of the target cell is then compared to the scaled average, and if the target cell is greater than the scaled average then it is designated as a target.

The constant μ is very important to the algorithm, and is the main adjustable parameter that must be tuned to minimize the false-positives and false-negatives reported by CFAR. This parameter is dependent on the characteristics of the specific radar sensor, and the theory behind its determination is beyond the scope or focus of this thesis research. Fortunately, the value of this constant does not affect the performance of the application. One final detail of the algorithm concerns the average computations at or near the boundaries of the Doppler bin. Near the beginning or end of the Doppler bin, one side of the sliding window will be "incomplete" or un-filled, and thus only a right-side or left-side scaled average comparison can be used near the boundaries as appropriate to handle these conditions [16].

Corner Turns

Another important operation for GMTI, and many other signal processing applications, is that of re-organization of data for different processing tasks. In order to increase the locality of memory accesses within any given computational stage of GMTI, it is desirable to organize the data in memory according to the dimension that the next processing stage will operate along. In distributed memory systems, where the data-cube is spread among all of the processing memories and no single node contains the entire data-cube, these corner-turns, or distributed transposes can heavily tax the local node memory hierarchies as well as the network fabric. It should be noted here that with GMTI, the corner-turns constitute the only inter-processor communication associated with the algorithm [10, 17]. Once all data is organized properly for a given stage of GMTI, processing may proceed at each node independent of one another in an embarrassingly parallel fashion. Thus, it is the performance of the corner-turn operations as you scale the system size up that determines the upper-bound on parallel efficiency for GMTI.

It is important to understand the steps required for a corner-turn, at the memory access level, in order to truly understand the source of inefficiency for this notorious operation. The basic distributed corner-turn operation can be visualized by observing Figure 2-10 below.

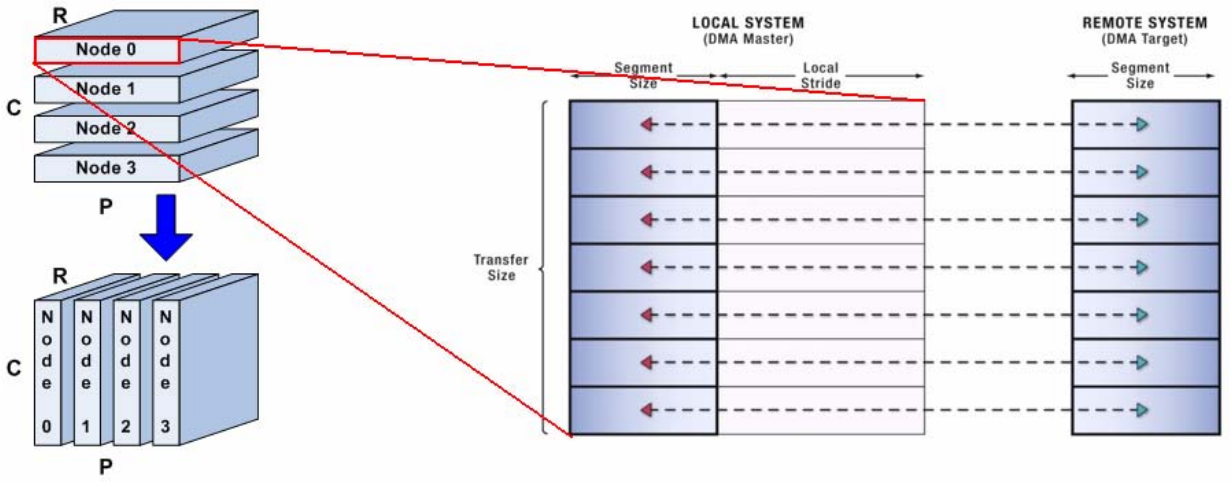


Figure 2-10. Basic corner turn

First, consider the most basic decomposition of the data-cube for any given task, along a single dimension of the cube (see Figure 2-10, left side). Assume the data-cube is equally divided among all the nodes in the system, and must be re-distributed among the nodes as illustrated. It can be seen that each node will have to send a faction of its current data set to each other node in the system, resulting in a *personalized all-to-all* communication event, which heavily taxes the system backplane with all nodes attempting to send to one another at the same time. Simulations in [3] suggest that careful synchronization of the communication events can help to control contention and improve corner-turn performance. However, look closer at the local memory access pattern at each node. The local memory can be visualized as shown to the right side of Figure 2-10. The data to be sent to each node does not completely lie in consecutive memory locations, and still must be transposed at some point. The exact ordering of the striding, grouping, and transposition is up to the designer, and will serve as a tradeoff study to be considered later. Because of this memory locality issue, corner-turns pose efficiency challenges both at the system interconnect level as well as at the node level in the local memory hierarchy.

A workshop presentation from Motorola [18] suggests an alternate approach to decomposition of data-cubes for GMTI, in order to improve the performance of corner-turn operations. Instead of decomposition along a single dimension, since all critical stages of GMTI operate on 1-dimensional data, the data-cube can be decomposed along two dimensions for each stage. The resulting data-cube would appear decomposed as shown below in Figure 2-11, showing two different decompositions to help visualize where the data must move from and to. The main advantage of this decomposition is that in order to re-distribute the data, only groups of nodes will need to communicate with one-another, as opposed to a system-wide all-to-all communication event.

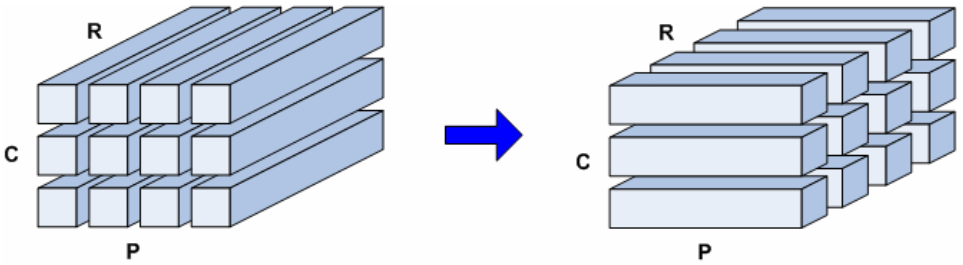
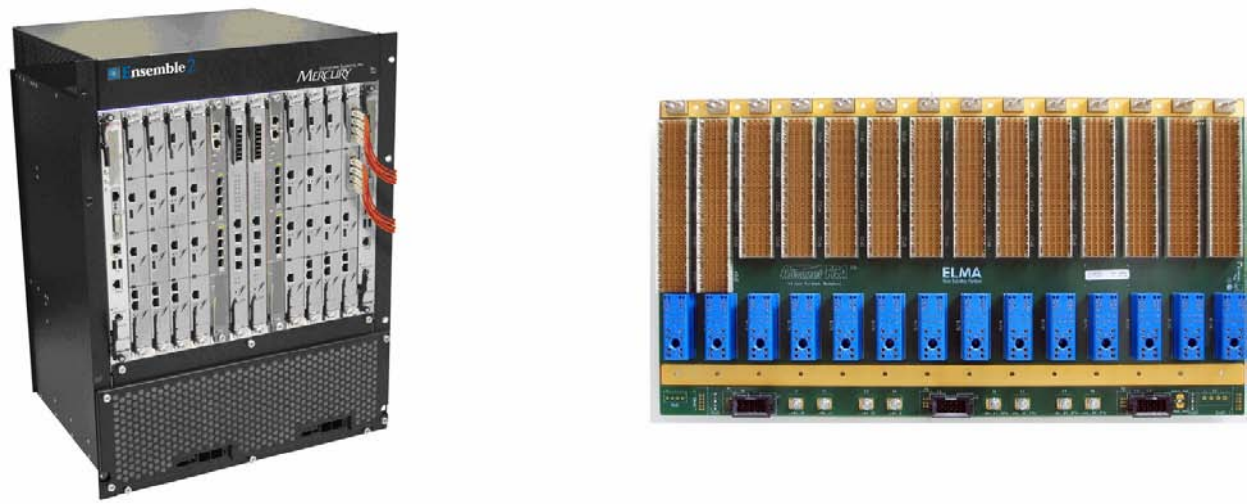


Figure 2-11. Proposed data-cube decomposition to improve corner-turn performance.

Other previous work [1] suggests a processor allocation technique to improve GMTI performance, and fits well with the decomposition approach suggested above. This allocation technique suggests that instead of using a completely data-parallel or completely pipelined implementation, a "staggered" parallelization should be adopted which breaks the system up into groups of nodes. Each group of nodes receives an entire data-cube, and performs all stages of GMTI in a data-parallel fashion. Consecutive data-cubes are handed to different groups in a round-robin fashion, thus the "staggered" title. The benefit of this processor allocation scheme is that the entire backplane is not taxed for corner-turns, instead only the local interconnect for each group of processors. Tuned to a specific topology, this processor allocation strategy could fit well with proper decomposition boundaries of the data-cube as suggested above, further improving overall application performance.

Embedded System Architectures for Space

Given the unique physical environment and architecture of satellite systems, it is worth reviewing typical embedded hardware to be aware of what kinds of components are available, and where the market is moving for next-generation embedded systems. Many high-performance embedded processing systems are housed in a small (<2 ft. × <2 ft. × <2 ft.) metal box called a *chassis* (example pictured in Figure 2-12(a)). The chassis provides a stable structure in which to install cards with various components, as well as one central "motherboard" or *backplane* that is a critical component of the embedded system.



A B

Figure 2-12. Example pictures of chassis and backplane hardware; A) Mercury's Ensemble2 Platform, an ATCA chassis for embedded processing, and B) example of a passive backplane.

The backplane is a large printed circuit board (PCB) that provides multiple sockets as well as connectivity between the sockets. There are two general classes of backplanes, passive backplanes and active backplanes. Active backplanes include components such as switches, arbiters, and bus transceivers, and actively drive the signals on the various sockets. Passive backplanes, by contrast, only provide electrical connections (e.g. copper traces on the PCB) between pins of the connectors, and the cards that are plugged into the sockets determine what kind of communication protocols are used. Passive backplanes are much more popular than active backplanes, as is indicated by the wide availability of passive backplane products compared to the scarcity of active backplane products. Figure 2-12(b) shows an example of a passive backplane PCB with a total of 14 card sockets.

However, even passive backplanes put some restriction on what kinds of interconnects can be used. Most importantly (and obviously), is the pin-count of the connectors, as well as the width and topology of the connections between sockets. Most backplanes are designed with one or more interconnect standards in mind, and until recently almost all backplane products were

built to support some kind of bus interconnect. For example, for decades one of the most popular embedded system backplane standards has been the VME standard [19].

In response to the growing demand for high-performance embedded processing systems, newer backplane standards are emerging, such as the new VME/VXS extension or the ATCA standard, that are designed from the ground-up with high-speed serial packet-switched interconnects and topologies in mind [19, 20]. In addition to providing the proper connection topologies between sockets to support packet-switched systems, extra care and effort must be put into the design and layout of the traces, as well as the selection of connectors, in order to support high-speed serial interconnects with multi-GHz speeds. High-speed signals on circuit boards are far more susceptible to signal-integrity hazards, and much effort is put into the specification of these backplane standards, such as VXS or ATCA, in order to provide the necessary PCB quality to support a number of different high-speed serial interconnect standards. Figure 2-13 shows three of the most popular serial backplane topologies:

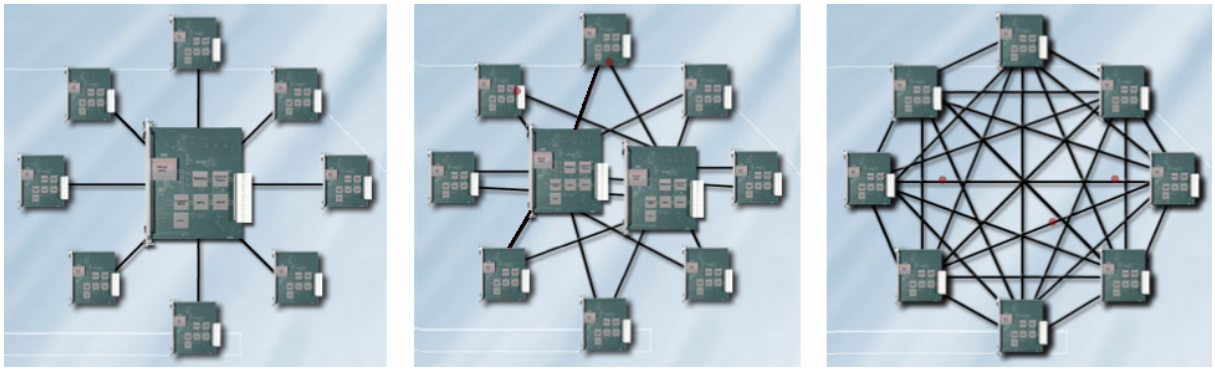


Figure 2-13. Three popular serial backplane topologies: A) star, B) dual-star, and C) full-mesh.

Generally, a serial backplane such as VXS or ATCA will have one or more switch card slots, and several general-purpose card slots. The example backplane pictured back in Figure 2-12(b) shows an example of this, as the two slots to the far left are larger than the rest, indicating

that those slots are the switch sockets. Most likely, the backplane picture in Figure 2-12(b) is a dual-star topology as is pictured above in Figure 2-13(b). As mentioned previously, the passive backplane itself does not dictate which interconnect must be used, although it may restrict which interconnects *can* be used. The cards that are plugged into the backplane will dictate what the system interconnect is, as well as what processing and memory resources are to be a part of the system.

There are many different products for processor, memory, and switch cards, all of which conform to one chassis/backplane standard or another. They combine any number of software processors, FPGAs and ASICs, and various memory devices, as well as connectivity through a switch or some other communication device to the backplane connector. While there are far too many products to review in this document, there was one particular publication at a recent High-Performance Embedded Computing (HPEC) workshop that is very relevant to this project, and warrants a brief discussion. The company SEAKR published a conference paper in 2005 highlighting several card products specifically designed for space-based radar processing [21]. These products include a switch card, a processor card, and a mass memory module, all of which are based on FPGAs and use RapidIO as the interconnect. The architecture of each board is detailed, revealing a processor board that features 4 FPGAs connected together via a RapidIO switch, with several other ports of the switch connecting to ports leading off the board to the backplane connector. Beyond the details of each board architecture, seeing such products begin to emerge during the course of this thesis research is a positive indication from the industry that pushing towards the use of FPGAs and RapidIO for high-performance radar and signal processing is indeed a feasible and beneficial direction to go.

Field-Programmable Gate Arrays

By this point in time, FPGAs have become fairly common and as such this thesis is written assuming that the reader already has a basic understanding of the architecture of FPGAs. Nevertheless, this technology is explicitly mentioned here for completeness, and to ensure the reader has the necessary basic knowledge of the capabilities and limitations of FPGAs in order to understand the designs and results presented later. For a detailed review of basic background information and research regarding FPGAs, see [22-27]. Since this research involved a lot of custom low-level design with the FPGAs, several documents and application notes from Xilinx were heavily used during the course of the research [28-33].

RapidIO

RapidIO is an open-standard, defined and steered by a collection of major industry partners including Motorola (now Freescale) Semiconductor, Mercury Computers, Texas Instruments, and Lucent Technologies, among others. Specially designed for embedded systems, the RapidIO standard seeks to provide a scalable, reliable, high-performance interconnect solution designed specifically to alleviate many of the challenges imposed upon such systems [34, 35]. The packet-switched protocol provides system scalability while also including features such as flow control, error detection and fault isolation, and guaranteed delivery of network transactions. RapidIO also makes use of Low-Voltage Differential Signaling (LVDS) at the pin-level, which is a high-speed signaling standard that enjoys higher attainable frequencies and lower power consumption due to low voltage swing, as well as improved signal integrity due to common-mode noise rejection between signal pairs (at the cost of a doubling of the pin count for all LVDS connections).

As a relatively new interconnect standard, there is a vacuum of academic research in conference and journal publications. In fact, at the conception of this research project in August

2004, there was only a single known academic research paper that featured a simulation of a RapidIO network [36], although the paper was brief and ambiguous as far as definitions of the simulation models and other important details. The majority of literature used to research and become familiar with RapidIO comes from industry whitepapers, conference and workshop presentations from companies, and standards specifications [34-42]. This thesis research [42], and the numerous publications to come from this sponsored project [1-3, 17], contribute much-needed academic research into the community investigating this new and promising high-performance embedded interconnect technology.

As defined by the standard, a single instance of a RapidIO "node," or communication port, is known as an "endpoint." The RapidIO standard was designed from the start with an explicit intention of keeping the protocol simple in order to ensure a small logic footprint, and not require a prohibitive amount of space on an IC to implement a RapidIO endpoint [35]. The RapidIO protocol has three explicit layers defined, which map to the classic Layered OSI Architecture as shown below in Figure 2-14. The RapidIO logical layer handles end-to-end transaction management, and provides a programming model to the application designer suited for the particular application (e.g. Message-Passing, Globally-Shared Memory, etc). The transport layer is very simple, and in current implementations is often simply combined with the logical layer. The transport layer is responsible for providing the routing mechanism to move packets from source to destination using source and destination IDs. The physical layer is the most complex part of the RapidIO standard, and handles the electrical signaling necessary to move data between two physical devices, in addition to low-level error detection and flow control.

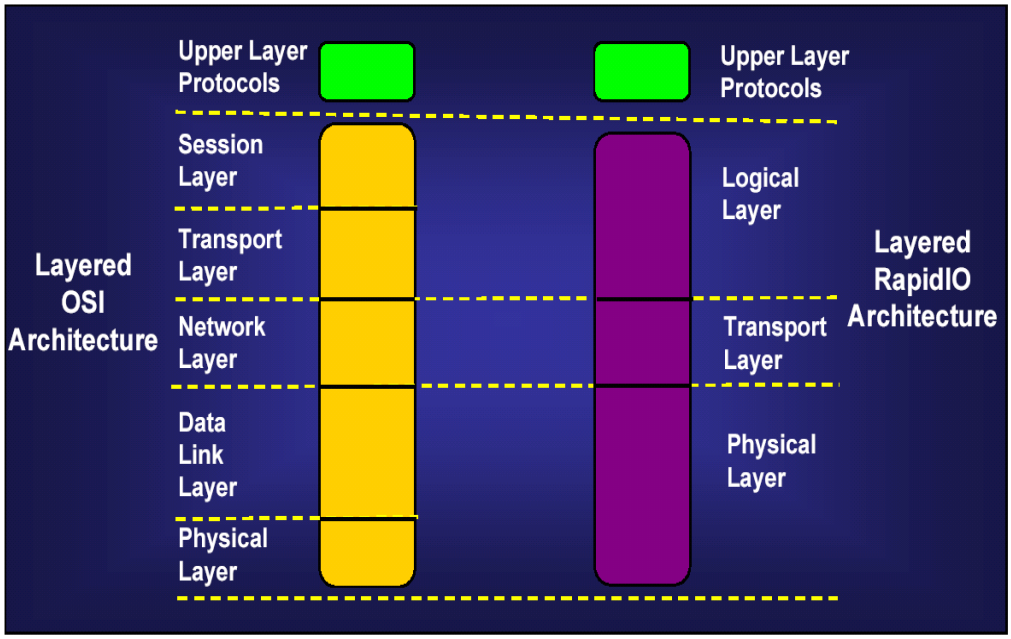


Figure 2-14. Layered RapidIO architecture vs. layered OSI architecture. RapidIO has multiple physical layer and logical layer specifications, and a common transport layer specification [35].

RapidIO offers different variations of the logical layer specification, specifically the message passing logical layer, the memory-mapped logical I/O logical layer, as well as the globally-shared memory logical layer [38, 39]. Both the logical I/O and the globally-shared memory logical layers are designed to provide direct, transparent access to remote memories through a global address space. Any node can read or write directly from/to the memory of any other node connected to the RapidIO fabric, simply by accessing an address located in the remote node's address space. The major difference between these two logical layer variants is that the globally-shared memory logical layer offers cache-coherency, while the simpler logical I/O logical layer provides no coherence. The message passing logical layer, by contrast, is somewhat analogous to the programming model of message passing interface (MPI), where communication is carried out through explicit send/receive pairs. With the logical I/O and globally-shared memory logical layers, the programmer has the option for some request types of requesting explicit acknowledgement through the logical layer at the receiving end, or otherwise

allowing the physical layer to guarantee successful delivery without requiring end-to-end acknowledgements for every transfer through logical layer mechanisms. Regardless of which logical layer variant is selected for a given endpoint implementation, the logical layer is defined over a single, simple, common transport layer specification.

