Multiparadigm Computing for Space-Based Synthetic Aperture Radar

Adam Jacobs*, Grzegorz Cieslewski*, Casey Reardon†, Alan D. George*†
*High-performance Computing and Simulation (HCS) Research Laboratory
†NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida
{jacobs, cieslewski, reardon, george}@hcs.ufl.edu

***Abstract*—Projected computational requirements for future space missions are outpacing technologies and trends in conventional embedded microprocessors. In order to meet the necessary levels of performance, new computing technologies are of increasing interest for space systems, such as reconfigurable devices and vector processing extensions. These new technologies can also be used in tandem with conventional general-purpose processors in the form of multiparadigm computing. By using FPGA resources and AltiVec extensions, as well as MPI extensions for multiprocessor support, we explore possible hardware/software designs for a synthetic aperture radar application. Design of key components of the SAR application including range compression and azimuth compression will be discussed, and hardware/software performance tradeoffs analyzed. The performance of these key components will be measured individually, as well as in the context of the entire application. Fault-tolerant versions of range and azimuth compression algorithms are proposed and their performance overhead is evaluated. Our analysis compares several possible multiparadigm systems, achieving up to $18\times$ speedup while also adding fault tolerance to a pre-existing SAR application.**

I. Introduction

As space-based remote sensor technologies increase in fidelity, the amount of data being collected by orbiting satellites will continue to outpace the ability to transmit that data down to ground stations. This downlink bottleneck, which is caused by bandwidth limitations and high latency, can be alleviated by allowing additional computational processing to be performed by satellites or other spacecraft in order to reduce their data transmission requirements. Even for applications that do not require a large downlink channel, improved computational performance could enable more complex and autonomous capabilities. Additionally, achieving real-time deadlines may also require improved processing performance. The required performance for these applications cannot be reached using current and near-future radiation-hardened microprocessors. Instead, higher performance commercial-off-the-shelf (COTS) devices, along with fault-tolerant computing techniques, may be used to achieve mission requirements.

Space environments have several additional requirements that are not always high priorities for terrestrial applications. The high-radiation environment usually prohibits the use of COTS microprocessors and FPGAs. Instead, radiation-hardened devices, which are resistant to Single-Event Upsets (SEUs), are used to ensure reliability and correctness. The downside of using rad-hard devices is lower operating frequencies, larger die sizes, higher power consumption, and higher costs. In order to get the performance and cost benefits of COTS components while maintaining the reliability of rad-hard parts, systems such as the NASA Dependable Multiprocessor may be used.

The NASA Dependable Multiprocessor (DM) project aims to achieve reliable computation with the use of COTS technologies in order to provide a small, low-power supercomputer for space [1]. The DM system uses a cluster of high-performance COTS PowerPC CPUs connected via Gigabit Ethernet to obtain high-performance data processing, while employing a reliable software middleware to handle SEUs when they occur. One of the goals of the DM system is to provide a familiar software interface for developers of scientific applications through the use of standard programming tools such as C and Message Passing Interface (MPI) running on Linux. Fault tolerance features, such as checkpointing and replication, can be added by using API calls within a program. Additional performance can be achieved by using the on-chip AltiVec vector processing engine and through external Field-Programmable Gate Array (FPGA) co-processors.

One potential application that could benefit from increased processing capabilities is Synthetic Aperture Radar (SAR). Space-based SAR applications process large amounts of raw radar data in order to create high-resolution topographical images of the Earth's surface. The size of SAR input datasets can be on the order of several gigabytes, while the output can be significantly smaller, depending on the application's goals. These algorithms are very computationally intensive, requiring both large amounts of processing and memory bandwidth. In this paper we explore several options for exploiting the parallelism of an example SAR application on a system similar to the DM. Parallelism at several different scales will be exploited using a variety of tools and technologies. The AltiVec engine can take advantage of low-level data parallelism, high-level data parallelism can be exploited with multiple processing nodes using MPI and FPGAs can be employed to process data showing intermediate levels of parallelism.

