PERFORMANCE AND DEPENDABILITY OF RAPIDIO-BASED SYSTEMS FOR REAL-TIME SPACE APPLICATIONS

By

DAVID R. BUENO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2006

To my parents, Ivan Bueno and Lisa Davis

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Alan D. George, for encouraging me to pursue my Ph.D. as well as providing immeasurable help and guidance throughout my graduate academic career. Without the opportunity given to me by Dr. George, as well as his constant words of encouragement, this work would not have been possible. I would also like to thank my other committee members, Dr. P. Oscar Boykin, Dr. Benjamin J. Fregly, and Dr. Janise McNair, for their time and effort dedicated to my research. Thanks also go to all the members of the HCS Lab that helped support my research in various ways, especially Chris Conger, Eric Grobelny, Adam Leko, Casey Reardon, and Ian Troxel.

I would also like to acknowledge Mark Allen, Dr. Dave Campagna, Cliff Kimmery, and Dr. John Samson at Honeywell, Inc. in Clearwater, FL for supporting this research both financially and technically.

I would especially like to thank my parents, Ivan Bueno and Lisa Davis, and my grandparents, Arnold and Mary Ellen Davis, for their endless love and support, and for all being wonderful role models for me to follow throughout my life. I would also like to thank my wonderful brother and sister, Nicholas and Gloria Bueno.

Finally, I would like to give a special thanks to my girlfriend, Shannon Downey, for her undying love, encouragement, support, and understanding over the years I spent on this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PERFORMANCE AND DEPENDABILITY OF RAPIDIO-BASED SYSTEMS FOR REAL-TIME SPACE APPLICATIONS

By

David R. Bueno

December 2006

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

Emerging space applications such as Space-Based Radar (SBR) require networks with higher throughput and lower latency than the bus-based interconnects traditionally used in spacecraft payload processing systems. The RapidIO embedded systems interconnect is one network that seeks to help meet the needs of future embedded applications by offering a switched topology with low latency and throughputs in the multi-gigabits per second range. In this research, we use modeling and simulation to study the tradeoffs associated with employing RapidIO in real-time embedded space systems with high processing and network demands.

The first phase of this research presents a network-centric study on the use of RapidIO for SBR Ground Moving Target Indicator (GMTI) and Synthetic Aperture Radar (SAR) applications. These results provide insight into the optimal partitioning of the SBR algorithms for the RapidIO network, as well as a sensitivity analysis of the algorithms to various network parameters such as clock rate and flow-control type.

In the second phase of this work, we propose several novel fault-tolerant architectures for RapidIO-based space systems and quantitatively evaluate these architectures through a unique combination of analytical metrics and simulation studies. The results from this phase show several promising architectures in terms of achieving a balance between performance, fault-tolerance, size, power, and cost.

In the third and final phase of this research, we extend the previous two phases into a set of detailed case studies with an expanded model set that is augmented with actual RapidIO testbed hardware results. These case studies lead to key insight into interactions and architectural tradeoffs between RapidIO, reconfigurable processing elements, and a shared local memory interface in a realistic embedded architecture.

CHAPTER 1
INTRODUCTION

Space-based computer systems have traditionally collected data using sensors and then beamed data to the ground for the bulk of the processing. The process of downlinking data to the ground is a high-latency, low-throughput operation that severely limits application performance and renders most real-time applications virtually impossible. The need to downlink was partially imposed by the limited processing power and networking capabilities of radiation-hardened components that were built to withstand the harsh environment of space. However, in the last few decades, an increase in the performance of radiation-hardened components has enabled more payload processing to be performed onboard a spacecraft in an embedded computing system. Payload processing systems generally contain multiple processing cards inside of a chassis, connected via some type of custom interconnect built around a combination of point-to-point and bus-based protocols. These custom-interconnected systems often provide a high-performance platform for a particular system, but at the cost of flexibility when applied to other systems and applications. More recently, the use of commercial-off-the-shelf (COTS) components has improved flexibility by allowing system designers to create a range of systems based upon inexpensive and interoperable standard bus-based components such as VMEbus [1] or CompactPCI (cPCI) [2]. However, today's payload processing applications are rapidly outgrowing the throughput and latency capabilities of bus-based architectures. In addition, the scalability and connectivity constraints (i.e., one

sender at a time) of conventional bus-based mediums are equally limiting to the performance of payload-processing networks.

In recent years, switched networks for embedded systems have emerged with the potential to mitigate the limitations of bus-based networks. Such networks have the potential to perform as well as a custom interconnect but with much greater flexibility and lower cost. These switched networks include RACEway [3], SpaceWire [4], and RapidIO [5] and allow hundreds to thousands of megabits per second of bandwidth between ports in the switched fabric, with aggregate bandwidth totals far surpassing that of VMEbus or CompactPCI. For example, a 64-bit, 33 MHz cPCI bus can provide roughly 2 gigabits per second (Gbps) of throughput, while a single 8-bit, double-data-rate (DDR) 250 MHz RapidIO endpoint can provide almost twice this throughput by sampling data on both edges of the clock. Because RapidIO is a switched network, even modest systems will provide tens of gigabits per second of aggregate throughput with many non-blocking links.