The RapidIO physical layer is defined to provide the designer a range of potential performance characteristics, depending on the requirements of his or her specific application. The RapidIO physical layer is bi-directional or full-duplex, is sampled on both rising and falling edges of the clock (DDR), and defines multiple legal clock frequencies. Individual designers may opt to run their RapidIO network at a frequency outside of the defined range. All RapidIO physical layer variants can be classified under one of two broad categories; (1) Parallel RapidIO, or (2) Serial RapidIO. Parallel RapidIO provides higher throughputs over shorter distances, and has both an 8-bit and a 16-bit variant defined as part of the RapidIO standard [38]. Serial RapidIO is intended for longer-distance, cable or backplane applications, and also has two different variants defined in the standard, the single-lane 1x Serial and the 4-lane 4x Serial [40].

Parallel RapidIO is source-synchronous, meaning that a clock signal is transmitted along with the data, and the receive logic at each end point is clocked on the received clock signal. Serial RapidIO, by contrast, uses 8b/10b encoding on each serial lane in order to enable reliable clock recovery from the serial bit stream at the receiver. This serial encoding significantly reduces the efficiency of Serial RapidIO relative to parallel RapidIO, but nearly all high-performance serial interconnects use this 8b/10b encoding, so Serial RapidIO does not suffer relative to other competing serial standards. Figure 2-15 depicts both RapidIO physical layer flavors along with several other high-performance interconnects, to show the intended domain of each.

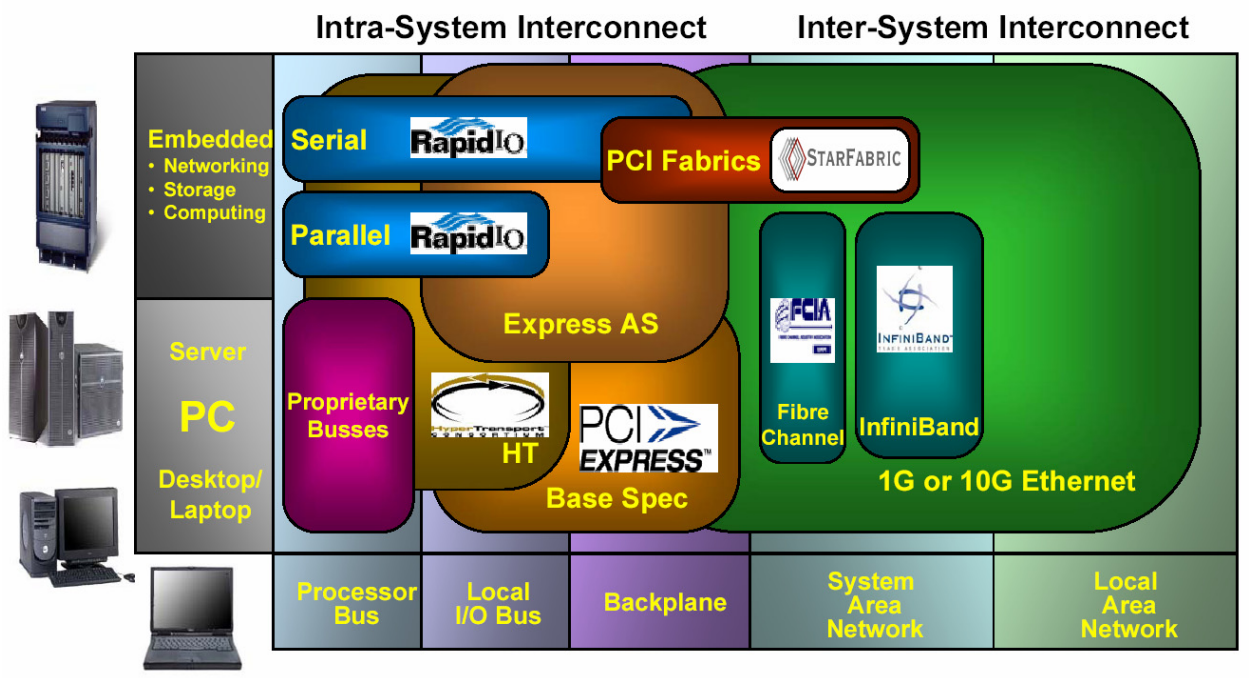


Figure 2-15. RapidIO compared to other high-performance interconnects [35].

There are two basic packet types in the RapidIO protocol; (1) standard packets, and (2) control symbols. Standard packets include types such as request packets and response packets, and will include both a header as well as some data payload. Control symbols are small 32-bit "symbols" or byte patterns that have special meanings and function as control messages between communicating endpoints. These control symbols can be embedded in the middle of the transmission of a regular packet, so that critical control information does not have to wait for a packet transfer to complete in order to make it across the link. This control information supports physical layer functions such as flow control, link maintenance, and endpoint training sequence requests, among others.

Error detection in RapidIO is accomplished at the physical layer in order to catch errors as early as possible, in order to minimize latency penalties. Errors are detected differently depending on the packet type, either through cyclic redundancy checks (CRC) on standard packets, or bitwise inverted replication of control symbols. For maximum-sized packets, the

RapidIO protocol will insert two CRC checksums into the packet, one in the middle and one at the end. By doing so, errors that occur near the beginning of a packet will be caught before the entire packet is transmitted, and can be stopped early to decrease the latency penalty on packet retries. This error detection mechanism in the physical layer is what the Logical I/O and Globally-Shared Memory logical layers rely upon for guaranteed delivery if the application design decides to not request explicit acknowledgements as mentioned previously. The physical layer will automatically continue to retry a packet until it is successfully transmitted, however without explicit acknowledgement through the logical layer, the user application may have no way of knowing when a transfer completes.

The RapidIO physical layer also specifies two types of flow-control; (1) transmitter-controlled flow-control, and (2) receiver-controlled flow-control. This flow-control mechanism is carried out by every pair of electrically-connected RapidIO devices. Both of these flow-control methods work as a "Go-Back-N" sliding window, with the difference lying in the method of identifying a need to re-transmit. The more basic type of flow-control is receiver-controlled, and relies on control symbols from the receiver to indicate that a packet was not accepted (e.g. if there is no more buffer space in the receiving endpoint). This flow-control method is required by the RapidIO standard to be supported by all implementations. The transmitter-controlled flow-control method is an optional specification that may be implemented at the designer's discretion, and provides slightly more efficient link operation. This flow-control variant works by having the transmit logic in each endpoint monitor the amount of buffer space available in its link partner's receive queue. This buffer status information is contained in most types of control symbols sent by each endpoint supporting transmitter-controller flow control, including idle symbols, so all endpoints constantly have up-to-date information regarding their partner's buffer

space. If a transmitting endpoint observes that there is no more buffer space in the receiver, the transmitting node will hold off transmitting until it observes the buffer space being freed, which results in less unnecessary packet transmissions.

The negotiation between receiver-controlled and transmitter-controlled flow-control is performed by observing a particular field of control symbols that are received. An endpoint that supports transmitter-controlled flow-control, endpoint *A*, will come out of reset assuming transmitter-controlled flow control. If endpoint *A* begins receiving control symbols from its partner, endpoint *B*, that indicate receiver-controlled flow control (i.e. the buffer status field of the symbol is set to a certain value), then endpoint *A* will revert to receiver-controlled flow-control by protocol with the assumption being that for some reason endpoint *B* does not support or does not currently want to support transmitter-controlled flow control. RapidIO endpoint implementations that do support both flow-control methods have a configuration register that allows the user to force the endpoint to operate in either mode if desired.

The RapidIO steering committee is continually releasing extensions to the RapidIO standard in order to incorporate changes in technology and enhance the capabilities of the RapidIO protocol for future embedded systems. One such example of an extension is the RapidIO Globally-Shared Memory logical layer, which was released after the Logical I/O logical layer specification. Other protocol extensions include flow control enhancements to provide end-to-end flow control for system-level congestion avoidance, a new data streaming logical layer specification with advanced traffic classification and prioritization mechanisms, and support for multi-cast communication events, to name a few examples. These newer RapidIO protocol extensions are not used directly in this research, however, and as such they are mentioned for completeness but will not be discussed in further detail.

CHAPTER 3
ARCHITECTURE DESCRIPTIONS

This chapter will provide a detailed description of the entire testbed, progressing in a top-down manner. In addition to defining the hardware structures, the application development environment will be described so as to identify the boundary between hardware and software in this system, as well as define the roles of each. The first section will introduce the entire testbed architecture from a system-level, followed by a detailed look at the individual node architecture in the second section. This node architecture is a novel design that represents a major contribution and focus of this thesis research. Finally, the last section will present and discuss the architectures of each of the hardware processing engines designed in the course of this research.

Testbed Architecture

Recall that the high-performance processing system will ultimately be composed of a collection of cards plugged into a backplane and chassis. These cards will contain components such as processors, memory controllers, memory, external communication elements, as well as high-speed sensor inputs and pre-processors. Figure 3-1 illustrates a conceptual high-performance satellite processing system, of which the main processing resources are multiple processor cards, each card containing multiple processors, each processor with dedicated memory. The multiple processors on each card are connected together via a RapidIO switch, and the cards themselves are connected to the system backplane also through the same RapidIO switches. While Figure 3-1 shows only three processor cards, a real system could contain anywhere from four to sixteen such cards. The system depicted in Figure 3-1 matches both the kind of system boasted by SEAKR in [21], as well as the target system described by Honeywell, the sponsor of this thesis research.

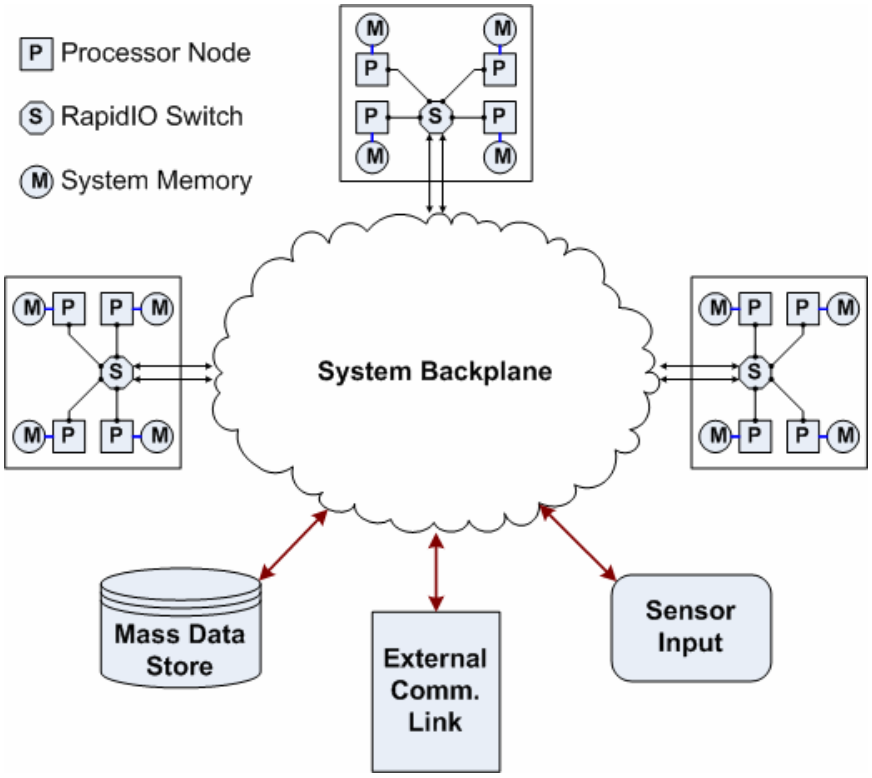


Figure 3-1. Conceptual radar satellite processing system diagram.

While it is cost-prohibitive to acquire or construct a full-scale system to match the conceptual hardware described above, it is possible to identify and capture the important parameters of such an architecture and study those parameters on a smaller hardware platform. There are three general parameters that affect overall application performance the most, (1) processor selection and performance, (2) processor-local memory performance, and (3) system interconnect selection and performance.

The experimental testbed used for this thesis research was designed from the ground-up using very basic "building blocks" such as FPGAs on blank prototyping boards, a donated RapidIO IP core (complements of Xilinx), custom-designed PCBs, and hand-coded Verilog and C. There are a total of two Xilinx Virtex-II Pro FPGAs (XC2VP40-FF1152-6) and prototyping boards (HW-AFX-FF1152-300), connected to each other over RapidIO. In addition to this dual-FPGA foundation, a small collection of custom-designed PCBs were fabricated and assembled in

order to enhance the capabilities of the FPGAs on the blank prototyping boards. These PCBs include items such as

- 2-layer global reset and configuration board
- 2-layer cable socket PCB, for high-speed cable connection to the pins of the FPGA
- 6-layer external SDRAM board: 128 MB, 64-bit @ 125 MHz
- 10-layer switch PCB, hosting a 4-port Parallel RapidIO switch from Tundra
- 2-layer parallel port JTAG cable adapter for the Tsi500 RapidIO switch

While the design, assembly, and testing of these circuits represented a major contribution of time and self-teaching, for this thesis research the circuits themselves are simply a means to an end to enable more advanced experimental case studies. As such, the details of these circuits such as the schematics, physical layouts, bill of materials, etc. are omitted from this thesis document. This information is instead provided as supplementary material (along with all Verilog and C source code for the testbed) in a separate document.

The overall experimental testbed is a stand-alone entity; however it does have several connections to a laboratory workstation for debugging and measurement. These connections will be discussed in detail in Chapter 4. All software written runs on the embedded PowerPC405 processors in the FPGAs, and all other functionality of the testbed is designed using Verilog and realized in the reconfigurable fabric of the FPGA. Each node (FPGA) of the testbed is assigned a node ID, which represents the upper address bits of the 32-bit global address space. One node is defined to be the master, and is responsible for initiating processing in the system. Communication between nodes, both data *and* control, is performed over RapidIO, by writing or reading to/from pre-defined locations in the external DRAM of remote nodes(Figure 3-2).

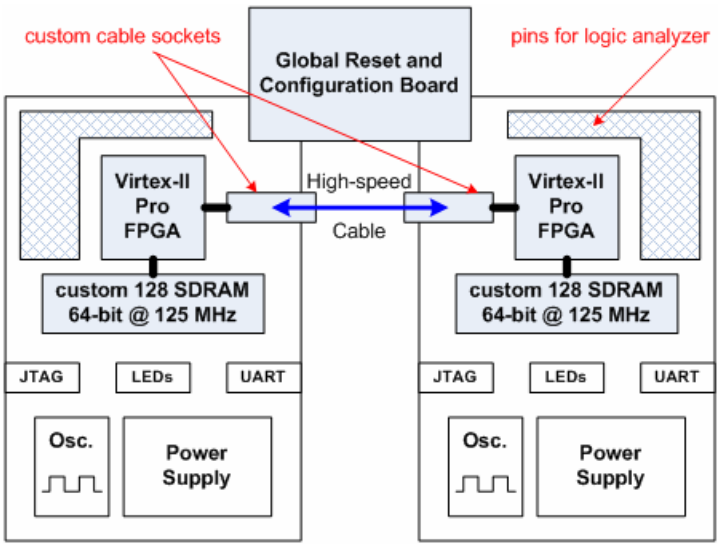


Figure 3-2. Experimental testbed system diagram.

Due to the potentially high cost of hardware, the high risk of prototype development, and the complexity of full-system design from near-zero infrastructure, several unavoidable restrictions were imposed by the experimental testbed used for this research. One major drawback to this hardware is the absence of larger-capacity external processing memory, such as QDR SRAM which is popular in today's embedded systems. The absence of this memory restricts the node to the memory available internal to the FPGA, which is much smaller. Because of this memory void, the maximum data-cube sizes able to be supported on this hardware are capped (though the performance results and conclusions drawn from those results remain valid). Furthermore, the high-risk nature of custom high-speed circuitry integration prevented the successful addition of a RapidIO switch to the testbed. In addition to limiting the testbed size, the absence of a switch also complicates the task of providing a high-speed data input into the system. As a result of that restriction, all data to be processed for a given experiment must be pre-loaded into system memory before processing begins, thus making a real-time processing demonstration impossible. Despite these challenges, an impressive system architecture is proposed and analyzed and valuable insight can still be gained from the experiments conducted.

Node Architecture

The node architecture presented in this section is an original design proposed, implemented, and tested in order to enable the experimental research of this thesis. The design itself represents a significant development effort and contribution of this research, and it is important to review the details of the organization and operation of this architecture in order to understand its potential benefits for space-based radar applications. Each node of the testbed contains three major blocks: (1) an external memory interface, (2) a network interface to the high-performance interconnect, and (3) one or more processing elements and on-chip memory controller. Beyond simply providing these various interfaces or components, the node architecture must also provide a highly efficient data path between these components, as well as a highly efficient control path across the node to enable tight control of the node's operation. This particular architecture features multiple processing units in each node, including both hardware processors implemented in the reconfigurable fabric and software processors in the form of hard-core embedded PowerPC405s. Given the dual-paradigm nature of each node, the optimal division of tasks between the hardware and software processors is critical to identify in order to maximize system performance. A block diagram illustrating the entire node architecture is shown in Figure 3-3.

It should be noted here that the design is *SDRAM-centric*, meaning that all data transfers occur to or from external SDRAM. With a single bank of local memory for each node, the performance of that SDRAM controller is critical to ultimate system performance. Most processors today share this characteristic, and furthermore the conventional method of providing connectivity to the external memory is through a local bus. Even if the system interconnect is a packet-switched protocol, internal to each node the various components (e.g. functional units, network interface, etc.) are connected over a local bus, such as CoreConnect from IBM or

AMBA Bus [43]. This local bus forces heavyweight bus protocol overheads and time division multiplexing for data transfer between different components of an individual node. An alternate approach to providing node-level connectivity to external memory is to use the programmable fabric of the FPGA to implement a multi-ported memory controller. The multi-port memory controller will provide a dedicated interface for each separate component, thus increasing intra-node concurrency and allowing each port to be optimized for its particular host. Each port of the memory controller has dedicated data FIFOs for buffering read and write data, allowing multiple components to transfer data to/from the controller concurrently, as well as dedicated command FIFOs to provide parallel control paths.

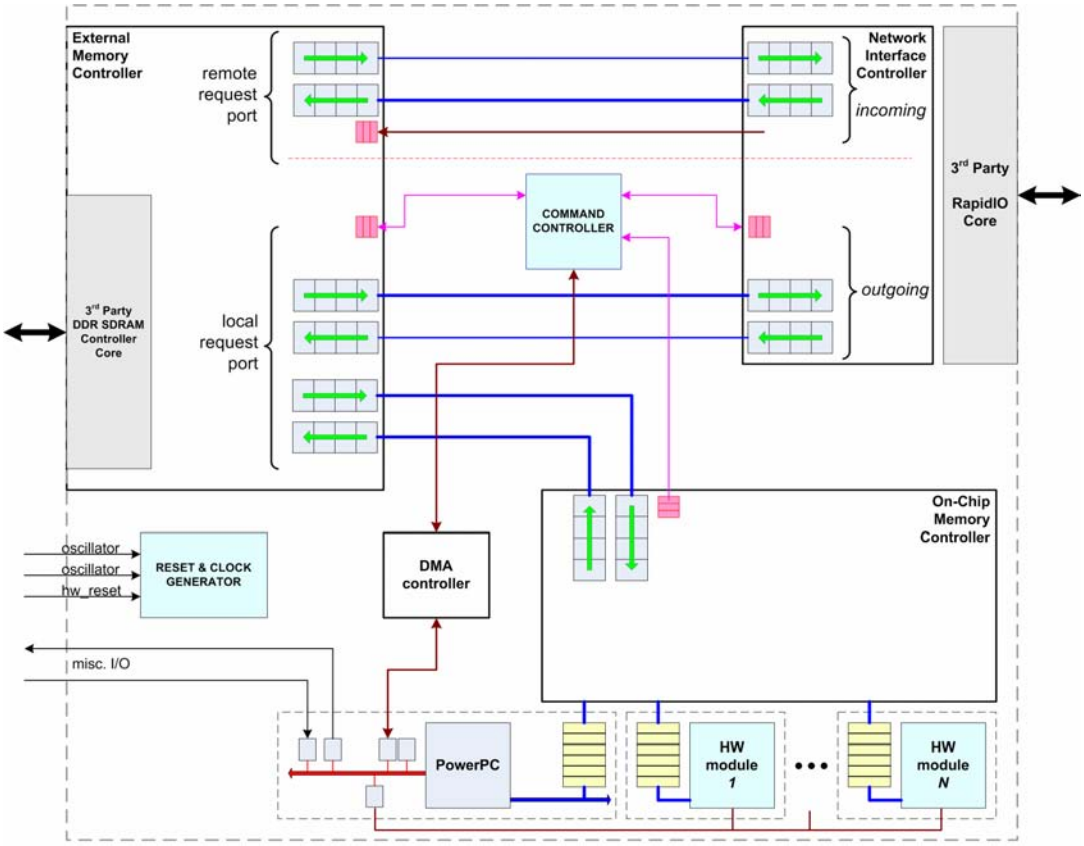


Figure 3-3. Node architecture block diagram.