The remaining sections of this paper are organized as follows. Section 2 gives an overview of the SAR algorithm and

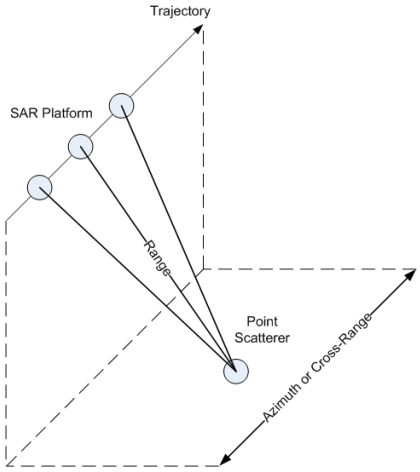

Fig. 1. Synthetic Aperture Radar Description

fault tolerance techniques that will be used in the following analysis. Section 3 examines the parallel partitioning strategies that will be examined on the target system. Section 4 discusses the design of components that are to be used on an FPGA for application speedup. Section 5 explores fault tolerance techniques that can be used to ensure data integrity for the SAR application. Section 6 presents results and analysis from experiments on the target platform as well as simulative results of other possible architectures. Finally, Section 7 presents conclusions and provides directions for future research.

II. Background

A. Synthetic Aperture Radar

SAR works by emitting high-frequency electromagnetic pulses and receiving the echoes as they are reflected by targets within a given scene. The basic principle is shown in Figure 1. Raw sensor data is interpreted as a two-dimensional matrix. Data along the first dimension corresponds to the range, which is the distance to the scatterer. The second dimension corresponds to the azimuth, which is the location of the scatterer along the trajectory of the SAR platform. The purpose of SAR processing is to convert the radar returns into an image of the scene being viewed. One common SAR processing algorithm is known as Range-Doppler Processing (RDP) [2]. The processing of data in this algorithm can be viewed as a 2D convolution and implemented with a linear filter. However, this approach is computationally inefficient. Instead, since the range and azimuth time scales are nearly orthogonal, the range and azimuth directions can be independently computed while processing in the frequency domain. The core steps of the SAR algorithm used in this paper will be discussed in the next section.

B. Fault-Tolerant Computing

For space applications, designing for reliability and fault tolerance are major requirements. Missions must function correctly in the presence of single-event effects without any human intervention. In order to ensure data integrity at a system level, designers employ several methodologies such as fault isolation and containment. A reliable system such as the DM ensures that system components are working correctly, and if not, takes corrective action. While these high-level methodologies will prevent total system failure, silent data corruption can still make computed results meaningless. A description of every fault mitigation technique is beyond the scope of this discussion, but we will review three techniques that closely relate to this work. The first technique, Algorithm-Based Fault Tolerance (ABFT), is applicable to many scientific applications whose computation consists of linear algebra operations. The other two techniques are important for FPGA-based computing, Triple Modular Redundancy (TMR) and scrubbing.

1) Algorithm-Based Fault Tolerance: ABFT, introduced by Huang and Abraham [3], makes use of properties of linear algebra kernels to achieve fault detection and correction. Each matrix of a linear algebra operation is encoded by adding redundant rows and/or columns filled with weighted checksums that will be preserved throughout the mathematical operation. Errors in the calculation can be detected by verifying the resulting checksums and, if a mismatch is detected, an appropriate recovery technique can be employed. Recovery may consist of correcting an individual, incorrectly computed value, recomputing an entire block of computation, or notifying a higher-level system that an error has been detected.

A version of fault-tolerant RDP employing ABFT has been proposed in [4] but it uses a simplified model for azimuth compression in which the filter applied remains constant. Such simplification allows the same fault-tolerant technique to be applied in range and azimuth compression but it reduces the quality of the processed image. Fang, Le and Taft [5] also investigated the RDP on an FPGA-based system and employed device-level TMR for fault tolerance via FPGAs. While this solution may be acceptable in some situations, an ABFT approach would significantly reduce the complexity and increase the efficiency of the system.