The control network within each node of the testbed provides the ability to request locally-initiated transfers, control co-processor activity, and handle incoming RapidIO requests

transparently. Locally-initiated data movement is controlled through a DMA engine driven by the PowerPC (shown as two components in Figure 3-3, the *DMA controller* and the *command controller*). Based on the requested source and destination memory addresses, the DMA engine determines if the memory transfer is local or remote, and sends personalized commands to the command FIFO of the external memory controller as well as the command FIFO of the other component, as appropriate for the transfer type. Each component acknowledges the completion of its portion of a data transfer to the DMA engine, providing a feedback path back to the PowerPC to indicate the successful completion of memory transfers. Co-processor activity, by contrast, is controlled by a dedicated control bus directly connecting the PowerPC to the co-processors (shown as the dark red line near the bottom of Figure 3-3). Similar to DMA transfers, processing and configuration commands are submitted to the co-processors and their activity is monitored using this control bus.

External Memory Controller

As the central component to each node in the system, the external memory controller affects both interconnect as well as processor efficiency. As such, it is the most high-performance part of the system, capable of handling traffic from both the interconnect as well as the processors concurrently. The interface to the controller is composed of multiple ports, each dedicated to a single component in the node providing dedicated data paths for every possible direction of transfer. All connections to the external memory controller interfaces are through asynchronous FIFOs, meaning that not only can the external memory controller operate in a different clock domain from the rest of the node, it can also have a different data path width. This logic isolation strategy is a key technique used through the entire node design, in order to enable performance optimization of each section independent of the requirements of other

sections of the design. Figure 3-4 shows a *very* simplified block diagram of the internal logic of the external memory controller.

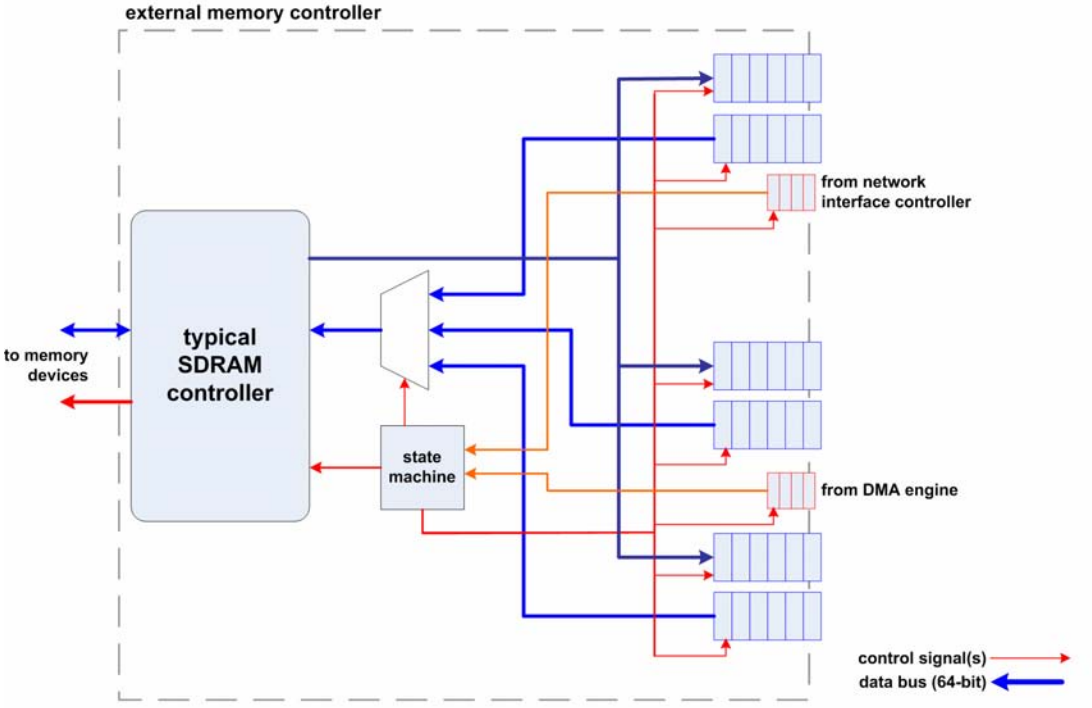


Figure 3-4. External memory controller block diagram.

The external memory controller is based on a single state machine that constantly monitors the two command queues. Notice that in Figure 3-4, there are two command FIFOs leading into the external memory controller; one of these command FIFOs is for locally-requested memory transfers (lower) and the other is for incoming remote transfer requests from the RapidIO fabric (upper). Commands are decoded and executed as soon as they arrive, with a simple arbitration scheme that prevents starvation of either interface to the controller. To further prevent congestion at the external memory controller of each node, the external memory controller data throughput is twice that of either the on-chip memory controller or the RapidIO interface. This over-provisioning of the bandwidth ensures that even if a processor is writing or reading to/from external memory, incoming requests from remote nodes can also be serviced at the full RapidIO link speed without interfering with the processor traffic through the controller.

Network Interface Controller

The network interface controller is a complex component, due to the multiple independent interfaces to the actual RapidIO core. In order to provide true full-duplex functionality, the user interface to the network controller (at the logical layer of RapidIO) must provide four parallel control and data paths, which translates to a total of four independent state machines. This multi-interface requirement is common for all full-duplex communication protocols, and the network interface controller would have a similar architecture even if Ethernet or some other interconnect standard was being used. The four interfaces to the RapidIO logical layer core can be conceptually grouped into two pairs of interfaces, and are labeled as follows: (1a) initiator request port, (1b) initiator response port, (2a) target request port, and (2b) target response port. The two initiator ports handle outgoing requests and incoming responses to locally-initiated remote transfers. The two target ports handle incoming requests and outgoing responses to remotely-initiated remote transfers. A simplified block diagram of the internal architecture of the network interface controller is illustrated in Figure 3-5.

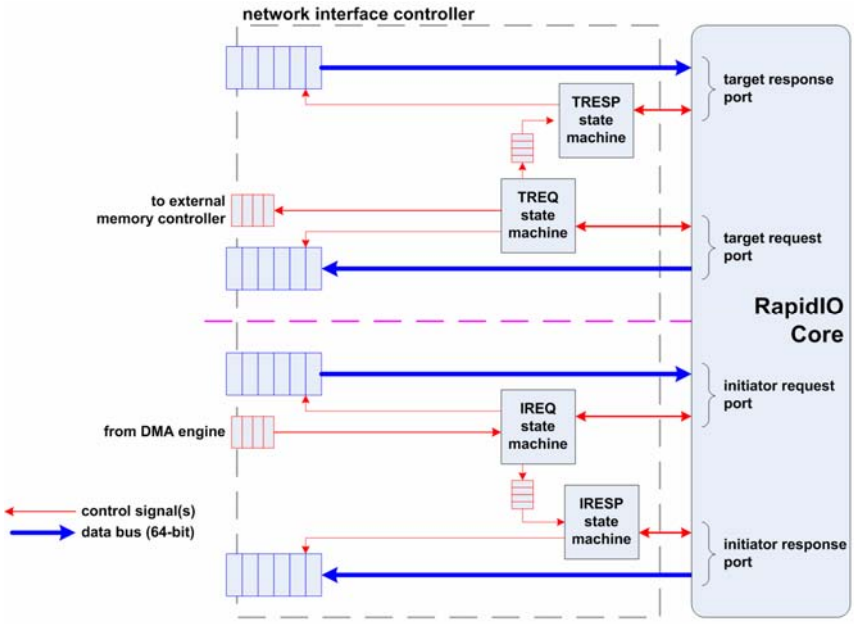


Figure 3-5. Network interface controller block diagram.

The reason for the multi-port requirement in full-duplex communication protocols is based on the idea that even if the physical layer supports full-duplex communication, if the upper-layer logic cannot do the same then a bottleneck would exist at the input to the interconnect fabric, and the full-duplex capability would never be used. For example, imagine if instead of four ports, there were only two, an initiator port and a target port. The initiator port would be responsible for handling outgoing requests *and* incoming responses, and by having the same logic that is performing transmissions need to also receive responses it would prevent the *true* ability to send and receive at the same time. Alternately, consider the case where the two ports are simple transmit and receive ports. In this case, while requests could be transmitted concurrently while receiving responses, a problem arises when remotely-initiated requests come in off the fabric in the middle of a locally-initiated transmission. It should now be clear why there is a need for four independent upper-layer interfaces to any full-duplex communication protocol in order to maximize network efficiency.

The two pairs of ports, the initiator pair and the target pair, are completely isolated from one-another. Within each pair, however, the two interface state machines must exchange some information. Small FIFOs take information from the initiator request and target request state machines, in order to inform the initiator response and target response state machines of the need to receive/send a response. The information contained in these messages is minimal, including request type, size, source ID (which is the destination ID for the response), etc. The initiator request state machine receives messages through the command FIFO from the main DMA engine when the PowerPC requests a remote read or write, and informs the initiator response state machine through the internal message FIFO described above if there will be any responses to expect. The target request state machine, by contrast, sits idle and waits for new requests to

come in from the RapidIO fabric. When a new request comes in, the target state machine informs the target response state machine to prepare to send responses, and also places a command in its command FIFO to the external memory controller that includes the size, address, and type of memory access to perform. In order to operate synchronously with the RapidIO link, as a requirement of the Xilinx RapidIO core all logic in the network interface controller operates at 62.5 MHz, which also happens to be half of the frequency of the external memory controller.

On-Chip Memory Controller

The on-chip memory (OCM) controller is fairly simple, and provides the mechanisms necessary to enable basic read or write transactions between the FIFOs of the main data path and the co-processor SRAM memories. Internal to the OCM controller, the data width is reduced to 32 bits from 64 bits on the SDRAM-side of the data path FIFOs, while the clock frequency remains the same at 125 MHz. The data interface to each co-processor is a simple SRAM port; as such, multiple co-processor interfaces can be logically combined to appear as one SRAM unit, with different address ranges corresponding to the memories physically located in different co-processors.

In the baseline node design, there is no method of transferring data directly from one co-processor to another, nor is there any way to transfer data directly from the co-processor memories to the RapidIO interface controller. There are arguable performance benefits to be gained by enabling such data paths, and these architecture enhancements will be considered later as part of experimentation. However, the addition of such capabilities does not come without cost, the tradeoff in this case being an increase in development effort through a rise in the complexity of the control logic, as well as potentially increasing the challenge of meeting timing constraints by crowding the overall design.

PowerPC and Software API

Overall control of each node's operation is provided by one of the two embedded PowerPC405s in each FPGA. This PowerPC is responsible for initiating data movement throughout the system, as well as controlling co-processor activity. However, the PowerPC is not used for any data computation or manipulation, other than briefly in some corner-turn implementations. The PowerPC405 is relatively low performance, and moreover this particular node design does not give the PowerPC a standard connection to the external memory. The PowerPC processor itself is directly integrated with several peripherals implemented in the reconfigurable fabric of the FPGA. These peripherals include a UART controller, several memory blocks, and general-purpose I/O (GPIO) modules for signal-level integration with user logic. One of the memory blocks is connected to the OCM controller to provide a data path for the PowerPC, and the GPIO modules connect to architecture components such as the DMA engine or the co-processor control busses. The external memory controller could be dedicated to the PowerPC through its standard bus interface, however that would prevent other components in the system from connecting directly to the external memory controller as well (recall the multi-port memory controller is a boasted feature of this node architecture). Instead, the PowerPC is used exclusively for control, and leaves data processing tasks to the co-processor engines.

To support application development and provide a software interface to the mechanisms provided by the node architecture, an API was created containing driver functions that handle all signal-level manipulation necessary to initiate data movement, control co-processors, and perform other miscellaneous actions. The functions of this API provide a simple an intuitive wrapper to the user for use in the application code. Table 3-1 indicates and describes the most important functions in this API, where classes *D, M,* and *P* mean data movement, miscellaneous, and processor control, respectively.

Table 3-1. Software API function descriptions.

Function	Class	Description
dma_blocking	D	Most common data transfer function. Provide source address, destination address, and size. Function will not return until entire transfer has completed.
dma_nonblocking	D	Non-blocking version of same function described above. This function will start the data transfer, and will immediately return while data is still being moved by the system.
dma_ack_blocking	D	Explicit function call necessary to acknowledge DMA transfers initiated with non-blocking calls. This function, once called, will not return until the DMA transfer completes.
dma_ack_nonblocking	D	Non-blocking version of the acknowledgement function. This function will check if the DMA transfer is complete yet, and will return immediately either way with a value indicating the status of the transfer.
barrier	M	Basic synchronization function, necessary to implement parallel applications. This function mimics the MPI function MPI_Barrier(), and ensures that all nodes reach the same point in the application before continuing.
coproc_init	P	Co-processor initialization function. Provide co-processor ID, and any configuration data required. The function will return once the co-processor has been initialized and is ready for processing.
coproc_blocking	P	Most common processor activity function. Provide co-processor ID, and any runtime data required. This function is blocking, and will not return until the co-processor indicates that it has completed processing.
coproc_nonblocking	P	Non-blocking version of the processing function described above. The function will initiate processing on the indicated co-processor, and will immediately return while the co-processor continues to process.
coproc_wait	P	Explicit function necessary to acknowledge co-processor execution runs initiated with the non-blocking function call.

Co-processor Engine Architectures

Each of the co-processor engines was designed to appear identical from the outside, presenting a standardized data and control interface for an arbitrary co-processor. Since all co-processors share a standard interface and control protocol, the common top-level design will be described first before briefly defining the architecture and internal operation of each specific co-processor engine in the following sections. The location of these co-processors in the overall node architecture is shown near the bottom of Figure 3-3, labeled *HW module 1* through *HW*

module N. The co-processor interface consists of a single data path, and a single control path. Data to be processed is written into the co-processor engine through the data port, and processed data is read back out of this port when processing completes. The control port provides the ability to configure the engine, start processing, and monitor engine status. Figure 3-6 illustrates this generic co-processor wrapper at the signal-level.

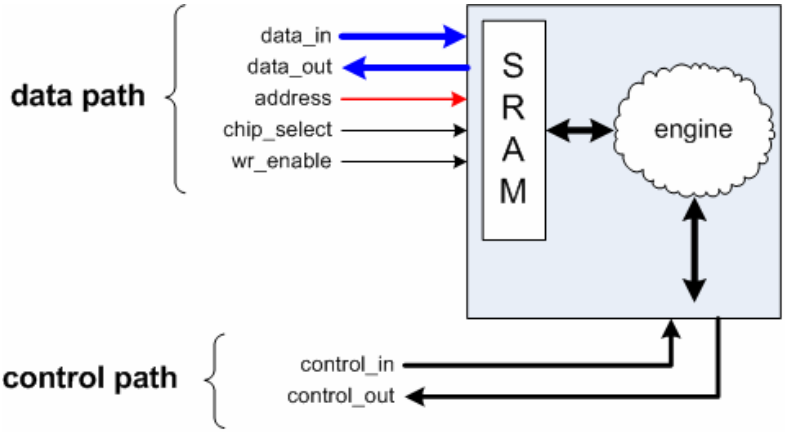


Figure 3-6. Standardized co-processor engine wrapper diagram.

Recall that all co-processor engines contain a small amount of SRAM memory (maximum of 32 KB), and that this memory is the main method of passing data to and from the engine. The single external data port can address this entire internal SRAM address space, through a standard SRAM interface containing a data bus input and output, as well as an address input, a chip select, and a read/write select signal (single-cycle access time). Regardless of what the processing engine is designed to do, data is written-to and read-from its processing memory like a simple SRAM, with certain address ranges defined for various purposes internal to the engine. All internal SRAMs, interchangeably referred to as BlockRAMs or BRAMs, of Xilinx FPGAs are true dual-port memories. This feature is critical to the design of this style of I/O for co-processor engines. With one port of each BRAM dedicated to the external data interface, the other port is used by the actual processing logic.

The control port of each co-processor wrapper is divided into two busses, one input and one output. Each bit or bit field of these control busses has a specific definition, as shown below in Table 3-2. When using multiple co-processing engines in a single FPGA, the control ports are combined as follows:

- The control bus outputs from each co-processor are kept separate, and monitored individually by the main processor (PowerPC)
- All control bus inputs to the co-processors are tied together

Table 3-2. Co-processor wrapper signal definitions.

Signal Name	Dir.	Width	Description
clock	input	1	Clock signal for logic internal to actual processor
reset_n	input	1	Global reset signal for entire design
coproc_id	input	2	Hard-coded co-processor ID; each command received on control bus is compared to this value to see if it is the intended target of the command
data_in	input	32	Data bus into co-processor for raw data
data_out	output	32	Data bus out of co-processor for processed data
address	input	32	Address input to co-processor
chip_select	input	1	Enable command to memory; read or write access determined by *wr_enable*
wr_enable	input	1	Write-enable signal; when chip_select is asserted and wr_enable = '1', a write is issued to the address present on the *address* bus; when *chip_select* is asserted and *wr_enable* = '0', a read is performed from the address specified on the *address* bus
control_in	input	16	Control bus input into co-processor from the PowerPC. The bits of the bus are defined as follows: 0-7: configuration data 8-9: coproc_ID 10: blocking (1)/non-blocking (0) 11: synchronous reset 12: reserved (currently un-used) 13: command enable 14-15: command "01" = *config* "10" = *start* "00", "11" = reserved (un-used)
control_out	output	4	Control bus from co-processor to the PowerPC. The bits of the bus are defined as follows: 0: *busy* 1: *done* 2: *uninitialized* 3: reserved (un-used)

Some of the co-processor engines have run-time configurable parameters in the form of internal registers in the co-processor engine itself. One example of such a configurable parameter is for the CFAR processor, where it needs to be told what the range dimension is of the current data-cubes so it knows how much to process before clearing its shift register for the next processing run. As part of the wrapper standard, all co-processors must be "initialized" or configured at least once before being used for processing. Whether or not the co-processor currently supports configurable parameters, according to the standard upon coming out of reset all co-processors will have their *uninitialized* bit set on the control bus, indicating that they need to be configured. The PowerPC will initialize each co-processor one at a time as part of its startup sequence by submitting *config* commands along with the appropriate *coproc_ID* and *data*. Once initialized, the co-processor will clear its *uninitialized* bit and can be used for processing at any time. The co-processor may also be re-configured with different parameters by executing another *config* command to set the config registers with the new data. This feature of all co-processors enables *virtual reconfiguration,* as proposed by a recent Master's Thesis [24].

To perform data processing with the co-processor, the PowerPC will first perform a DMA transfer of the data to be processed from external memory into the co-processor's address space. The exact memory map of each co-processor varies, and will be defined in the appropriate co-processor section later. Once the data has been transferred into the engine, the PowerPC submits a *start* command to the appropriate engine, and monitors the *busy* and *done* bits of that co-processor's control bus output. Upon observing the *done* bit being asserted, another DMA transfer can be performed to move the processed data out of the engine and back to external storage. It should be noted here that all of the co-processor engines implement double-buffering in order to maximize processing efficiency and minimize idle time.

Pulse Compression Engine

The pulse compression engine design adopted for this research follows that of a commercial IP core [9] by Pentek, Inc. The engine is centered around a pipelined, selectable FFT/IFFT core that provides streaming computation, an important feature for this high-performance design (Figure 3-7). Recall that pulse compression operates along the range dimension, and is composed of an FFT, followed by a point-by-point complex vector multiply with a pre-computed vector (also stored in the co-processor's SRAM), and finally an IFFT of the vector product. To implement this algorithm, the aforementioned FFT core is re-used for the final IFFT as well, in order to minimize the amount of resources required to implement the pulse compression co-processor. The numerical precision used for the pulse compression co-processor is 16-bit fixed point (with 8.8 decimal format) as suggested in Aggarwal's work [24], resulting in 32 bits per complex data element.

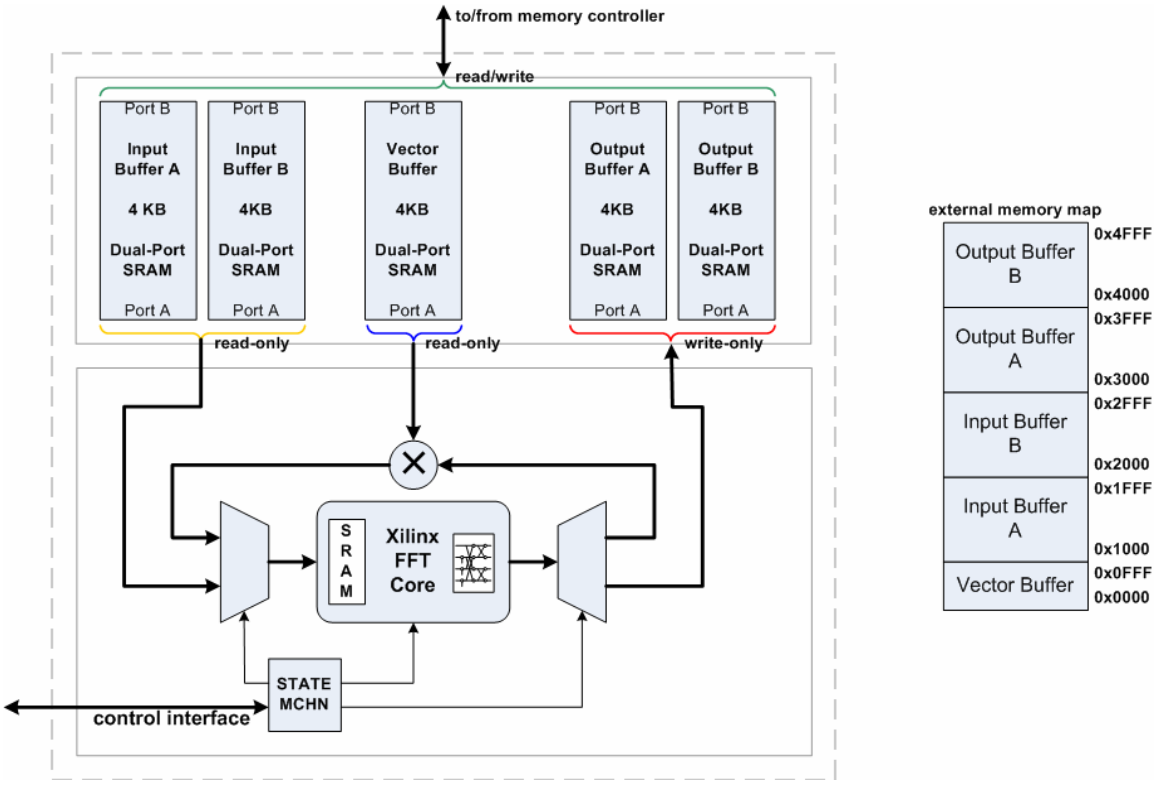


Figure 3-7. Pulse compression co-processor architecture block diagram.

Once given a command to begin processing, the engine will start by reading data out of the specified input buffer directly into the FFT core, through a multiplexer. As soon as possible, the FFT core is triggered to begin processing, and the state machine waits until the core begins reading out the transformed results. At this time, the transformed data exits the FFT core one element at a time, and is sent through a demultiplexer to a complex multiplier. The other input of the multiplier is driven by the output of the vector constant SRAM, which is read in synch with the data output of the FFT core. The complex multiplier output is then routed back to the input of the FFT core, which by this point has been configured to run as an IFFT core. The final IFFT is performed, and the core begins outputting the pulse compression results. These results are routed to the specified output buffer, and the state machine completes the processing run by indicating to the PowerPC through the control interface outputs that the co-processor is finished.

The architecture of the co-processor's SRAM bank is an important topic that warrants discussion, and the pulse compression engine will be used as an example. The pulse compression engine contains a total of five independent buffers or SRAM units (see Figure 3-7): two input buffers, two output buffers, and a single vector constant buffer. Each buffer is 4 KB in size, resulting in a total pulse compression co-processor memory size of 20 KB. The two SRAM features taken advantage of for this architecture are (1) true dual-port interface, and (2) ability to combine smaller SRAMs into one larger logical SRAM unit. The external data/SRAM port to the co-processor has both read and write access to the entire 20 KB of memory, by combining one port of all SRAMs to appear as single logical SRAM unit (according to the memory map shown in Figure 3-7). However, there is no rule to say that the other port of each SRAM unit must be combined in the same manner; therefore, the other port of each SRAM unit is kept independent, to allow concurrent access of all buffers to the internal computational engine.

Doppler Processing Engine

The Doppler processing engine is very similar to the pulse compression co-processor, also based on an FFT core at its heart. Recall that Doppler processing is composed of a complex vector multiplication followed by an FFT. Upon receiving a command to begin processing, the input data will be read out of the specified input buffer concurrently with the pre-computed constant vector from its vector buffer (as indicated below in Figure 3-8), and fed into a complex multiplier unit. The product output of this multiplier is used to drive the input of the FFT core, which computes the transform before unloading the transformed results directly into the output buffer. Like the pulse compression co-processor, the Doppler processor unit is originally designed assuming 16-bit fixed-point precision (32-bits per complex element), with an 8.8 decimal format.

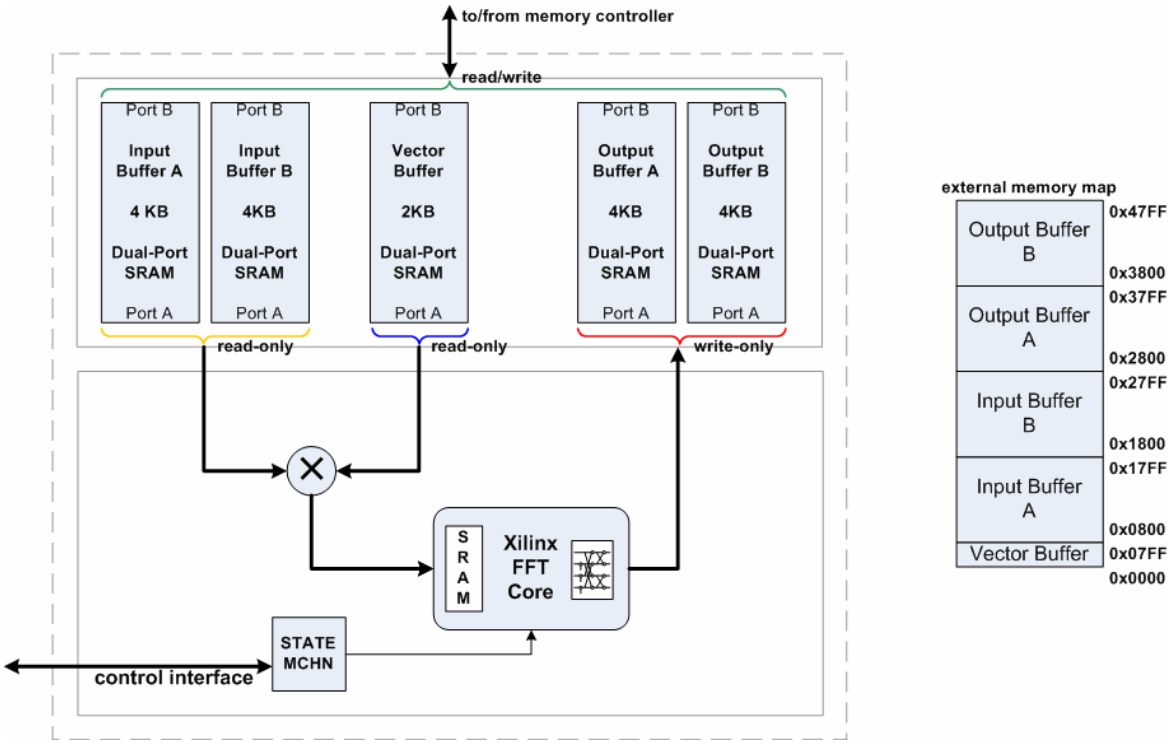


Figure 3-8. Doppler processing co-processor architecture block diagram.

Beamforming Engine

The beamforming co-processor is based on a matrix multiplication core described in [44]. This engine computes all output beams in parallel, by using as many multiply-accumulate (MAC) units as there are output beams. The weight vector provided by the AWC step is stored in a specially-designed SRAM module, that provides as many independent read-only ports to the internal engine as there are MAC units. By doing so, the input data-cube can be read out of the input buffer one element at a time in a steaming fashion (recall data is organized along the channel dimension for beamforming), and after an entire channel dimension has been read out, and entire output beam dimension will have been computed. While the next channel dimension is being read out of the input buffer, the previously-computed beam is registered and stored in the output buffer one element at a time. The complex-format beamformed output is partially magnituded (no square root) before being written to the output buffer as 32-bit real values.

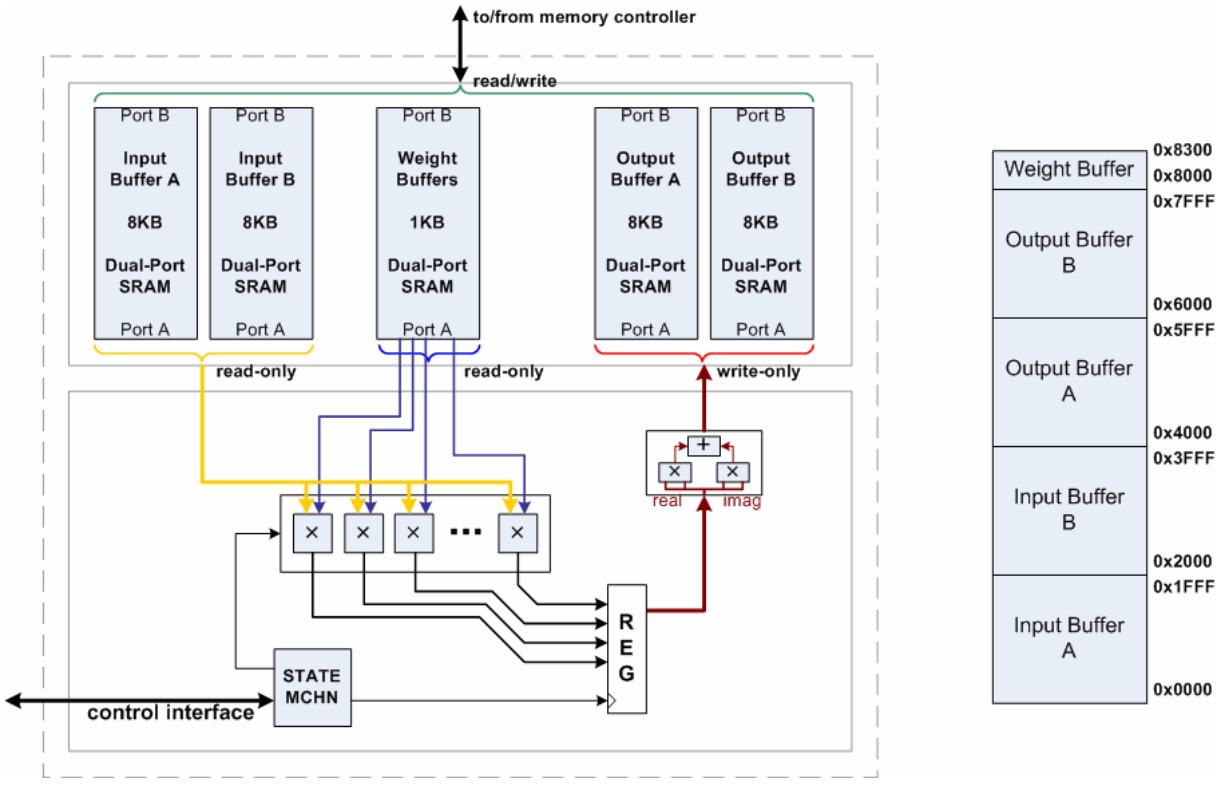


Figure 3-9. Beamforming co-processor architecture block diagram.

Figure 3-9 illustrates the architecture of the beamforming co-processor. Notice that unlike the other co-processor designs, the input buffers and output buffers are different sizes. This asymmetry is due to the reduction of one dimension of the data-cube from channels to beams, the assumption being that the number of beams formed will be less than the number of channels in the original cube. The actual computations performed by this architecture can be better visualized by considering Figure 3.10 below. The weight vector for each beam to be formed is stored in a small, independent buffer, so that the weights for all beams can be read in parallel. The elements of the input data-cube ($B_{c,p,r}$) are read sequentially from the input buffer, and broadcast to all of the MAC units. The other input to each MAC unit is driven by its respective weight vector buffer, and so every C clock cycles (where C = channel dimension length) an entire beam dimension is completely processed. These values will be registered and held for storage, while the next set of beam elements are computed by re-reading the weight vectors while reading the next C element of the input data-cube.

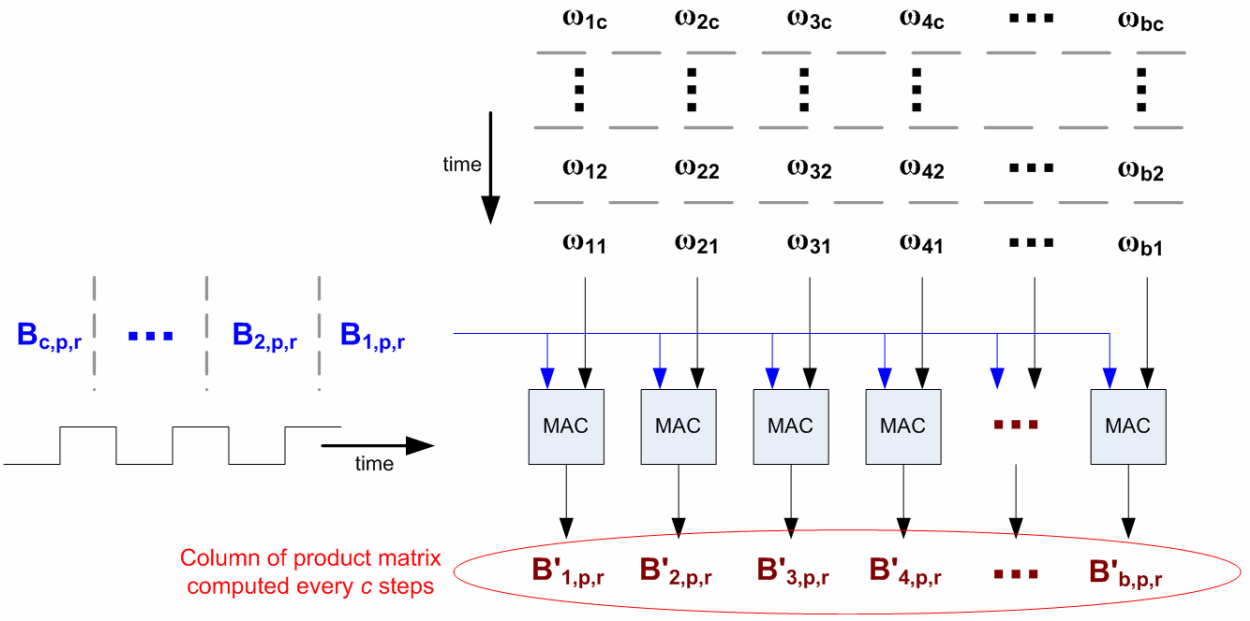


Figure 3-10. Illustration of beamforming computations.

CFAR Engine

The CFAR detection co-processor architecture is based on a design presented in [16]. The major component of the CFAR engine is a shift register, as wide as each data element and as long as the CFAR sliding window length. Once processing begins, data is read out of the appropriate input buffer and fed into this shift register. Several locations along the shift register are tapped and used in the computation of the running local averages as well as for the thresholding against the target cell. Recall that the ultimate goal of CFAR is to determine a list of targets detected from each data-cube [15]. This output requirement implies that the entire data-cube does not need to be outputted from CFAR, and in fact the minimum required information results in a basically negligible output data size. However, to provide a simple method of verification of processing results, the output of this CFAR co-processor is the entire data-cube, with zeros overwriting all elements not deemed to be targets. To provide realistic processing latency, for all performance measurements no output data is read from the processor.

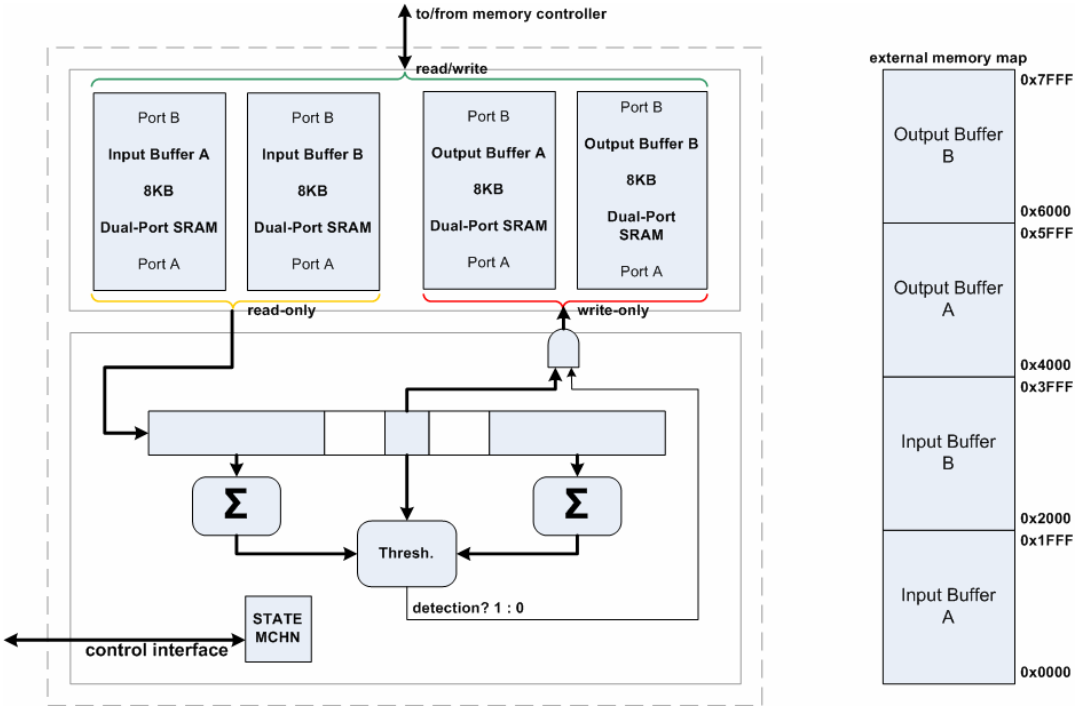


Figure 3-11. CFAR detection co-processor architecture block diagram.

As mentioned back in Chapter 2, CFAR is not only parallel at a coarse-grained level with each Doppler bin being processed independently, but the computations performed for each consecutive data element also contain significant parallelism through partial results reuse of the locally-computed averages [15]. For each averaging window (left and right), almost all data elements to be summed remain the same between two adjacent target cells, with the exception of one data element on each end of the windows. As the CFAR window moves one element over, the oldest element needs to be subtracted from the sum, and the newest element added to the sum. By taking advantage of this property, CFAR detection can be performed $O(n)$, where n represents the length of the range dimension, *independent of the width of the sliding window*.

CHAPTER 4
ENVIRONMENT AND METHODS

This chapter will briefly describe the experimental environment and methods used in the course of this research. The first section will define the interfaces to the testbed available for the user, followed by a detailed description of measurement techniques and experimental procedures in the second section. Finally, the third section will formally define all assumptions, metrics, and parameters relevant to the experiments carried out.

Experimental Environment

The RapidIO testbed is connected to a number of cables and devices, providing a control interface, a method of file transfer, and measurement capabilities. The testbed connects to a workstation through both an RS-232 serial cable, as well as a USB port for configuration through JTAG. Additionally, there is a PC-based logic analyzer that connects to up to 80 pins of the FPGAs, as well as to the user workstation through another USB port for transfer of captured waveforms. The embedded PowerPCs in each FPGA drive the UART ports on the testbed, and the user can monitor and control testbed progress through a HyperTerminal window or other similar UART communication program (only the "master" node's UART port is used, for a single UART connection to the workstation). This HyperTerminal window interface provides the ability to use 'printf()'-style printouts to display application progress as well as assist in debugging to a degree, and also enables file transfer between the user workstation and the RapidIO testbed. In addition to the testbed itself and the application that runs on it, a suite of basic utility C programs was written to run on the workstation for functions such as file creation, file translation to/from the testbed data file format, comparison of testbed results with golden results, etc. Figure 4-1 illustrates the relationship between the user, the workstation, the testbed, and the various cables and connections.