2) FPGA Fault Tolerance Techniques: The most significant fault-tolerance issue for COTS FPGAs is the use of an SRAM-based configuration memory. When a single-event upset (SEU) affects the configuration memory, it is possible that the functionality of a given FPGA design will change (through altered internal lookup tables or routing resources). Such a fault is persistent, and additional faults can accumulate until the device is reconfigured. While many radiation-tolerant FPGAs using antifuse or flash-based configuration memories do not experience these faults, they suffer limited reconfiguration abilities and are usually much smaller than their SRAM-based counterparts. In addition to configuration memory upsets, most COTS FPGAs are also susceptible to transient logic errors that can occur in any non-radiation-hardened logic device.

There are two widely-accepted methods for protecting against SEUs in FPGAs: triple modular redundancy (TMR) and scrubbing. With TMR, internal design components are

triplicated and their outputs are periodically voted upon. For Xilinx FPGAs, the Xilinx TMRTool can help designers implement a TMR version of their already existing design [6]. TMR will mask any error in one of the three replicas, as well as any transient faults that occur. The drawback to this approach is the increased resource utilization ($> 3\times$) needed for replication and voting. Additionally, multiple errors can and will accumulate in the configuration memory if it is not periodically refreshed.

The other fault-tolerant technique, scrubbing, periodically reconfigures the FPGA device with a valid copy of its configuration to ensure correctness. The verification copy of the bitfile must be stored in a radiation-hardened device to ensure that it will not become corrupted. The scrub rate (frequency of reconfiguration) can be varied based on performance and environmental factors. For devices that support partial reconfiguration, the current configuration memory can be periodically compared to a known good value, and only corrupted portions of the configuration memory need to be updated. TMR is usually augmented with scrubbing to eliminate the collection of configuration errors.

III. Application Description

The SAR application featured in this paper is a range-doppler processing algorithm that processes raw radar data from an ERS-2 satellite [7]. Both sequential and parallel versions of the code exist. In the sequential case, the initial data is split into "patches" that will fit into a system's main memory and avoid disk-based swap files. Each patch is processed independently, creating a portion of the final output image. For each patch, the following procedure is followed:

1) Preprocessing
2) Range compression for each range
3) Transpose data
4) Azimuth compression for each azimuth
5) Postprocessing

Preprocessing performs basic signal conditioning operations such as removing DC offset and accounting for sensor bias. Range compression uses a constant filter to perform pulse compression. The transpose orients the data in memory to allow for better data locality for the subsequent processing stage. Azimuth compression is similar to range compression, except that a space-variant filter is used, varying for each azimuth. Finally, postprocessing transposes the data and performs additional computations to transform the complex results into real data for human readable images.

Our parallel version of SAR was originally developed to be executed on a cluster of traditional microprocessors. In the parallel version, the data parallelism inherent between patches is used to efficiently map the application to a clustered architecture. Patches are distributed in a round-robin fashion to individual worker nodes of a multi-node system and the processed data is collected and written to disk on a single master node. The five steps listed previously are performed on each worker node.

Before exploring the design space for areas that can most benefit from the parallelism provided by FPGAs, an investigation of the application's performance characteristics can identify and prioritize the major computational areas of the software. Efforts were made to optimize the software code whenever possible. Fast Fourier Transforms (FFT) were performed using the FFTW3 library, a widely recognized, efficient FFT library [8]. The FFTW library makes extensive use of the processor's AltiVec resources to obtain its high performance. The application was compiled using GCC 4.1.2 using `-O2` and `-mabi=altivec` compiler optimizations. Table II shows the percentage of execution time required by each portion of the serial application. Preprocessing is included in the range compression stage.

TABLE I
SAR Application Profile

Stage	Exec. Time	Exec. Percentage
Range compression	4.21 s	11.6%
Transpose	2.97 s	8.2%
Azimuth compression	23.75 s	65.6%
Postprocessing	5.28 s	14.6%

The profiling results in Table I suggest that the azimuth compression is a prime candidate to offload to an FPGA device since it occupies nearly 66% of the execution time for SAR. Additionally, this stage contains many computations, such as multiple Fast Fourier Transform (FFT) and filter calculations, that could be accelerated using FPGAs by exploiting their fine-grained parallelism. Even though range compression represents less than 12% of the execution time, it contains many similarities to azimuth compression, and both will be explored in the next section. The transpose and postprocessing functions represent a non-trivial portion of execution time that does not favorably map to devices (such as FPGAs) that cannot directly and efficiently manipulate the system's main memory.