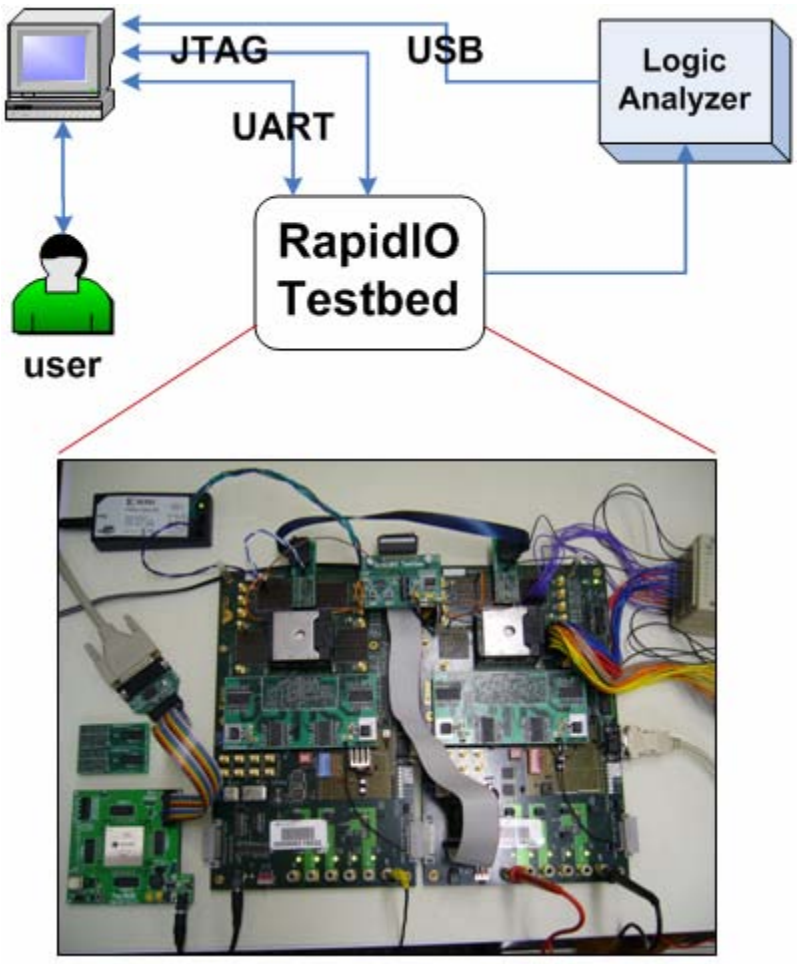


Figure 4-1. Testbed environment and interface.

Measurement Procedure

The main tool for collecting timing and performance measurements is a PC-based logic analyzer from Link Instruments, Inc. The logic analyzer has a maximum sampling frequency of 500 MHz, providing extremely fine-grained timing measurements of down to 2 ns. With up to 80 channels available on the logic analyzer, the desired internal signals can be brought out to un-used pins of the FPGA to provide a window into the internal operation of the design. The only signals that are invisible, or unable to be brought out for capture, are the signals internal to the Xilinx RapidIO core, which are intentionally kept hidden by Xilinx as proprietary information. However, the interfaces to the different RapidIO cores *are* visible, and provide sufficient

information to accurately characterize the operation of the interconnect. Generally, only one logic analyzer capture can be taken per processing run on the testbed, due to the incredibly short execution time of the testbed compared to the speed of the logic analyzer data transfer over USB for the first capture. However, two key advantages of using a PC-based logic analyzer are the convenience afforded by a large viewing area during use, as well as the ability to save captures for archiving of results.

The experimental procedure is fairly simple and straightforward, however it will be explicitly defined here for completeness. Assuming the proper files for programming the FPGAs are on-hand, the first thing the experimenter must do is prepare the input data files to be sent to the testbed using the utility programs that run on the user workstation. These files include data such as pre-computed vector constants, data-cubes for GMTI processing, or other data for debugging and testing, whatever the current experiment calls for. With all files on-hand, the testbed is powered on and the FPGAs are configured with their appropriate bitfiles. The user is greeted with a start-up log and prompt through the HyperTerminal interface, and transfers the data files from the workstation to the testbed as prompted. Once all files have been loaded, the experimenter is prompted one final time to press any key to begin the timed software procedure. Before doing so, the measurement equipment is prepared and set to trigger a capture as needed. The user may then commence the main timed procedure, which typically completes in under a second. The logic analyzer will automatically take its capture and transfer the waveform to the workstation for display, and the testbed will prompt the user to accept the output file transfer containing the processing results. At this point, the user may reset and repeat, or otherwise power down the testbed, and perform any necessary post-analysis of the experimental results.

In addition to the logic analyzer which provides performance measurements, the previously-mentioned suite of utility programs provide the necessary tools for data verification of output files from the testbed. For reference, a parallel software implementation (using MPI) of GMTI was acquired from Northwestern University, described in [10]. This software implementation comes with sample data-cubes, and provides a method of verification of processing results from the RapidIO testbed. Furthermore, this software implementation was used for some simple performance measurements in order to provide some basis on which to compare the raw performance results of the testbed.

Metrics and Parameter Definitions

There are two basic types of operations that make up all activity on the testbed, (1) DMA transfers and (2) processing "runs." DMA transfers involve a start-up handshake between the PowerPC and the hardware DMA controller, the data transfer itself, and an acknowledgement handshake at the completion. A processing "run" refers to the submission of a "process" command to a co-processor engine, and waiting for its completion. Each processing run does not involve a start-up handshake, but does involve a completion-acknowledgement handshake. In order to characterize the performance of the testbed, three primitive metrics are identified that are based on the basic operations described above. All metrics are defined from the perspective of the user application, in order to provide the most meaningful statistics in terms of ultimately-experienced performance, and so the control overhead from handshaking is included. These different latency metrics are defined as shown here:

t_{xfer} = DMA transfer latency

t_{stage} = processing latency of one kernel for an entire data-cube

t_{buff} = processing latency for a single "process" command

Based on these latency definitions (all of which may be measured experimentally), other classical metrics may be derived, for example throughput. Data throughput is defined as follows:

$$throughput_{data} = \frac{N_{bytes}}{t_{xfer}} \tag{4.1}$$

where the number of bytes (N_{bytes}) will be known for a specific transfer. Another pair of important metrics used in this research is processor efficiency, or percentage of time that a processor is busy, and memory efficiency, which illustrates the amount of time that the data path is busy transferring data. Processor efficiency can be an indicator of whether or not the system is capable of fully-utilizing the processor resources that are available, and can help identify performance bottlenecks in a given application on a given architecture. Memory efficiency, by contrast, helps indicate when there is significant idle time in the data path, meaning additional co-processor engines could be kept fed with data without impacting the performance of the co-processor engine(s) currently being used. These efficiency metrics are defined as follows:

$$eff_{memory} = \frac{\alpha \cdot M}{t_{stage} \cdot throughput_{ideal}} \tag{4.2}$$

$$eff_{proc} = \frac{N \cdot t_{buff}}{t_{stage}} \tag{4.3}$$

For memory efficiency, M is a general term representing the size of an entire data-cube (per processor), and α is a scaling factor (usually 1 or 2) that addresses the fact that, for any given stage, the entire data-cube might be sent *and* received by the co-processor engine. Some stages, for example beamforming or CFAR, either reduce or completely consume the data, and thus α for those cases would be less than two. Basically, the numerator of the memory efficiency metric represents how much data *is* transferred during the time it took to process the cube, where

the denominator represents how much data *could have* been transferred in that same amount of time. For processor efficiency, N is a general term for the number of "process" commands that are required to process an entire data-cube. Recall that the t_{stage} term will include overheads associated with repeated process and acknowledge commands being submitted to the co-processor, as well as overhead associated with handling data transfers in the midst of the processing runs. Any efficiency value less than 100% will indicate a processor that has to sit idle while control handshakes and data transfers are occurring.

In addition to these metrics, there are a handful of parameters that are assumed for each experiment in this thesis, unless otherwise noted for a specific experiment. These parameters (defined in Table 4-1) include the data-cube size, the system clock frequencies, memory sizes, as well as the standard numerical format for all co-processors.

Table 4-1. Experimental parameter definitions.

Parameter	Value	Description
Ranges	1024	Range dimension of raw input data-cube
Pulses	128	Pulse dimension of raw input data-cube
Channels	16	Channel dimension of raw input data-cube
Beams	6	Beam dimension of data-cube after beamforming
Processor Frequency	100 MHz	PowerPC/Co-processor clock frequency
Memory Frequency	125 MHz	Main FPGA clock frequency, including SDRAM and data path
RapidIO Frequency	250 MHz	RapidIO link speed
Co-processor SRAM size	32 KB	Maximum SRAM internal to any one co-processor
FIFO size (each)	8 KB	Size of FIFOs to/from SDRAM
Numerical Format	s.7.8	Signed, 16-bit fixed-point with 8 fraction bits

CHAPTER 5
RESULTS

A total of five formal experiments were conducted, complemented by miscellaneous observations of interest. The five experiments included: (1) basic performance analysis of local and remote memory transfers, (2) co-processor performance analysis, (3) data verification, (4) a corner-turn study, and (5) analytical performance modeling and projection. The following sections present the results of each experiment, and will provide analysis of those results.

Experiment 1

The first experiment provided an initial look at the performance capability of the RapidIO testbed for moving data. This experiment performed simple memory transfers of varying types and sizes, and measured the completion latency of each transfer. These latencies were used to determine the actual throughput achieved for each type of transfer, and also illustrated the sensitivity of the architecture to transfer size by observing how quickly the throughput dropped off for smaller operations. Local transfers addressed data movement between the external SDRAM and the co-processor memories, and remote transfers covered movement of data from one node's external SDRAM to another node's external SDRAM over the RapidIO fabric.

Since the internal co-processor memories were size-constrained to be a maximum of 32 KB, the range of transaction sizes used for local transfers was between 64 B to 32 KB. Only two transfer types exist for local memory accesses, reads and writes. Remote transfers could be of any size, so transaction sizes from 64 B to 1 MB were measured experimentally for RapidIO transfers. The RapidIO logical I/O logical layer provides four main types of requests: (1) NREAD, for all remote read requests (includes a response with data), (2) NWRITE_R for write operations requesting a response to confirm the completion of the operation, (3) NWRITE for write operations without a response, and (4) SWRITE, a streaming write request with less packet

header information for slightly higher packet efficiency (also a response-less request type). All four RapidIO request types are measured experimentally for the full range of transfer sizes, in order to characterize the RapidIO testbed's remote memory access performance.

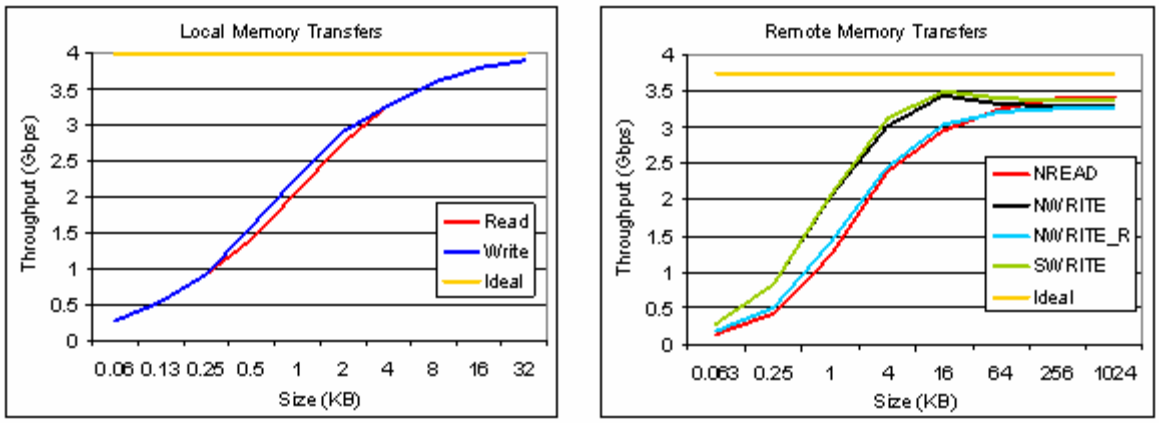


Figure 5-1. Baseline throughput performance results.

First, consider the local memory transfer throughputs; even though the SDRAM interface operates at a theoretical maximum throughput of 8 Gbps (64-bit @ 125 MHz), the on-chip memories only operate at 4 Gbps (32-bit @ 125 MHz), and thus the transfer throughputs are capped at 4 Gbps. Recall, the over-provisioned SDRAM throughput is provided intentionally, so that local memory transfers will not interfere with RapidIO transfers. The overheads associated with DMA transfer startup and acknowledgement handshakes cause the drop-off in throughput for very small transfers, but it can be seen that at least half of the available throughput is achieved for modest transfer sizes of 1 KB. The transfer size at which half of the available throughput is achieved for a given interconnect or communication path is referred to as the half-power point, and is often treated as a metric for comparison between different communication protocols in order to rate the overall efficiency of the connection for realistic transfer sizes.

For remote memory transfers, as one might expect the response-less transaction types (NWRITE, SWRITE) achieve better throughput for smaller transfer sizes. The higher

throughput is a result of lower transfer latency, as the source endpoint does not have to wait for the entire transfer to be received by the destination. As soon as all of the data has been passed into the logical layer of the source endpoint, the control logic is considered done and can indicate to the DMA engine that the transfer is complete. These transaction types take advantage of RapidIO's guaranteed delivery to ensure that the transfers will complete without forcing the source endpoint to sit and wait for a completion acknowledgement. The only *potential* drawback to these transaction types is that the source endpoint cannot be sure of exactly when the transfer really completes, but most of the time that does not matter. The "hump" seen around 16 KB in NWRITE/SWRITE throughputs is due to the various FIFOs/buffers in the communication path filling up, and causing the transfer to be capped by the actual RapidIO throughput. Before that point, the transfer is occurring at the speed of SDRAM, however once the buffers fill up the SDRAM must throttle itself in order to not overrun the buffers. Transfer types that require a response do not experience this same "hump" in throughput, since for all transfer sizes they must wait for the transfer to traverse the entire interconnect before receiving the response. As transfer sizes grow, all transfer types settle to the maximum sustainable throughput.

One interesting observation is that for very large transfer sizes (> 256KB), the throughput achieved by NREAD transactions overtakes the throughput of other transfer types, including the lower-overhead SWRITE type. The reason is due to implementation-specific overheads, most likely differences in the state transitions (i.e. numbers of states) involved in the state machines that control the RapidIO link, where even a single additional state can result in an observable decrease in actual throughput. The conclusions to be taken from these results are that for smaller transfers, unless an acknowledgement is required for some reason, the NWRITE or SWRITE transaction types are desirable to minimize the effective latency of the transfer. However, when

moving a large block of data, it would be better to have the receiving node initiate the transfer with an NREAD transaction as opposed to having the sending node initiate the transfer with a write, once again in order to maximize the performance achieved for that transfer since NREADs achieve the highest throughput for large transfer sizes. It will be shown in the coming experiments that this particular case-study application does not move large blocks of data during the course of data-cube processing, and as such SWRITE transaction types are used for all remote transfers unless otherwise noted.

Experiment 2

This experiment investigated the individual performance of each co-processor engine. All processing for this experiment was done on a single node, in order to focus on the internal operation of each node. Two latencies were measured for each co-processor: (1) t_{buff}, the amount of time necessary to completely process a single buffer, and (2) t_{stage}, the amount of time necessary to completely process an entire data-cube. The buffer latency t_{buff} does not include any data movement to/from external memory, and only considers the control overhead associated with starting and acknowledging the co-processor operation. Thus, this metric provides the raw performance afforded by a given co-processor design, without any dependence on the memory hierarchy performance. The data-cube processing latency t_{stage} includes all data movement as well as processing overheads, and is used in conjunction with the buffer latency results to derive the processor efficiency for each co-processor. Figure 5-2 below summarizes the basic processing latencies of each of the co-processor engines; for the t_{stage} results, both double-buffered (DB) and non-double-buffered (NDB) results are shown. For all remaining experiments, double-buffered processing is assumed; the non-double-buffered results are shown simply to provide a quantitative look at exactly what the benefit is of enabling double-buffing.

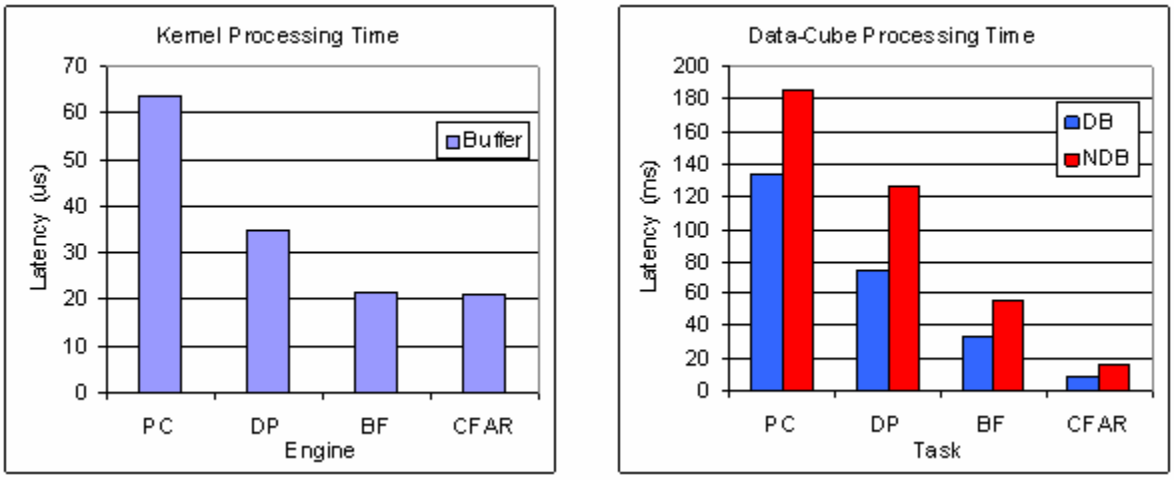


Figure 5-2. Single-node co-processor processing performance.

As shown above in Figure 5-2, all of the co-processors complete the processing of a single buffer of data in the microsecond range, with pulse compression taking the longest. Both beamforming and CFAR have the same input buffer size and operate $O(n)$ as implemented in this thesis, so t_{buff} for both of those co-processors is the same, as expected. The increase of execution time for pulse compression and Doppler processing relative to the other stages remains the same for both t_{buff} and t_{stage}, which suggests that the performance of those two co-processors is bound by computation time. However, consider the results of t_{buff} and t_{stage} for beamforming and CFAR detection; while t_{buff} is the same for both co-processors, for some reason t_{stage} for beamforming is significantly longer than that of CFAR. The reason for this disparity in t_{stage} is rooted in the data transfer requirements associated with processing an entire data-cube. Recall that for CFAR detection, no output is written back to memory since all data is consumed in this stage. However, if both processors are computation-bound, that would not make a difference and t_{stage} should also be the same between those two co-processors. To more clearly identify where the bottleneck is, consider Figure 5-3.

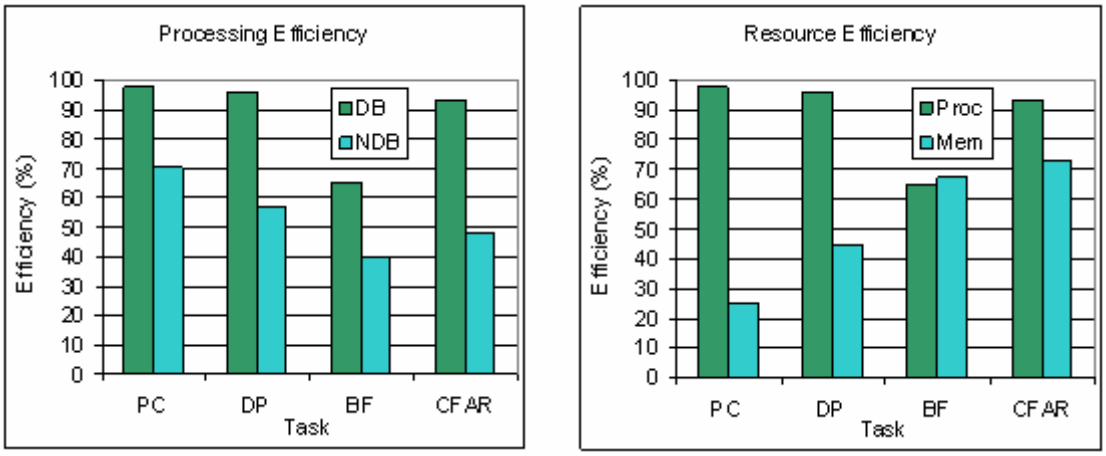


Figure 5-3. Processing and memory efficiency for the different stages of GMTI.

The chart to the left of Figure 5-3 shows processing efficiency, again with both double-buffered and non-double-buffered execution. The NDB results are simply shown to quantitatively illustrate the benefit of double-buffering from a utilization perspective. As predicted above, both pulse compression and Doppler processing are computation-bound, meaning that the co-processor engines can be kept busy nearly 100% of the time during processing of an entire data-cube as data transfer latencies can be completely hidden behind buffer processing latencies. Without double-buffering, however, no co-processor engine can be kept busy all of the time. The chart to the right in Figure 5-3 shows both processing efficiency (double-buffered only) as well as memory efficiency, which can tell us even more about the operation of the node during data-cube processing. Again, as predicted above, it can be seen that while the performance of the CFAR engine is bound by computation, the beamforming engine is not, and even when double-buffered it must sit idle about 35% of the time waiting on data transfers. This inefficiency partially explains the disparity in t_{stage} between beamforming and CFAR, shown in Figure 5-2, where beamforming takes longer to completely process an entire data-cube than CFAR. Furthermore, recall the data reduction that occurs as a result of beamforming; to process an entire data-cube, beamforming must process more data than CFAR.

Even though the input buffers of each co-processor engine are the same size, beamforming must go through more buffers of data than CFAR. The combination of more data and lower efficiency explains why beamforming takes longer than CFAR when looking at full data-cube processing time.

Another important observation from the efficiency results is the memory efficiency of the different co-processor engines. Low memory efficiency implies that the data path is sitting idle during the course of processing, since processing a single buffer takes longer than it takes to read and write one buffer of data. Specifically, notice that pulse compression has a very low memory efficiency of about 25%. What this low data path utilization tells us is that multiple pulse compression co-processors could be instantiated in each node of the testbed, and all of them could be kept fed with data. Exactly how many co-processor engines could be instantiated can be determined by multiplying the single-engine memory efficiency until it reaches about 75% (a realistic upper-limit considering control overheads, based on the I/O-bound beamforming engine results). For future experiments, two pulse-compression engines per node can be safely assumed. By contrast, all other co-processor engines already use at least ~50% of the memory bandwidth, and as such only one co-processor engine could be supported per node. This restriction reveals an important conclusion that it will not be worthwhile to instantiate all four co-processor engines in each node. Instead, each node of the system should be dedicated to performing a single stage of GMTI, which results in a pipelined parallel decomposition. If resource utilization is not a concern, then all four co-processor engines could be instantiated in each node of the testbed, with only one being used at a time. Doing so would enable data-parallel decomposition of the algorithm across the system, but again at the cost of having most of the co-processor engines sit idle the majority of the time.

Finally, to establish a frame of reference for how fast these co-processor designs are compared to other more conventional technology, a performance comparison is offered between the RapidIO testbed and the Northwestern University software implementation executed sequentially on a Linux workstation. Figure 5-4 below shows the result of this performance comparison. To defend the validity of this comparison, even though the hardware being compared is not apples-to-apples, the assumption for both cases is that the entire data-cube to be processed resides completely in main memory (external SDRAM) both before and after processing. Since the capability of a given node to move data between external memory and the internal processing element(s) is a critical facet of any realistic node architecture, as long as both systems start and end with data in analogous locations, the comparison is in fact valid and covers more than just pure processing performance. The Linux workstation used in this comparison is a 2.4 GHz Xeon processor, with DDR SDRAM operating at a theoretical maximum of 17 Gbps. The next experiment will provide a comparison of the output data between these two implementations, to further defend the validity of the HW-SW comparison by showing that the same (or approximately the same) results are achieved.

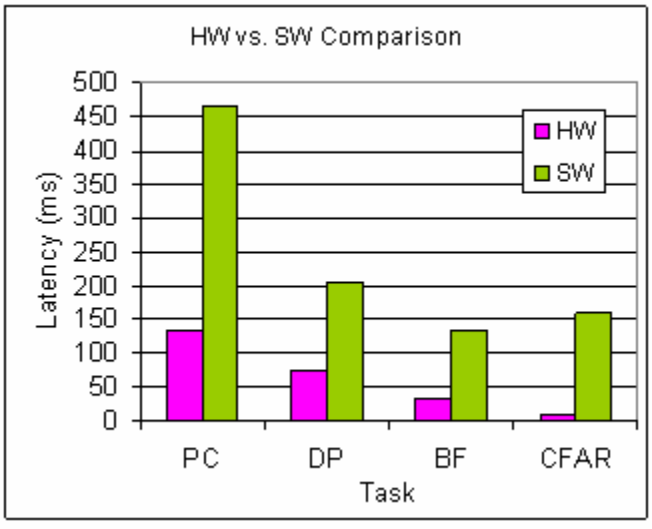


Figure 5-4. Performance comparison between RapidIO testbed and Linux workstation.

The results shown in Figure 5-4 are significant, since the RapidIO testbed co-processors are running at 100 MHz compared to the workstation's 2.4 GHz. The maximum clock frequency in the overall FPGA node design is only 125 MHz, aside from the physical layer of the RapidIO endpoint which runs at 250 MHz (however, recall that during any given stage of processing, no inter-processor communication is occurring). Despite the relatively low frequency design, impressive sustained performance is achieved and demonstrated. Processors operating in space are typically unable to achieve frequencies in the multi-GHz range, and so it is important to show that low-frequency hardware processors are still able to match or exceed the performance of conventional processors with highly efficient architectures.

Experiment 3

To establish confidence in the architectures presented in this thesis research, the processing results of the co-processor engines were validated against reference data, some of which was obtained from researchers at Northwestern University [10]. Each of the co-processing engines was provided with a known set of data, processed the entire set, and unloaded the processed results in the form of a file returned to the user workstation. With the exception of pulse compression and Doppler processing, a software implementation of GMTI was the source of the reference processing results, and employs a double-precision floating point numerical format for maximum accuracy. For the pulse compression and Doppler processing cores, it turns out that the 16-bit fixed-point precision with 8 fractional bits is not nearly enough precision to maintain accuracy, and to illustrate this deficiency a more simplified data set is provided for processing.

First, consider the output of the Doppler processing engine. Since pulse compression and Doppler processing contain nearly identical operations (pulse compression simply contains one extra FFT), only verification of Doppler processing results are shown. Pulse compression is guaranteed to have worse error than Doppler processing, so if 16-bit fixed point with 8 fractional

bits is shown to result in intolerable error in the output of Doppler processing, there is no need to test the pulse compression engine. As mentioned above, the reference data-cubes from the NWU implementation of GMTI are not used for this particular kernel. The data contained in the reference data-cubes is very dynamic, and in most cases the output obtained from running those data sets through the testbed implementation exhibits significant overflow and underflow, resulting in vast differences between golden results and testbed results. In order to instill confidence that this kernel implementation is indeed performing the correct operations in spite of round-off errors, a simplified vector is provided to the testbed implementation, as well as a MATLAB implementation of the Doppler processing kernel.

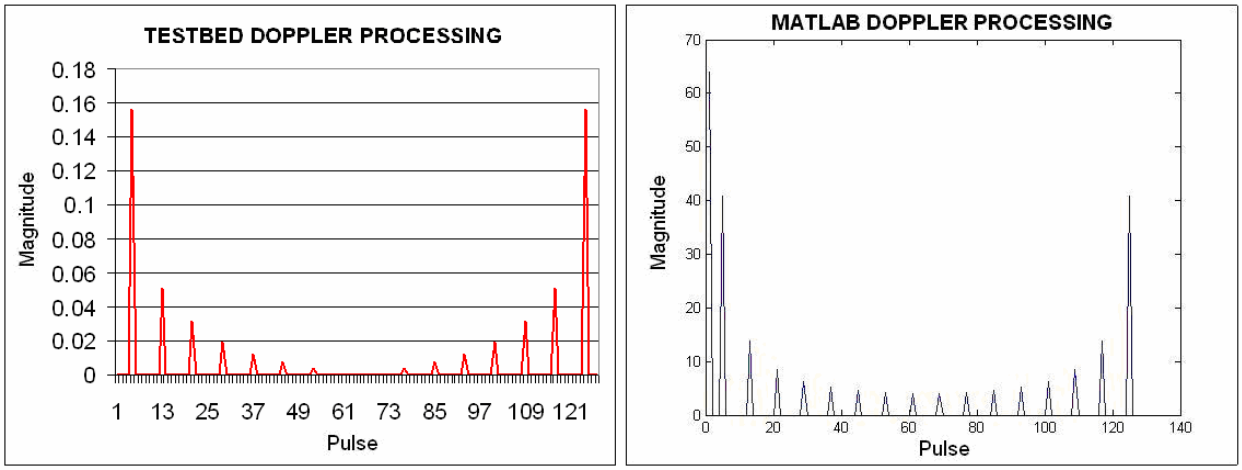


Figure 5-5. Doppler processing output comparison.

Figure 5-5 shows both "golden" (i.e. reference) results alongside the actual testbed output. Notice the underflow that occurs in the middle of the testbed output, where the signal is suppressed to zero. Furthermore, in order to attempt to avoid overflow, an overly-aggressive scaling was employed in the testbed implementation that resulted in all numbers being scaled to << 1. The result of this scaling is that the majority of the available bits in the 16-bit fixed-point number are not used, which is extremely wasteful for such limited precision. More time could have been spent tuning the scaling of the FFT, however it should be emphasized here that the

goal of this experiment was simply to instill confidence that the correct operations are being performed, not to tune the accuracy of each kernel implementation. Despite the obvious design flaws in terms of precision and accuracy, the performance of the kernel implementations are completely independent of the selected precision and scaling, and by inspection the correct-ness of the kernel implementation on the testbed can be seen.

Next, consider the verification of the beamforming co-processor. For this kernel, the actual data from the NWU testbed was fed to the testbed, and the outputs show significantly less error than the Doppler processing and pulse compression kernels. Figure 5-6 below presents a small window of the overall data-cube after beamforming, specifically a 6×7 grid of data points from one particular output beam. The window was chosen to provide a "typical" picture of the output, as showing an entire 1024×128 beam would be difficult to visually inspect.

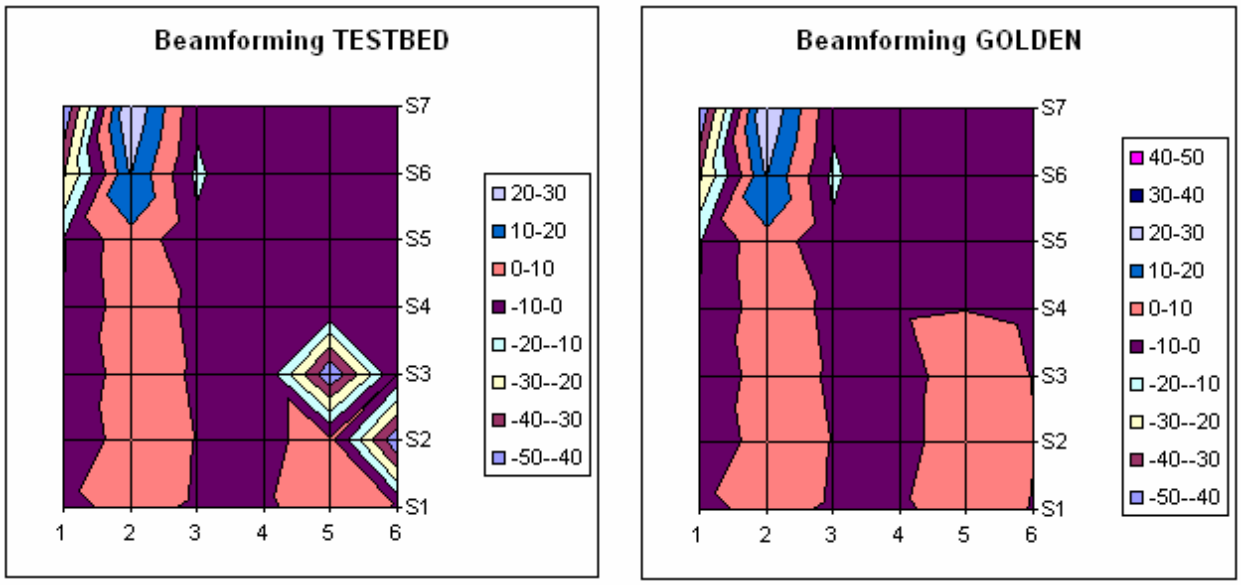


Figure 5-6. Beamforming output comparison.

Aside from two points of underflow, (5, S3) and (6, S2), the outputs are nearly identical. Again, a detailed analysis of the exact error obtained (e.g. mean-square error analysis) is not provided, as fine-grained tuning of kernel accuracy is not the goal of this experiment. In fact, the

two points of underflow shown in the left chart are a result of arithmetic logic errors, since the real data points should be close to zero, as shown in the golden results in the right chart. The actual data values received from the testbed were near or equal to -128, but the scale of the charts was truncated at -50 in order to provide a sufficient color scale to compare the rest of the window. The immediate jump from 0 to -128 suggests a logic error as opposed to typical round-off error, considering the 16-bit fixed-point format with 8 fractional bits and signed-magnitude format. Similar to Doppler processing, correcting this logic error would not affect the performance of the beamforming core, which is the real focus of this research. The results shown above in Figure 5-6 are sufficient to instill confidence that the beamforming core is performing the operations that are expected.

Finally, consider the CFAR detection kernel results. This kernel implementation does not suffer from any minor arithmetic design flaws, and the results that are shown in Figure 5-7 best illustrate why 16-bit fixed-point precision is simply not sufficient for GMTI processing. The radar data from the NWU implementation is used to verify the CFAR engine, with a full 1024×128 beam of the input data shown in the top chart. As can be seen, the range of values goes from very small near the left side of the beam, all the way to very large to the right side of the beam. The middle chart shows the targets that are actually in the data, and the chart on the bottom shows the targets detected by the testbed implementation. By visual inspection, when the input data is small, there are many false-positives (targets reported that really are not targets); when the input data is large, there are many false-negatives (targets not detected). However, in the middle of the beam where the input data values are in the "comfort zone", the testbed implementation is very accurate.

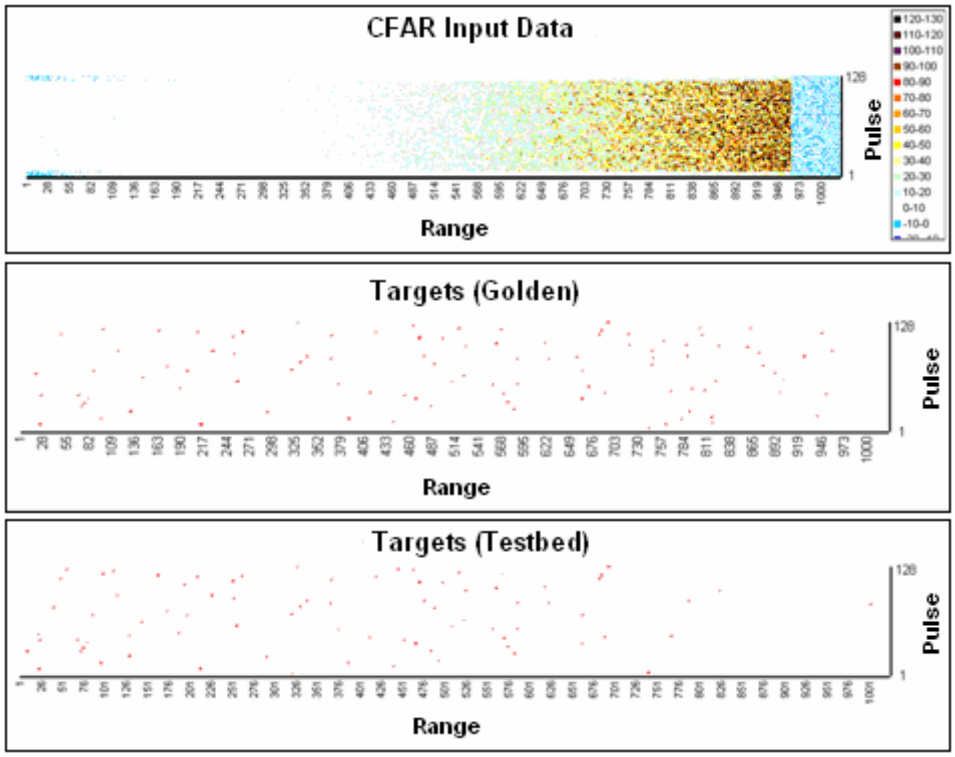


Figure 5-7. CFAR detection output comparison.

One final point to make about the results observed in the above verification experiments is that aside from beamforming (possibly), when tested independently each kernel implementation exhibits intolerable overflow and underflow in the processing output. In a real implementation, these kernels are chained together, and thus any round-off or over/underflow errors would propagate through the entire application, resulting in even worse target detections by the end. However, all hope is not lost for an FPGA implementation of GMTI, as the results simply suggest that more precision is required if fixed-point format is desired, or alternately floating-point could be employed to avoid the over/underflow problem all-together. However, doing so would result in a doubling of the data set size due to the use of more bytes per element, and thus the memory throughput would need to be doubled in order to maintain the real-time deadline.

To defend the accuracy of the performance results presented in other experiments, consider the following upgrades to the architecture that could be implemented in order to provide this

increased throughput. Upgrading the external memory from standard SDRAM to DDR SDRAM would double the effective throughput of the external memory controller from 8 Gbps to 16 Gbps. Also, increasing the OCM data path width from 32 bits to 64 bits would provide 8 Gbps to on-chip memory as opposed to the current 4 Gbps. These two enhancements would provide the necessary data throughput throughout the system to sustain the performance reported in other experiments when using either 32-bit fixed-point, or single-precision floating point. While floating point arithmetic cores do typically require several clock cycles per operation, they are also pipelined, meaning once again the reported throughputs could be sustained. The only cost would be a nominal cost of a few clock cycles to fill the pipelines for each co-processor engine, which translates to an additional latency on the order of a few hundred nanoseconds or so (depending on the particular arithmetic core latency). Such a small increase in processing latency is negligible.

Experiment 4

The fourth experiment addressed the final kernel of GMTI, the corner-turn, which performs a re-organization of the distributed data-cube among the nodes in the system in preparation for the following processing stage. The corner-turn operation is very critical to system performance, as well as system scalability as it is the parallel efficiency of the corner-turns in GMTI that determine the upper-bound on the number of useful processing nodes for a given system architecture. Recall that the only inter-processor communication in GMTI is the corner-turns that occur between processing stages. Since the data-cube of GMTI is non-square (or non-cubic), the dimension that the data-cube is originally decomposed along and the target decomposition may have an effect on the performance of the corner-turn, as it affects the individual transfer sizes required to implement that specific corner-turn. For this experiment,

two different decomposition strategies of the data-cube are implemented, and the corner-turn operation latency of each decomposition strategy is measured and compared.

Figure 5-8 illustrates these two different decompositions, with the red line indicating the dimension along which processing is to be performed both before and after the corner-turn. There are a total of three corner-turns in the implementation of GMTI used for this thesis research (one after pulse compression, one after Doppler processing, and the final one after beamforming), and the data-cube is distributed across two nodes in the experimental testbed. Due to the limited system size, it is impossible to implement the two-dimensional decomposition strategy discussed in Chapter 2, and the cube is only decomposed along a single dimension for each stage.

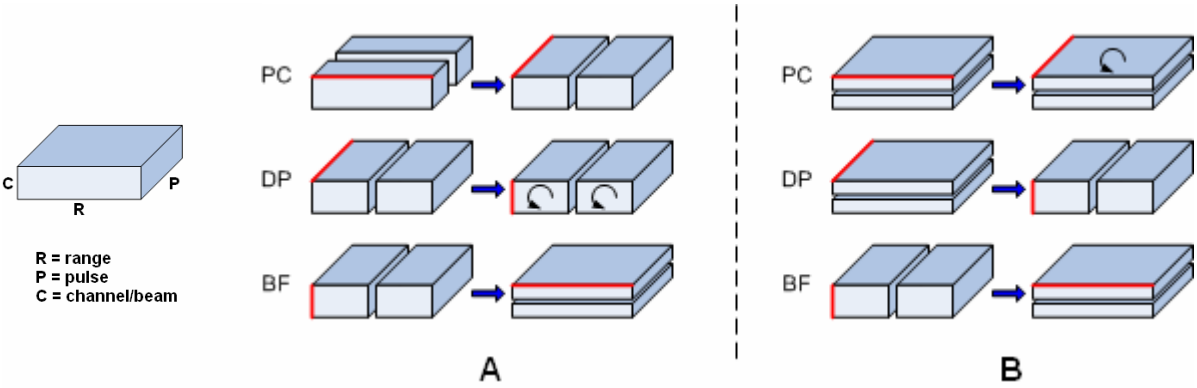


Figure 5-8. Data-cube dimensional orientation, A) primary decomposition strategy and B) secondary decomposition strategy.

It should be noted here that other decompositions and orders of operations are possible, but in order to limit the scope of this experiment only two were selected in order to provide a basic illustration of the sensitivity of performance to decomposition strategy. To avoid confusion, note that there are a total of three corner-turns for each "type" defined above, corresponding to the corner-turns that follow each stage of GMTI. Figure 5-9 presents the corner-turn operation latencies that were measured for both type A and type B decompositions.