IV. FPGA Core Design

The Alpha Data ADM-XRC-4 FPGA boards used for this work each consist of a Xilinx Virtex4-SX55 FPGA along with 16MB of SRAM split into four separate banks. The board connects to the remainder of the system through a 32-bit PCI bus running at 66 MHz. An external PCI controller is used to simplify communication with the host. The Xilinx Virtex4 has a very limited amount of on-chip memory (approximately 6 Mbits total) and the ADM-XRC-4's SRAM is also limited, especially considering the dataset size (300MB per patch) for this SAR application. However, the SRAM memory is used as much as possible because the number of data transfers from the CPU to the FPGA needs to be minimized to alleviate the bottleneck presented by the PCI bus. Transferring data across the PCI bus is much more efficient for a few, large segments than it is for many small segments. Intermediate results are stored in the on-board memory, instead of being communicated back to the host processor in order to avoid unnecessary communication.

In this section, we will present hardware architectures for range and azimuth compression and discuss their functionality and amenability on the target platform.

A. Range Compression

Range compression is a process that correlates a received radar signal with the transmitted pulse that it was generated from. This process is accomplished with the use of a matched filter. While there are several methods for performing this calculation, the most efficient method requires conversion to the frequency domain using an FFT. The resulting spectrum is then multiplied by the filter and the result is converted back to the spatial domain using an inverse FFT.

Depending on the actual operational scenario, the preprocessing stage may contain several operations. For the current study, preprocessing takes the input data (8-bit complex data pairs) and removes the DC offset. Additionally, data vectors are zero-padded to allow for the use of high-performance FFT algorithms. For the data set used in this discussion, range vectors are increased from 5,616 to 8,192 elements.

While the software version of SAR uses floating-point precision for all arithmetic operations, there are a number of limitations to using floating-point cores in an FPGA. The main drawback of floating-point precision is the amount of logic resources necessary for adders and multipliers. For the FPGA design, a fixed-point FFT core, available from Xilinx [9], was used to implement forward and inverse FFTs. The FPGA design uses a pipeline approach where new input data can be processed every clock cycle. Since the input data set does not have a large dynamic range, the fixed-precision design will produce accurate results.

Figure 2 shows an architectural diagram of the range compression core. The initial 8192-point FFT uses 8-bit complex input data. The FFT outputs are then multiplied by filter coefficients and scaled to 16-bit fixed-precision numbers. The inverse transform uses 16-bit inputs and produces 16-bit outputs, which are then converted to single-precision floating-point format.

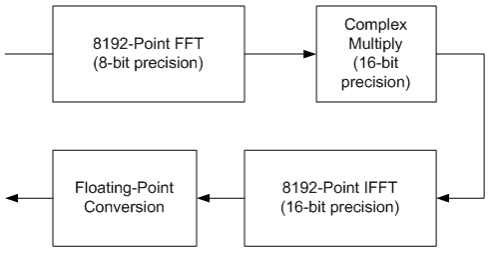

Fig. 2. Range Compression Processing Flow

As data is processed, the amount of storage required grows. The amount of data that can be processed between communications with the host processor is limited by the amount of storage available to the FPGA. For this design, the output will be stored in one 4MB SRAM bank present on the FPGA board. This method allows us to process 64 ranges at a time before communicating results back to the host processor.

B. Azimuth Compression

The azimuth compression computation in the SAR application is very similar to range compression. While the range compression calculations use a single, constant filter during every iteration, azimuth compression uses a much more complex filter that varies for every azimuth. This filter must be calculated on-the-fly due to memory limitations, since the complete azimuth compression filter would be equal in size to the entire image being processed.

The data being sent from the host processor is already in floating-point format. This format is converted to 16-bit fixed-point precision on the FPGA, which is then used as input to an FFT. The output of the FFT is multiplied by a filter generated on-chip. This result is then converted back to the time domain with an IFFT. As a postprocessing step, the magnitude of the complex result is computed and converted back to floating-point. A graphical depiction of the processing steps is shown in the Figure 3.