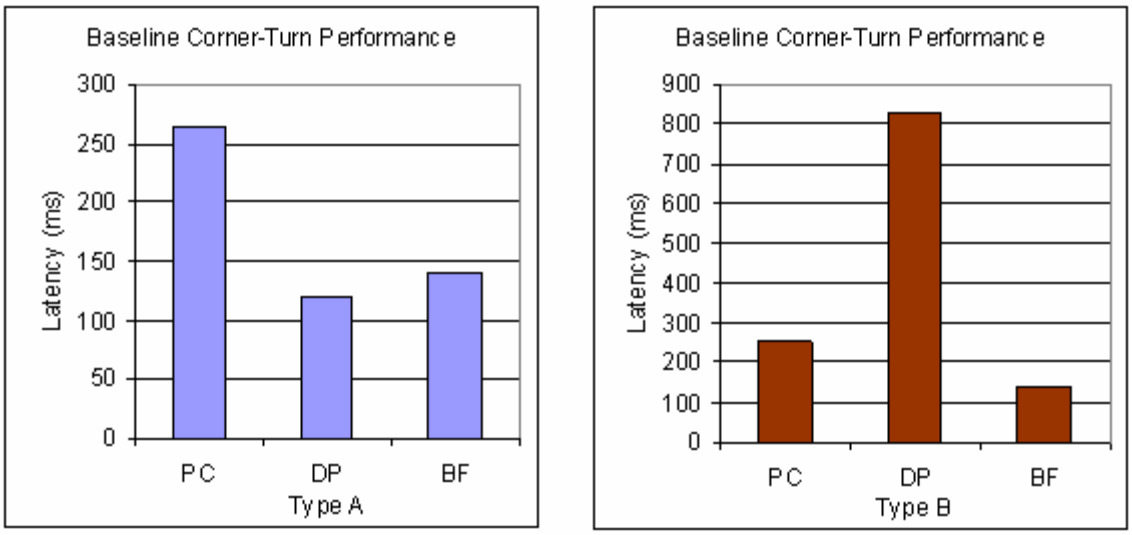


Figure 5-9. Data-cube performance results.

Notice the difference in y-axis scale in each of the two charts in Figure 5-9. As can be seen from the illustrations in Figure 5-8, the final corner-turn of both type A and B has the same starting and ending decomposition. As such, one would expect the performance of that final corner-turn to be the same for both cases, and the experimental results confirm that expectation. The performance of the first corner-turn in both type A and type B are also seen to be similar, however this result is simply a coincidence. The main difference between the two decomposition strategies is therefore the corner-turn that follows Doppler processing, where the operation latency is significantly higher in the type B decomposition than it is for the type A decomposition. For type A, because of the way the data-cube is decomposed before and after Doppler processing, there is no inter-processor communication required, and each node can independently perform a simple local transpose of its portion of the data-cube. However, for type B not only do the nodes need to exchange data, it just so happens that half of the individual local memory accesses are only 32 bytes, the smallest transfer size of all corner-turns considered in this experiment. Also, since such a small amount of data is being read for each local DMA transfer, more DMA transfers must be performed than in the other cases in order to move the

entire data-cube. Since smaller transfer sizes achieve lower throughput efficiency, the significantly-increased operation latency of the second corner-turn for type B decompositions can be explained by this poor dimensional decomposition strategy.

One final observation to discuss about the corner-turn operations as they are implemented in this thesis is the fact that the low-performance PowerPC is the root cause of the inefficiency experienced when performing corner-turns. Due to the high-overhead handshaking that occurs before and after each individual DMA transfer, and the high number of DMA transfers that occur for each corner-turn (as well as relatively small transfer sizes, typically from 32-512 bytes per DMA), almost all of the operation latency is caused by control overhead in the node architecture. With a higher-performance control network, or otherwise hardware-assisted corner-turns, the operation latency could be dramatically improved. The final experiment performed considers such an enhancement to the baseline node architecture, as well as other enhancements to improve the overall GMTI application latency.

Experiment 5

The fifth and final experiment performed did not involve any operation of the testbed. Due to the highly deterministic nature of the node architectures, analytical expressions could be derived fairly easily to accurately model the performance of basic operations. Using these analytical models, the performance of more complex operations was predicted in order to enable performance projection beyond the capabilities of the current RapidIO testbed. It should be noted that the analytical modeling in this section is not presented as a generic methodology to model arbitrary architectures; instead, this analytical modeling approach applies only for performance prediction of this particular testbed design. This section provide both validation results as well as performance projections for GMTI running on an architecture that features enhancements over the baseline design presented in Chapter 3.

The two main operations modeled in this section are: (1) DMA transfers of N bytes, D_N, and (2) co-processor operation latencies, P_{buff}. The GMTI case-study application implemented for this thesis research is composed of these two primitive operations, which can be expressed as equations composed of known or experimentally-measured quantities. The analytical expressions for the two primitive operations defined above are shown here:

$$D_N = \frac{N_{bytes}}{BW_{ideal}} + o_{dma} \tag{5.1}$$

$$P_{buff} = t_{buff} + o_{comp} \tag{5.2}$$

For DMA transfer latencies, the number of bytes (N_{bytes}) is known for a given transfer size and BW_{ideal} is the throughput achieved as the transfer size goes to infinity (see Experiment #1). For Co-processor operation latencies, t_{buff} is an experimentally-measured quantity that represents the amount of time it takes a given co-processor to completely process one input buffer of data, once all data has been transferred to the co-processor (as defined in Experiment #2). The two remaining terms in Equations 5.1 and 5.2 above, o_{dma} and o_{comp}, are defined as "overhead terms" and represent the latency of control logic operations. Specifically, these overhead terms represent the handshaking that occurs between the PowerPC controller and the DMA engine or co-processor engine. The overhead terms o_{dma} and o_{comp} are measured experimentally using the logic analyzer to be 1.8 µs and 1.5 µs, respectively. Using only the terms and equations defined above, the first validation test of the analytical models is done by predicting local DMA transfers and comparing the predicted transfer latency and throughput with the experimentally-measured values presented in Experiment #1. Figure 5-10 shows the results of this first simple validation experiment.

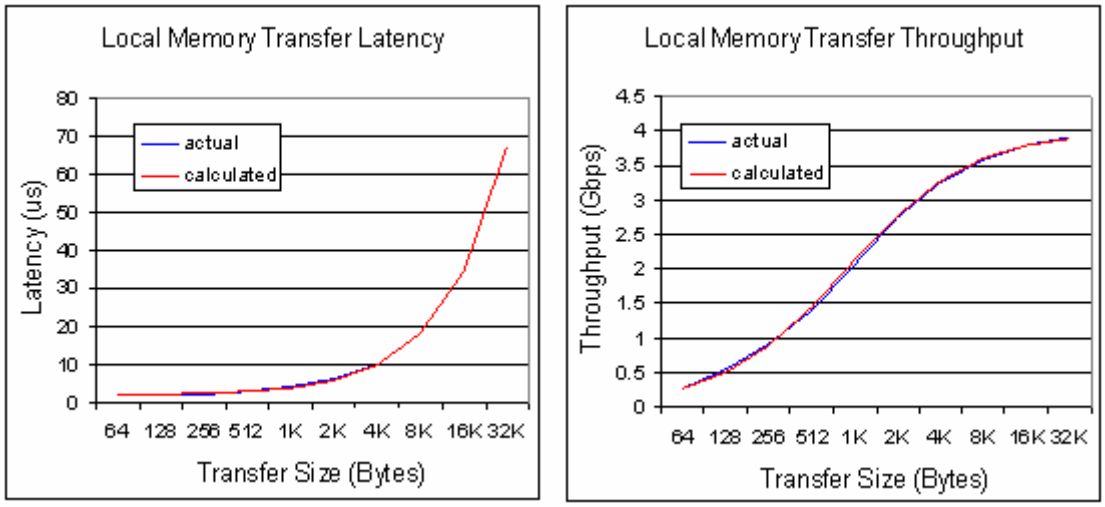


Figure 5-10. Local DMA transfer latency prediction and validation.

As can be seen by the charts above, the analytical models for simple DMA transfers are extremely accurate, due to the highly deterministic nature of the node architecture. However, more complex situations need to be analyzed before considering the analytical models of the testbed to be truly validated. The next validation experiment considers the latency of each co-processor engine to process and entire data-cube (t_{stage}), which involves both DMA transfers as well as co-processor activity. Analytical predictions of processing latencies will be compared against the experimentally-measured results presented in Experiment #2. The analytical expression that models double-buffered data-cube processing latency is shown below:

$$t_{stage} = \left(D + MAX\left(D, P_{buff}\right)\right) + \left(M \cdot MAX\left(2 \cdot D, P_{buff}\right)\right) + \left(MAX\left(D, P_{buff}\right) + D\right) \qquad (5.3)$$

The first term in Equation 5.3 represents the "startup" phase of double-buffering. The second term is the dominant term and represents the "steady-state" phase of double-buffered processing, and the third and final term represents the "wrap-up" phase that completes processing. The constant M in Equation 5.3 is a generic term that represents the number of buffers that must be filled and processed to complete an entire data-cube, and is known for any given co-processor engine and data-cube size. P_{buff} was defined earlier in this section, and the

variable D is defined as the sum of the previously-defined D_N term and a new overhead term, o_{sys}. This new overhead term is a generic term that captures "system overhead," or the non-deterministic amount of time in between consecutive DMA transfers. The o_{sys} term is not experimentally-measured, since it is not deterministic like the other two overhead terms (o_{dma} and o_{comp}), and is instead treated as a "knob" or variable to tune the predictions produced by the analytical expression. It should be noted here that all three overhead terms are a result of the relatively low-performance PowerPC controller in each node.

Notice that in Equation 5.3, there are MAX(a, b) terms; these terms reflect the behavior of double-buffered processing, which can be bounded by either computation time or communication time. Figure 5-11 below shows an illustration of both cases, with labels indicating the startup, steady-state, and wrap-up phases referred to earlier. Comparing the diagram in Figure 5-11 to Equation 5.3 should clarify how the expression was arrived at to model double-buffered processing. Figure 5-12 presents a comparison of predicted data-cube processing latencies with experimentally measured results for each co-processor engine, assuming an o_{sys} value of 3.7 µs. This value of o_{sys} is observed to be a realistic value for typical delays in between DMA transfers, based on observations made from logic analyzer captures.

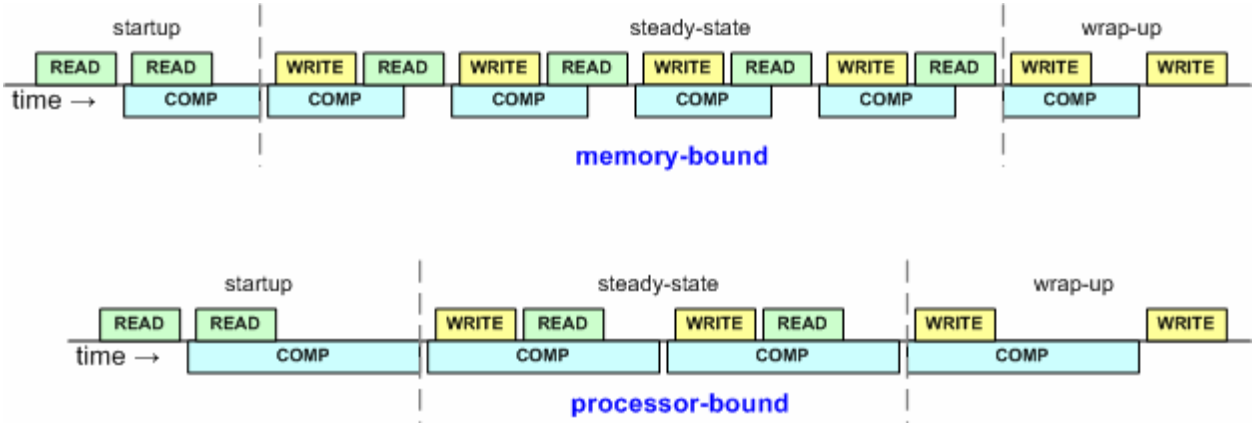


Figure 5-11. Illustration of double-buffered processing.

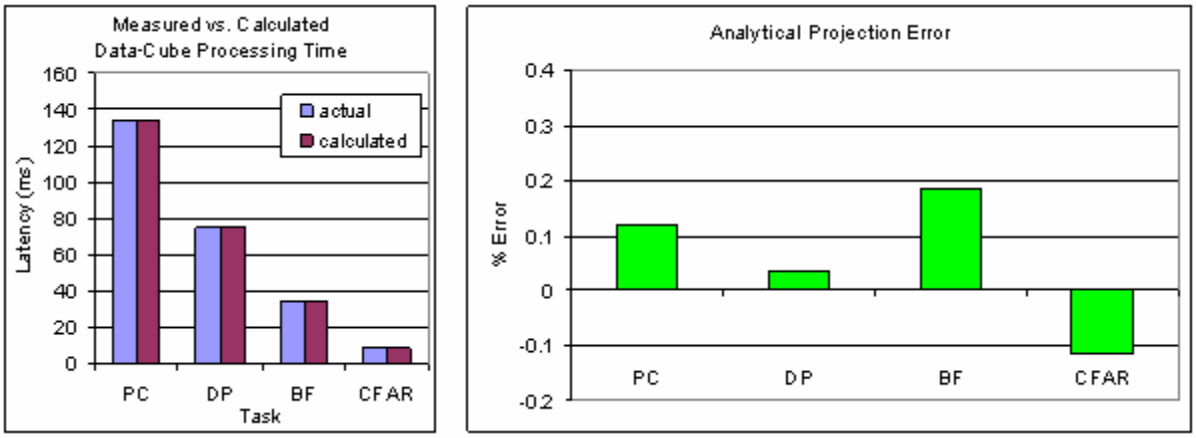


Figure 5-12. Data-cube processing latency prediction and validation.

The prediction results for data-cube processing latencies are impressively accurate, and the analytical model used to calculate those predictions are significantly more complex than the DMA transfer latency predictions presented earlier in this experiment. There is one final operation whose analytical model will be defined and validated, the corner-turn operation, before proceeding to performance projections of enhanced architectures. Based on the performance observations from Experiment #4, only "type A" corner-turns will be used. Recall that the corner-turn operation is composed of many small DMA transfers and local transposes, with the PowerPC performing the local transposes. Some of the data will stay at the same node after the corner-turn, while the rest of the data will need to be sent to a remote node's memory. Figure 5-13 shows a simplified illustration of the order of operations in a corner-turn.

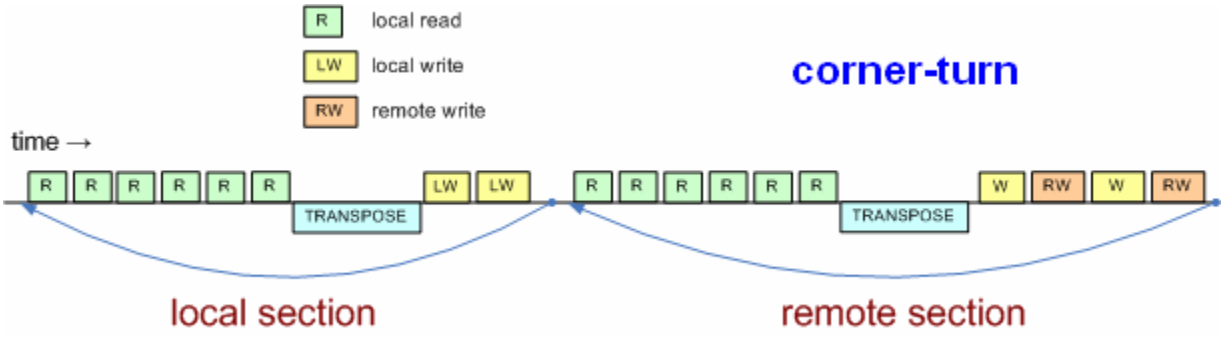


Figure 5-13. Illustration of a corner-turn operation.

The analytical expression used to model corner-turn operations is shown in Equation 5.4. The first line represents the "local section" as illustrated in Figure 5-13, while the second line represents the "remote section." Since all memory transfers must involve the external memory of the initiating node, in order to send data from local on-chip memory to a remote node, the data must first be transferred from on-chip memory to local SDRAM, and then transferred from local SDRAM to remote SDRAM.

$$t_{cornerturn} \approx A \cdot \left(\left(M \cdot \left(D_X + o_{sys} \right) \right) + t_{trans} + \left(N \cdot \left(D_Y + o_{sys} \right) \right) \right) + A \cdot \left(\left(M \cdot \left(D_X + o_{sys} \right) \right) + t_{trans} + \left(N \cdot \left(D_Y + t_{RW} + 2 \cdot o_{sys} \right) \right) \right) \quad (5.4)$$

The constants A, M, and N in Equation 5.4 represent generic terms for the number of iterations, and are determined by the size and orientation of the data-cube as well as the size of the on-chip memory used for local transposition. The D_X and D_Y terms represent local DMA transfer latencies of two different sizes, as defined in Equation 5.1. The new terms t_{trans} and t_{RW} are experimentally-measured quantities. The t_{trans} term represents the amount of time it takes to perform the local transpose of a block of data, and t_{RW} is the latency of a remote memory transfer of a given size. Figure 5-14 shows the results of the analytical model predictions.

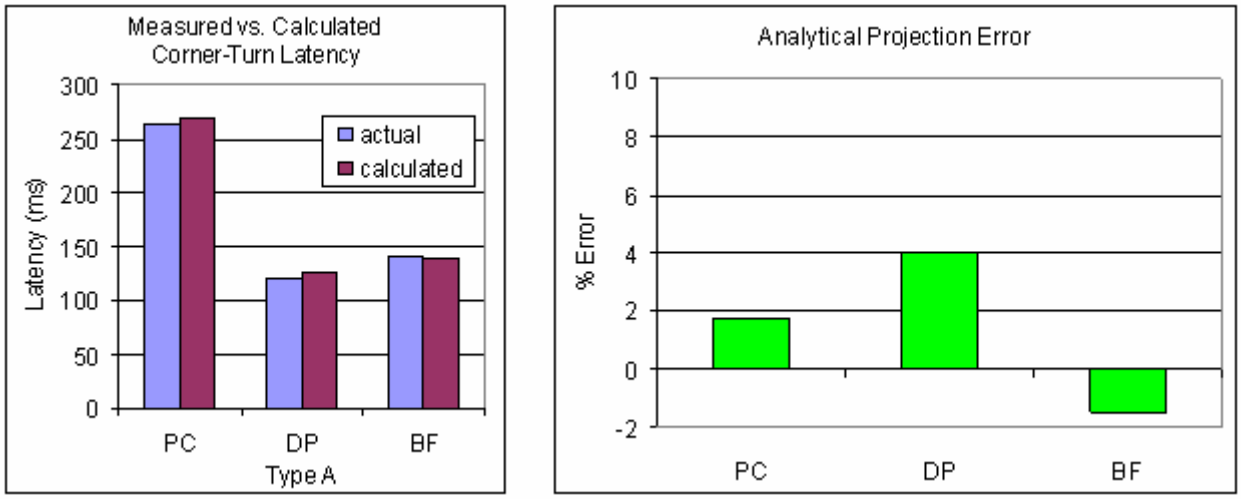


Figure 5-14. Corner-turn latency prediction and validation.

The corner-turn latency predictions are noticeably less accurate than the previous analytical models, due to the DMA-bound progression of the operation. However, the maximum error is still less than 5% for all cases, which is a sufficiently accurate prediction of performance to instill confidence in the analytical model proposed in Equation 5.4. Now that the analytical models of GMTI have been completely defined and validated, architectural enhancements will be analyzed for potential performance improvement using these analytical models.

The first performance projection looks at reducing the high overhead introduced by the PowerPC controller in each node. Additionally, two pulse compression engines per node are assumed, since there is sufficient memory bandwidth to keep both engines at near 100% utilization based on observations from Experiment #2. In order to model system performance assuming minimal overhead, the three overhead terms, o_{dma}, o_{comp}, and o_{sys} are set to 50 nanoseconds each. While it would not be realistic to reduce the overhead terms to zero, a small latency of 50 nanoseconds is feasible as it corresponds to several clock cycles around 100 MHz. Highly-efficient control logic implemented in the reconfigurable fabric of the FPGA would absolutely be able to achieve such latencies. As a complementary case-study, to further reduce the negative impact of the PowerPC on system performance, corner-turns could be enhanced with hardware-assisted local transposes. Instead of having the PowerPC perform the transpose through software, a fifth co-processor engine could be designed that performs a simple data transfer from one BRAM to another, with address registers that progress through the appropriate order of addresses to transpose the data from the input BRAM to the output. Using this type of co-processor engine, one data element would be moved per clock cycle, drastically reducing the t_{trans} term in Equation 5.4. Figure 5-15 shows the full predicted GMTI application latency for three cases: (1) baseline architecture (labeled ORIGINAL), (2) optimized control logic and two

pulse compressions engines (labeled OPTIMIZED), and (3) optimized control logic and two pulse compression engines, along with hardware-assisted corner-turns (labeled HW-CT).