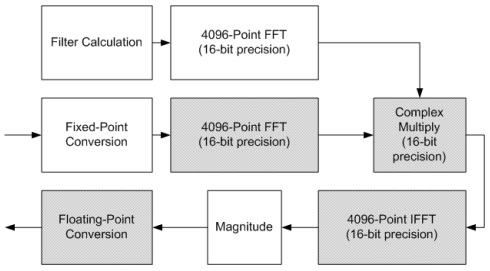

Fig. 3. Azimuth Compression Processing Flow

The range compression core uses 34% of the available slices on the FPGA. The azimuth compression stage uses 65% of the logic slices. The additional FPGA resources in the latter stage are due to an additional FFT core and a filter calculation module. The shaded boxes in Figure 3 show the additonal components needed for azimuth compression. Both designs were tested and can operate at 100 MHz and higher. However, due to limitations of the on-board SRAM, the designs are executed at 66 MHz. Worst-case power estimates from Xilinx tools predict that the range compression core will consume 3.6 W and the azimuth compression core will consume 4.8 W.

V. Fault Tolerance Additions

While TMR is a conceptually simple solution for achieving fault tolerance, the large resource overhead makes alternative solutions very appealing. This section will discuss changes that can be made to range and azimuth compression to achieve fault tolerance with minimal overhead. In addition to the following techniques, scrubbing is also employed to protect the FPGA configuration memory. The FPGA is reconfigured while switching between range and azimuth compression, as well as whenever an erroneous result is detected.

A. Range Compression

As stated earlier, range compression is filtering along the range dimension. The filtering operation can be described as in

Equation 1, where A is the input data matrix, B is the filtered output, F_N is a Discrete Fourier Transform (DFT) matrix, $\vec{x}$ is the filter vector applied to each row of the matrix, and $\bigotimes$ denotes a modified Hadamard product also called an element-wise matrix-vector product.

$$B = \left((A \cdot F_N) \bigotimes \vec{x}\right) \cdot F_N^{-1} \tag{1}$$

To verify the results of the range compression we can augment the data input matrix with an extra row containing the weighted sum of each column. This matrix expansion is accomplished by multiplying A by a vector of checksum weights. A sample weight vector is described in Equation 2. The augmented matrix A_C is commonly called the column checksum matrix.

$$E_N^T = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \tag{2}$$

$$A_C = \begin{pmatrix} A \\ E_N^T \cdot A \end{pmatrix} \tag{3}$$

Since the operations used in the filtering step are all linear, the resulting matrix B_C will contain checksums that are preserved through the operation as shown in Equation 4.

$$B_C = \left(\left(\begin{pmatrix} A \\ E_N^T \cdot A \end{pmatrix} \cdot F_N\right) \bigotimes \vec{x}\right) \cdot F_N^{-1} \tag{4}$$

$$= \begin{pmatrix} ((A \cdot F_N) \bigotimes \vec{x}) \cdot F_N^{-1} \\ E_N^T \cdot ((A \cdot F_N) \bigotimes \vec{x}) \cdot F_N^{-1} \end{pmatrix} \tag{5}$$

This method requires groups of FFTs to be performed together in order to have performance benefits versus traditional TMR techniques. The addition of this fault-tolerant range compression algorithm does not require modification to the FPGA core. The only differences are that the CPU must compute the checksum row and verify it while the filtering is being completed on the FPGA. The AltiVec resources of the PowerPC can be employed to improve performance of the checksum process by computing four elements in parallel every clock cycle.

B. Azimuth Compression

Unfortunately, the fault-tolerant algorithm designed for the range compression calculation cannot be applied to azimuth compression due to non-constant filter $\vec{x}$. Instead, each individual FFT (or IFFT) can be protected using an alternative approach called concurrent error detection (CED). The filter generation and multiplication between the FFT and IFFT can be protected using a self-checking pair (SCP) approach. Two independent filter generators run concurrently and an error can be detected if the results do not agree. Figure 4 illustrates the concept. Although not illustrated, the fixed-point and floating-point conversion functions can be protected using an SCP or TMR approach.