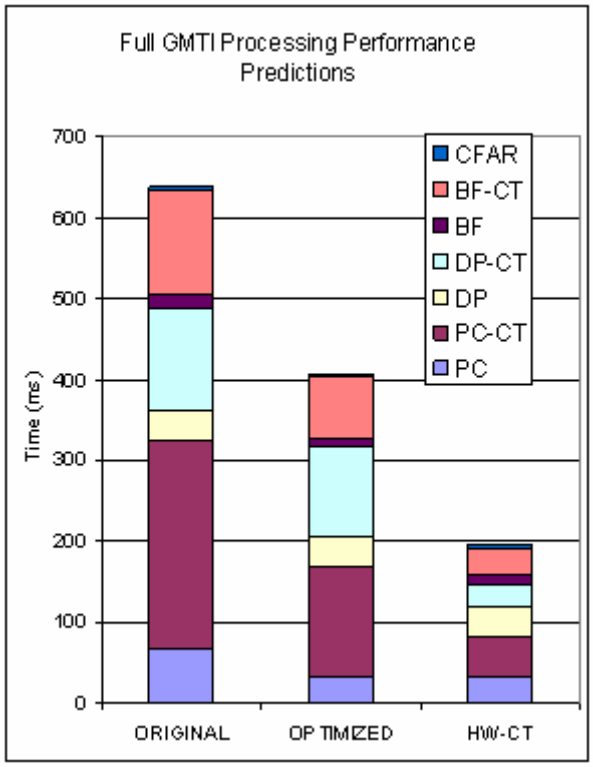


Figure 5-15. Full GMTI application processing latency predictions.

Aside from halving the pulse compression latency due to having two pulse compression engines, the other processing stages do not experience significant performance gains by minimizing the control overhead. Because the Doppler processing and CFAR stages achieve nearly 100% processor efficiency, all of the control logic latency associated with DMA transfers is hidden behind the co-processor activity, so little performance gain from optimized control logic is expected. For the beamforming stage, although it is difficult to discern from the chart, the beamforming processing latency is reduced from 17.2 ms to 11.6 ms. This result is also intuitive since the beamforming stage did not achieve 100% processor utilization in the baseline design, and thus beamforming stands to benefit from improving control logic overhead.

However, notice the significant improvement in corner-turn operation latencies. Since corner-turns do not implement any latency hiding techniques (such as double-buffering), the majority of the operation latency for the baseline design is caused by the high control overhead. However, with only optimized control logic, and no hardware-assisted local transposes, the performance gain is somewhat limited. The relatively low performance of the PowerPC is further exposed by the hardware-assisted corner-turn predictions, in which the corner-turn operations experience another significant performance improvement by offloading the local transposes to dedicated logic in the reconfigurable fabric of the FPGA.

One additional architectural enhancement is modeled to further improve corner-turn performance for GMTI. The second performance projection is made assuming that there is a direct path connecting the internal on-chip memory with the RapidIO network interface at each node, so that data from on-chip memory does not need to go through local SDRAM before being sent to a remote node. The only kernel of this implementation of GMTI that could benefit from such an enhancement is the corner-turn, since the only inter-processor communication occurs during corner-turns. From Figure 5-13, it can be seen that there are two write operations that occur per loop iteration in the remote section of a corner-turn. By enhancing the node architecture with the new data path, the remote section would be reduced to a single write operation that goes straight from the scratchpad memory of the local transpose to the remote node's external SDRAM. An additional benefit of this enhancement is a reduction of the load on local external memory. For the baseline corner-turn, both of the write operations in the remote section require the use of local external memory, whereas with this new transfer directly from on-chip memory to the network, the local external memory is not used at all. Figure 5-16 shows the predicted performance improvement for the corner-turn operation with this new data path.

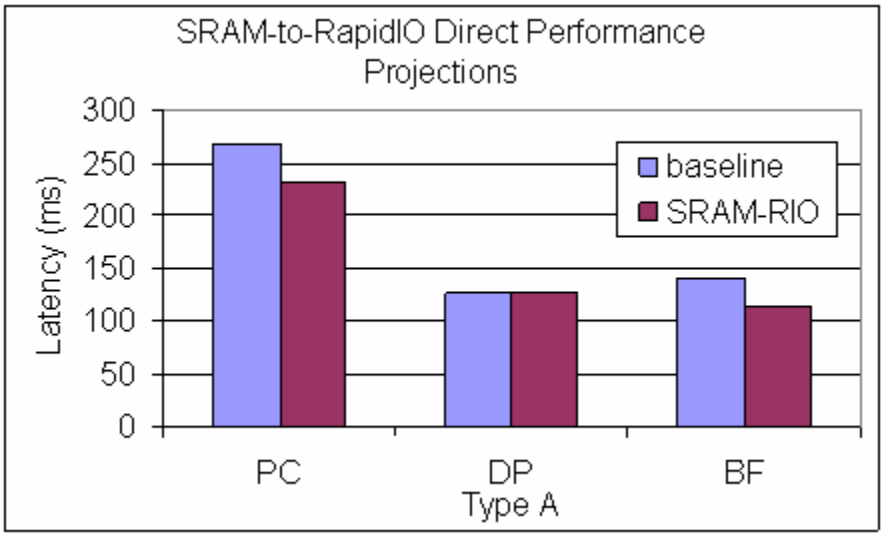


Figure 5-16. Corner-turn latency predictions with a direct SRAM-to-RapidIO data path.

The results shown in Figure 5-16 are made relative to the baseline node architecture, without the control logic optimizations or hardware-assisted corner-turns described in the previous projections. Surprisingly, not much improvement is seen in the enhanced architecture with the new data path between on-chip memory and the network interface. For the corner-turn that follows Doppler processing, *no* improvement is seen, however this result is expected since no data is exchanged between nodes for this particular corner-turn. But for the other two corner-turns, the limited performance improvement is due to a couple of reasons. First, as can be seen by inspecting Figure 5-13, only a small portion of the overall corner-turn operation involves remote memory transfers (which are the only transfers that benefit from the new data path). In the two-node system, only half of the data at each node needs to be exchanged with the other node. If the node count of the system were to increase, so too would the percentage of data at each node that would need to be written remotely (for example, in a four-node system, 75% of the data at each node would need to be written remotely), and as a result the performance improvement relative to the baseline design would increase. Secondly, any performance gain due to the new data path would be additive with a performance improvement from hardware-

assisted local transposes. Therefore, if compared to a baseline architecture that includes the faster hardware-assisted transposes, the improvement due to the new data path would be more significant relative to the baseline performance.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

This section will summarize the research presented in this thesis, as well as offer concluding remarks to review the insight gained from the experiments performed in the course of this work. Some suggestions are also offered for potential directions that could be taken to extend the findings of this research, as well as expand the capabilities of the experimental platform.

Summary and Conclusions

A promising new system architecture based on FPGAs was presented to support demanding space-based radar applications. This research goes far beyond the simple implementation and demonstration of IP cores on an FPGA, by investigating node-level and system-level integration of cores and components. The relatively low-frequency design was able to demonstrate impressive sustained performance, and the highly-modular architecture will ease the integration of more high-performance components without disrupting the overall design philosophy. By building and demonstrating one of the first working academic RapidIO testbeds, this thesis contributed much-needed academic research on this relatively new interconnect standard.

Several challenges were encountered during the course of this research, namely restrictions imposed by the hardware platform such as limited node memory and insufficient numerical precision for the case-study application. Most real systems will provide some external SRAM to each node for processing memory, which would permit larger data sets necessary to truly emulate realistic space-based radar data sizes. Furthermore, the limited system size prevented scalability experiments and restricted the complexity of the communication patterns that might stress the underlying system interconnect.

One important observation to come from this research is that, with the proposed node architecture, a pipelined parallel decomposition would maximize the processor utilization at each node. The highly efficient co-processor engines typically require the full bandwidth provided to local external memory at each node in order to maximize utilization, making it impossible to use multiple co-processor engines concurrently in each node. However, pipelined decompositions may exhibit a higher processing latency per data cube and, due to the nature and purpose of the real-time GMTI algorithm, this increased latency may be intolerable. Even if the system is able to provide the necessary throughput to keep up with incoming data cubes, if the target detections for each cube are provided too long after the raw data is acquired, the detections may be stale and thus useless. The system designer should carefully consider how important individual co-processor utilization is compared to the latency requirements for their particular implementation. If co-processor utilization is not considered important, then a data-parallel decomposition would be possible which would reduce the latency of results for each individual data cube.

A novel design approach presented in this research is the use of a multi-ported external memory controller at each node. While most current processor technologies connect to the memory controller through a single bus which must be shared by various on-chip components, the node design proposed in this thesis provides multiple dedicated ports to the memory controller. This multi-port design philosophy allows each component to have a dedicated control and data path to the memory controller, increasing intra-node concurrency. By over-provisioning the bandwidth to the physical external memory devices relative to any given internal interface to the controller, each port can be serviced at its maximum bandwidth without interfering with the other ports by using only a simple arbitration scheme.

Another design technique demonstrated in this research that takes advantage of the flexible nature of FPGAs is the "domain isolation" of the internal components of each node. Each component within a node, such as the external memory controller, on-chip memory controller, co-processor engines, and network interface, operate in different clock domains and may even have different data path widths. The logic isolation allows each part of the node to be optimized independent of the other components, so that the requirements of one component do not place unnecessary restrictions on the other components.

It is important to suggest a path to industry adoption of any new technology, and to understand the concerns that might prevent the community from accepting a new technology or design philosophy. Today, almost all computer systems make use of a dedicated memory controller chip, typically in the form of a Northbridge chipset in typical compute nodes or otherwise a dedicated memory controller on a mass-memory board in a chassis system. Instead of replacing the main processors of each node with the FPGA node design presented in this thesis, a less-risky approach would be to replace the dedicated memory controller chips in these systems with this FPGA design. This approach would integrate some limited processing capability close to the memory, circumventing the need to provide an efficient data path from host memory to an FPGA co-processor, as well as permit a variety of high-performance interconnects through the reconfigurable fabric of the FPGA, as opposed to the typical PCI-based interconnects associated with most memory controller chips.

Future Work

There are a variety of promising and interesting directions that could be taken to extend the findings of this research. While valuable insight has been gained, there were several restrictions imposed by the experimental hardware, as well as compromises made in order to reduce the scope of the research to a manageable extent. From expanding the size of the testbed, to

including Serial RapidIO and other logical layer variants, to experimenting with different applications or even application domains, this thesis is just the start of a potentially large and lucrative body of unique academic research.

The most immediate step to extend this research is to increase the size of the testbed, either through integration of RapidIO switches with the experimental platform and increasing the node count, or by selecting a new, more stable experimental testbed all-together. A larger node count is necessary to mimic a realistic flight system, and switched topologies would enable more complex inter-node traffic patterns as well as high-speed data input into the processing system. A high-speed data source is needed to enable true real-time processing for applications such as GMTI. Ideally, a chassis-based platform could be used to replace the stand-alone boards used for this research, providing a more realistic platform as well as reduce the risk of development.

Furthermore, it should be noted that GMTI is only one example application of interest for high-performance space computing. Other applications within the domain of space-based radar and remote sensing, such as Synthetic Aperture Radar or Hyperspectral Imaging, place different loads and requirements on the underlying system architecture. Other application domains, such as telecommunication or autonomous control, could also be investigated to cover a diverse range of application behaviors and requirements.

Finally, only one of several RapidIO variants was used in this thesis research, the 8-bit parallel physical layer and the Logical I/O logical layer. Serial RapidIO has become much more popular since the initial release of the RapidIO specification, and other logical layer variants such as the Message Passing logical layer would provide an interesting contrast to the programming model offered by the memory-mapped Logical I/O logical layer variant. To provide a thorough evaluation of RapidIO's suitability for the targeted application domain,

several high-performance interconnect protocols could be evaluated and compared in order to highlight the strengths and weaknesses of each.

LIST OF REFERENCES

[1] D. Bueno, C. Conger, A. Leko, I. Troxel and A. George, "Virtual Prototyping and Performance Analysis of RapidIO-based System Architectures for Space-Based Radar," Proc. of 8th High-Performance Embedded Computing (HPEC) Workshop, MIT Lincoln Lab, Lexington, MA, September 28-30, 2004.

[2] D. Bueno, A. Leko, C. Conger, I. Troxel, and A. George, "Simulative Analysis of the RapidIO Embedded Interconnect Architecture for Real-Time, Network-Intensive Applications," Proc. of 29th IEEE Conference on Local Computer Networks (LCN) via the IEEE Workshop on High-Speed Local Networks (HSLN), Tampa, FL, November 16-18, 2004.

[3] D. Bueno, C. Conger, A. Leko, I. Troxel, and A. George, "RapidIO-based Space Systems Architectures for Synthetic Aperture Radar and Ground Moving Target Indicator," Proc. of 9th High-Performance Embedded Computing (HPEC) Workshop, MIT Lincoln Lab, Lexington, MA, September 20-22, 2005.

[4] T. Hacker, R. Sedwick, and D. Miller, "Performance Analysis of a Space-Based GMTI Radar System Using Separated Spacecraft Interferometry," M.S. Thesis, Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, Boston, MA, 2000.

[5] M. Linderman and R. Linderman, "Real-Time STAP Demonstration on an Embedded High Performance Computer," Proc. of 1997 IEEE National Radar Conference, Syracuse, NY, May 13-15, 1997.

[6] R. Brown and R. Linderman, "Algorithm Development for an Airborne Real-Time STAP Demonstration," Proc. of 1997 IEEE National Radar Conference, Syracuse, NY, May 13-15, 1997.

[7] D. Rabideau and S. Kogon, "A Signal Processing Architecture for Space-Based GMTI Radar," Proc. of 1999 IEEE Radar Conference, April 20-22, 1999.

[8] D. Sandwell, "SAR Image Formation: ERS SAR Processor Coded in MATLAB," Lecture Notes - Radar and Sonar Interferometry, Dept. of Geography, University of Zurich, Copyright 2002.

[9] Pentek, Inc., "GateFlow Pulse Compression IP Core," Product Sheet, July, 2003.

[10] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, M. Linderman, and R. Brown, "Design, Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers," IEEE Trans. on Aerospace and Electrical Systems, vol. 36, pp. 528-548, April 2000.

[11] M. Lee, W. Liu, and V. Prasanna, "Parallel Implementation of a Class of Adaptive Signal Processing Applications," Algorithmica, 2001.

[12] M. Lee and V. Prasanna, “High Throughput-Rate Parallel Algorithms for Space-Time Adaptive Processing,” Proc. of 2nd International Workshop on Embedded HPC Systems and Applications, April 1997.

[13] A. Ahlander, M. Taveniku, and B. Svensson, “A Multiple SIMD Approach to Radar Signal Processing,” Proc. of 1996 IEEE TENCON, Digital Signal Processing Applications, vol. 2, pp. 852-857, November 1996.

[14] T. Haynes, “A Primer on Digital Beamforming,” Whitepaper, Spectrum Signal Processing, Inc., March 26, 1998.

[15] J. Lebak, A. Reuther, and E. Wong, “Polymorphous Computing Architecture (PCA) Kernel-Level Benchmarks,” HPEC Challenge Benchmark Suite Specification, Rev. 1, June 13, 2005.

[16] R. Cumplido, C. Torres, S. Lopez, “On the Implementation of an Efficient FPGA-based CFAR Processor for Target Detection,” International Conference on Electrical and Electronics Engineering (ICEEE), Acapulco, Guerrero; Mexico, September 8-10, 2004.

[17] D. Bueno, “Performance and Dependability of RapidIO-based Systems for Real-time Space Applications,” Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL, 2006.

[18] M. Skalabrin and T. Einstein, “STAP Processing on Multiprocessor Systems: Distribution of 3-D Data Sets and Processor Allocation for Efficient Interprocessor Communication,” Proc. of 4th Annual Adaptive Sensor Array Processing (ASAP) Workshop, March 13-15, 1996.

[19] K. Varnavas, “Serial Back-plane Technologies in Advanced Avionics Architectures,” Proc. of 24th Digital Avionics System Conference (DASC), October 30-November 3, 2005.

[20] Xilinx, Inc., “Xilinx Solutions for Serial Backplanes,” 2004. http://www.xilinx.com/esp/networks_telecom/optical/collateral/backplanes_xlnx.pdf. Last accessed: March 2007.

[21] S. Vaillancourt, “Space Based Radar On-Board Processing Architecture,” Proc. of 2005 IEEE Aerospace Conference, pp.2190-2195, March 5-12, 2005.

[22] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," ACM Computing Surveys, vol. 34, No. 2, pp. 171-210, June 2002.

[23] P. Murray, “Re-Programmable FPGAs in Space Environments,” White Paper, SEAKR Engineering, Inc., Denver, CO, July 2002.

[24] V. Aggarwal, “Remote Sensing and Imaging in a Reconfigurable Computing Environment”, M.S. Thesis, Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL, 2005.

[25] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, and A. George, “Hardware/software Interface for High-performance Space Computing with FPGA Coprocessors,” Proc. IEEE Aerospace Conference, Big Sky, MN, March 4-11, 2006.

[26] I. Troxel, “CARMA: Management Infrastructure and Middleware for Multi-Paradigm Computing,” Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL, 2006.

[27] C. Conger, I. Troxel, D. Espinosa, V. Aggarwal and A. George, “NARC: Network-Attached Reconfigurable Computing for High-performance, Network-based Applications,” International Conference on Military and Aerospace Programmable Logic Devices (MAPLD), Washington D.C., September 8-10, 2005.

[28] Xilinx, Inc., “Xilinx LogiCORE RapidIO 8-bit Port Physical Layer Interface, DS243 (v1.3),” Product Specification, 2003.

[29] Xilinx, Inc., “LogiCORE RapidIO 8-bit Port Physical Layer Interface Design Guide,” Product Manual, November 2004.

[30] Xilinx, Inc., “Xilinx LogiCORE RapidIO Logical I/O ad Transport Layer Interface, DS242 (v1.3),” Product Specification, 2003.

[31] Xilinx, Inc., “LogiCORE RapidIO Logical I/O & Transport Layer Interface Design Guide,” Product Manual, November 2004.

[32] S. Elzinga, J. Lin, and V. Singhal, “Design Tips for HDL Implementation of Arithmetic Functions,” Xilinx Application Note, No. 215, June 28, 2000.

[33] N. Gupta and M. George, “Creating High-Speed Memory Interfaces with Virtex-II and Virtex-II Pro FPGAs,” Xilinx Application Note, No. 688, May 3, 2004.

[34] S. Fuller, “RapidIO - The Next Generation Communication Fabric for Embedded Application,” Book, John Wiley & Sons, Ltd., West Sussex, England, January 2005.

[35] G. Shippen, “RapidIO Technical Deep Dive 1: Architecture and Protocol,” Motorola Smart Network Developers Forum 2003, Dallas, TX, March 20-23, 2003.

[36] J. Adams, C. Katsinis, W. Rosen, D. Hecht, V. Adams, H. Narravula, S. Sukhtankar, and R. Lachenmaier, “Simulation Experiments of a High-Performance RapidIO-based Processing Architecture,” Proc. of IEEE International Symposium on Network Computing and Applications, Cambridge, MA, Oct. 8-10, 2001.

[37] RapidIO Trade Association, "RapidIO Interconnect Specification Documentation Overview," Specification, June 2002.

[38] RapidIO Trade Association, "RapidIO Interconnect Specification (Parts I-IV)," Specification, June 2002.

[39] RapidIO Trade Association, "RapidIO Interconnect Specification, Part V: Globally Shared Memory Logical Specification," Specification, June 2002.

[40] RapidIO Trade Association, "RapidIO Interconnect Specification, Part VI: Physical Layer 1x/4x LP-Serial Specification," Specification, June 2002.

[41] RapidIO Trade Association, "RapidIO Interconnect Specification, Part VIII: Error Management Extensions Specification," Specification, June 2002.

[42] C. Conger, D. Bueno, and A. George, "Experimental Analysis of Multi-FPGA Architectures over RapidIO for Space-Based Radar Processing," Proc. of 10th High-Performance Embedded Computing (HPEC) Workshop, MIT Lincoln Lab, Lexington, MA, September 19-21, 2006.

[43] IBM Corporation, "The CoreConnect Bus Architecture," White paper, Armonk, NY, 1999.

[44] J. Jou and J. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," Proc. of IEEE, vol. 74, No. 5, pp. 732-741, May 1986.

BIOGRAPHICAL SKETCH

Chris Conger received a Bachelor of Science in Electrical Engineering from The Florida State University in May of 2003. In June of 2003, he also received his Engineer Intern certificate from the Florida Board of Professional Engineers in preparation for a Professional Engineer license. In his final year at Florida State, he began volunteering for the High-performance Computing and Simulation (HCS) Research Lab under Dr. Alan George through the Tallahassee branch of the lab.

Upon graduation from Florida State, Chris moved to Gainesville, FL, to continue his education under Dr. George and with the HCS Lab at The University of Florida. He became a research assistant in January 2004, and has since worked on a variety of research projects involving FPGAs, parallel algorithms, and the design of high-performance embedded processing systems.