The fault-tolerant 1D-FFT has been thoroughly studied [10], [11], [4], [12]. CED, an efficient and reliable scheme using an ABFT-like method, was first described by Wang and Jha [4]. The general idea behind the CED scheme is to pre-compute

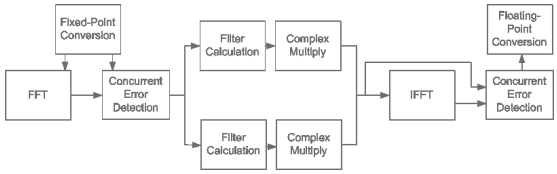

Fig. 4. Fault-Tolerant Azimuth Compression

encoding v_e and decoding v_d row vectors that will make the following equations hold:

$$\vec{v_e} \cdot \vec{x}^T = \vec{v_d} \cdot \vec{X}^T \tag{6}$$

$$\vec{X} = \vec{x} \cdot F_N \tag{7}$$

where F_N denotes the DFT matrix. Such an approach can achieve a low overhead and few false-positives, but the general case requires the computation of new coding vector pairs for each size of DFT. Since range and azimuth compression work with constant-sized vectors, the encoding and decoding vectors are constant and only need to be calculated once at application run-time.

VI. Experimental Results & Analysis

The experimental testbed used for this application consists of a 4-node cluster of AltiVec-equipped 1.42GHz G4 PowerPC processors with 1GB of SDRAM each. Each node contains an ADM-XRC-4 FPGA board connected by a 32-bit PCI bus at 66 MHz, providing 264 MB/s of ideal throughput. This system is a close approximation to the data nodes for the first DM flight system, but with a faster processor. The PCI bus is known to be a limiting factor for performance due to the low-bandwidth connection to the host processor's main memory, so we additionally examine the performance of systems without these bottlenecks through the use of simulation.

The simulative performance results are produced using a modeling and simulation framework designed for analysis of FPGA-based systems and applications, developed at the University of Florida site of the NSF CHREC Center. The simulation framework employs discrete-event models to represent the platform, which are stimulated by application scripts used to characterize the behavior of the application under study. The scripts abstract the application into a sequence of events, supporting the timely simulation of large-scale high-performance and reconfigurable systems. The simulation models are calibrated based on experimental data gained from our experimental testbed, and then extended to model the enhanced systems under consideration [13]. Table II outlines the important features of the systems that will be examined in this section.

The experimental testbed (System 1) uses a faster and more power-consuming processor than the one that will likely be used on the initial DM flight system. Using simulation, System 2 models an architecture that is similar to the hardware of a projected initial flight system configuration, reducing the clock speed of the CPU to 800 MHz. In addition to performance

TABLE II
SYSTEMS UNDER TEST

#	System Description	System Type	CPU Speed (GHz)	CPU-FPGA Interconnect
1	Experimental Testbed	Experimental	1.42	32-bit PCI
2	Flight System	Simulative	0.8	32-bit PCI
3	Testbed with Improved PCI	Simulative	1.42	64-bit PCI
4	Testbed with HyperTransport	Simulative	1.42	1.6 GB/s HyperTransport

results, we also consider the power consumption of the flight system model, since space systems have a particularly strict power envelope.

One simple improvement that may be able to increase performance is upgrading the width of the PCI bus from 32-bit to 64-bit, and therefore increasing the ideal bandwidth to the FPGA to 528 MB/s. System 3 examines the effect on performance caused by changing PCI bandwidth.

System 4 further improves bandwidth from the host CPU to the FPGA by using 1.6 GB/s HyperTransport as the connection. Products from companies such as DRC and XtremeData Inc. allow an FPGA to reside directly in a processor socket with a high-bandwidth connection to the host processor and main memory. A single-board computer (SBC) with two processor sockets would make an ideal platform to use with this type of FPGA architecture. System 4 simulates how this SAR application will perform with the improved bandwidth that such a solution would provide.

A. Single-Node Performance Results

In order to assess the performance of the FPGA-enabled versions of the SAR application, the results of each version are compared to the original microprocessor-only baseline running on the experimental testbed. Efforts were made to optimize the software code whenever possible, as in Section 3. The AltiVec-enabled FFTW library was used computing FFTs. For the FPGA-enabled application, the ABFT error-detection techniques discussed in the previous section are used, and their overhead is included in the following performance results.

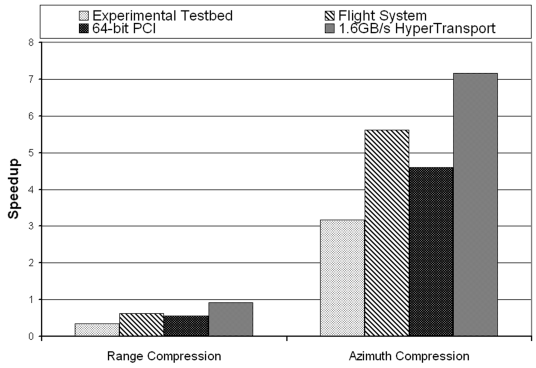

Fig. 5. Range and Azimuth Compression Speedup

Figure 5 illustrates the speedup attained on each test system using FPGAs versus using only the CPU for processing. For each of the systems in this study, the FPGA-assisted version was unable to provide an improvement in performance over the CPU baseline for range compression. This result was largely due to the fact that the optimized FFT libraries for the PowerPC executed in less time than that required to transfer the necessary data across the PCI bus. The azimuth compression core, in contrast, is capable of producing speedups of $3\times$ and higher due to the large amount of computation that is performed in the filter calculation process. With the 1.6 GB/s HyperTransport connection, the FPGA version of azimuth compression can attain $7\times$ speedup while running at only 66 MHz. The results suggest that the I/O bottleneck to the FPGA is very important for these kernels, as increases in performance are approximately proportional to increases in the I/O bandwidth.

TABLE III
FLIGHT SYSTEM POWER CHARACTERISTICS

SAR Core	CPU Power	FPGA Power	Speedup	Perf./Watt
Range	10 W	3.6 W	$0.6\times$	$1.7\times$
Azimuth	10 W	4.8 W	$5.6\times$	$11.7\times$

TABLE IV
FULL SAR APPLICATION SPEEDUP

System	CPU Baseline	Execution Time	Speedup
1	37.56 s	20.66 s	1.8
2	59.96 s	30.56 s	2.0
3	37.56 s	18.22 s	2.1
4	37.56 s	16.31 s	2.3

For the initial flight system configuration, the maximum power consumption of the 800 MHz processor is 10 W. By comparison, the FPGA designs use a maximum of 3.6 W and 4.8 W for the range and azimuth compression cores, respectively. The performance per Watt metric, normalized to the CPU-only baseline, is shown in Table III. Although the FPGA-enabled range compression does not achieve speedup over the CPU, the normalized performance per Watt of the FPGA is almost twice the CPU-only value. For azimuth compression, the performance speedup is accompanied by power savings, leading to high performance per Watt.

The performance for the entire application is shown in Table IV. These results represent the combination of speedup from azimuth compression and the time spent solely using the CPU for the remaining operations. The FPGA design for range compression was not used for this case, since the CPU outperforms the FPGA during range compression on all four systems. Additionally, a 600 millisecond one-time FPGA configuration delay before azimuth compression was included for completeness.

B. Multi-Node Performance Results

In the multi-node SAR case, each node works on an individual patch, with a full SAR image composed of several

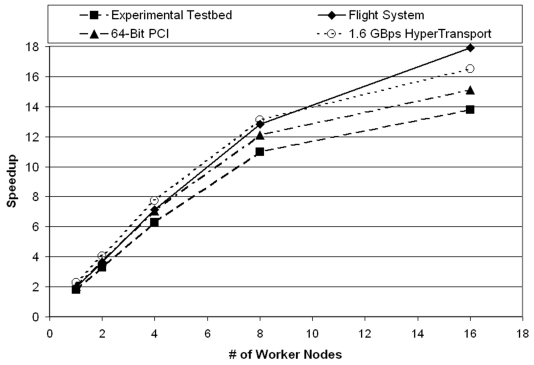

Fig. 6. Multi-Node SAR Speedup

independent patches. The "patched" parallel approach was chosen due to the independent nature of each processing patch. This approach can generate high overall throughput, although the latency of computing any single patch remains high. The parallel program was written using MPI for communication between nodes, and nodes were connected together using Gigabit Ethernet. FPGAs were reconfigured before each new patch in order to scrub the configuration memory. Due to the available FPGA hardware, experimental testing was conducted for system sizes up to 4 nodes. Extrapolation to larger system sizes, as well as the representation of additional systems beyond the experimental testbed, were accomplished using simulation. Figure 6 shows the execution times of our parallel SAR program processing a 16-patch image.

As seen in Figure 6, the 4-node experimental testbed with FPGAs is able to achieve a speedup of $6\times$ over the single CPU-only baseline. The 16-node high-bandwidth HyperTransport model shows the best absolute performance, attaining $16.5\times$ speedup. Meanwhile, the Flight System model is predicted to achieve $18\times$ speedup with 16 nodes due to a slower CPU baseline and better parallel scalability.

The "patched" parallel approach exhibits good parallel efficiency (greater than 85%) up to 4 nodes. However, scalability beyond 8 nodes is very poor. At 16 nodes, the Gigabit Ethernet interconnect does not allow the master node to pass data to every node before the processing is complete, leading to stalled processing nodes. The incremental improvement from 8 nodes to 16 nodes is only 25%. For systems with more than 8 worker nodes, the performance improvement from FPGA acceleration is partially negated by the poor network performance.

VII. Conclusions

The use of non-traditional processing resources such as FPGAs or AltiVec engines is an effective method for increasing performance in systems where computational power is the largest concern. Using a few simple profiling and estimation techniques on an original sequential program, candidate functions for acceleration are easily determined. The azimuth compression calculation, accelerated using an FPGA co-processor on an experimental testbed, was able to achieve a $3\times$ speedup. Simulation was used to overcome technological restrictions in our testbed system and predict the performance of systems with better FPGA interconnect technologies. These systems achieved a projected speedup of $7\times$ when increasing the current platform's bandwidth capabilities. Additional performance gains were possible with multiple nodes communicating using MPI, achieving up to a $18\times$ speedup over a single AltiVec-enabled node.

Future work may explore architectures that support high-memory bandwidth for FPGAs, such as systems from DRC and XtremeData, through experimentation. Since memory bandwidth is the largest limitation from fully realizing the capabilities of FPGAs in a multi-paradigm system, the unique architecture may be useful for many applications. Additionally, we plan to continue exploring the use of algorithm-based fault tolerance for space applications.

Acknowledgements

This work was supported in part by the NMP Program at NASA, our Dependable Multiprocessor project partners at Honeywell Inc., and the Florida High-Technology Corridor Council. Additionally, this work was supported by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422.

References

[1] I. Troxel, E. Grobelny, and A. George. System management services for high-performance in-situ aerospace computing. *AIAA Journal of Aerospace Computing, Information, and Communication*, 4(2):636–656, February 2007.

[2] A. Hein. *Processing of SAR data: fundamentals, signal processing, interferometry*. Springer-Verlag, Berlin, 2004.

[3] J. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, June 1984.

[4] Sying-Jyan Wang and N. K. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Transactions on Computers*, 43(7):849–854, October 1994.

[5] Wai-Chi Fang, C. Le, and S. Taft. On-board fault-tolerant SAR processor for spaceborne imaging radar systems. In *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 420–423, Kobe, Japan, 2005.

[6] Xilinx TMRTool. Product Website, http://www.xilinx.com/ise/optional_prod/tmrtool.htm.

[7] C. Conger, A. Jacobs, and A. George. Application-level benchmarking with synthetic aperture radar. In *Proc. of High-Performance Embedded Computing (HPEC) Workshop*, MIT Lincoln Lab, Lexington, MA, September 2007.

[8] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[9] Xilinx CORE generator. Product Website, http://www.xilinx.com/ipcenter/index.htm.

[10] Y. H. Choi and M. Malek. A fault-tolerant FFT processor. *IEEE Transactions on Computing*, 37(5):617–621, May 1988.

[11] A. L. Narasimha Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10):1304–1308, October 1990.

[12] D. Tao and C. Hartmann. A novel concurrent error detection scheme for FFT networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):198–221, February 1993.

[13] E. Grobelny, C. Reardon, A. Jacobs, and A. George. Simulation framework for performance prediction in the engineering of RC systems and applications. In *Proc. of 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 2007.