While switched networks are not a new concept, many unique issues come into play when designing embedded systems such as size, weight, power, and fault tolerance. There are a large number of variables in the design space, leaving many questions currently unanswered about the ideal methods for constructing high-performance, fault-tolerant, real-time embedded systems built around standard switched interconnects. One extremely important challenge is the design of a network that can fit the needs of multiple applications. For example, how can we best design a network to meet the needs of a throughput-bound application as well as a latency-bound application? Network topology certainly plays an important role in performance, but the mapping of the application to

the particular system and network architecture is equally important. It is important to have a high level of understanding of the network in question in order to answer all the questions related to a given application or class of applications. There are also numerous challenges from a fault tolerance standpoint. While typical bus-based systems simply add redundant buses for fault tolerance, it is unclear how to best design a fault-tolerant switched network for space. At one end of the spectrum, the entire network might be duplicated, while at the other end it might be possible to just build a few redundant paths into the network—possibly not even requiring a single extra switch.

In order to keep the scope of this work manageable, RapidIO is the only network we study in great detail. RapidIO was chosen as our primary focus due to heavy (and growing) industry support and its emphasis on high performance and scalability. RapidIO is an open standard, which is steered by the RapidIO Trade Association [6]. The initial public version of the standard was released in 2002, and the past several years have seen many products come to market such as RapidIO switches, FPGA cores, and bridges to other networks such as PCI. The relative maturity of the RapidIO specification and its increasing level of industry adoption make it an ideal choice for this study. It is important to note, however, that much of the general insight gained regarding RapidIO networks will also apply to other current and even future point-to-point embedded systems interconnects. Later in this study, we will briefly examine some of RapidIO's competitors and their similarities and differences with RapidIO.

With so many tradeoffs in the RapidIO design space, it is impossible to thoroughly evaluate all of the relevant tradeoffs using hardware prototypes. The high costs associated with building space-based computing systems as well as the time constraints

typically faced in a space program make it imperative to quickly evaluate the worthiness of a particular architecture for a given application very early in the design process. As part of this research, we develop a suite of simulation models for rapid virtual prototyping of space systems based on RapidIO networks. This model set is generalizable and easily configurable to model any RapidIO implementation. The models are also highly modular, allowing parts to be swapped out, or other components to be built around the network models to model an entire space system. We use the RapidIO models to study several emerging space applications in order to gain insight into the design tradeoffs and considerations for building RapidIO networks for real-time embedded space systems. In addition, we closely examine the applications under study and evaluate tradeoffs in mapping the specific applications to the RapidIO-based systems.

In the first phase of this research, use our simulation models to study the Ground Moving Target Indicator (GMTI) and Synthetic Aperture Radar (SAR) algorithms for Space-Based Radar (SBR) running on simulated RapidIO networks. We examine a wide range of system configurations as well as algorithm mappings and parameters in order gain insight into the tradeoffs in mapping GMTI and SAR onto a RapidIO network. Performance, scalability, and efficiency are among the important factors considered as we attempt to determine the sensitivity of the GMTI and SAR algorithms to various RapidIO network parameters such as clock rate, number of switch ports, and flow-control type. Since GMTI and SAR have conflicting needs in terms of hardware and system requirements, it is a great challenge to balance the system parameters in order to provide optimal performance for both algorithms without wasting resources when running one application or the other. A very important outcome of this phase of work is a thorough

understanding of the tradeoffs involved in performing both real-time GMTI and SAR on the same system in an alternating fashion.

In the second phase of this research, we investigate the options for creating fault-tolerant RapidIO networks for space. An in-depth literature search was performed to study the spectrum of options for building fault-tolerant networks out of moderately-sized (approximately 6- to 16-port) switching elements, and this phase of research seeks to adapt some of these commonly-used techniques to space systems, such as alternate-path routing and extra-stage networks. We use analytical metrics as well as an expanded simulation environment to study tradeoffs in fault tolerance, performance, size, and power between the network configurations under study. These network configurations and techniques are also compared with the more traditional approaches of simply duplicating the entire network or portions of the network. In addition, combinations of approaches are examined in order to find the optimal methods for bringing an appropriate level of both fault tolerance and performance to a given system, while staying within reasonable size and power constraints.

The third and final phase of this research brings the lessons learned in the first and second phases together into a set of three detailed, system-level case studies in order to explore the impact of the RapidIO network on a real-world system that includes reconfigurable processing elements, memory elements, and external data interfaces. In particular, the interface between the RapidIO network and the memory interface of the processing elements is carefully examined. This memory interface is a likely bottleneck in systems that have very high incoming data rates being written to a memory that is also shared for processing purposes. The first case study revolves around a detailed

implementation of the GMTI algorithm, while our second case examines methods for improving the latency and jitter of critical RapidIO traffic in systems running GMTI. We conclude with a case study on global-memory-based Fast Fourier Transform (FFT) processing, which is a major element of the SAR algorithm we study in the first phase of this research. Our simulation model set is greatly expanded for this phase of work, with detailed processor and network memory access simulation in addition to the previously studied network traffic simulation. While the system and applications used for this phase of work are based on actual implementations in our RapidIO hardware testbed, one of the key advantages of our simulation-based approach is that we can adjust parameters to model hardware that we do not have available in the testbed, in terms of performance, scale, and features.

In this study, Chapter 2 provides background on space-based payload processing, as well as background and related research on embedded networks for space with an emphasis placed on RapidIO. Chapter 3 describes our RapidIO simulation environment that is used as the basis for the simulation experiments in all phases of this research. Chapter 4 details the results of the study of GMTI and SAR algorithms performed over RapidIO-based space systems and draws conclusions on the optimal system and network configuration for systems capable of both GMTI and SAR. Chapter 5 presents our study on high-performance, fault-tolerant space systems based on RapidIO, proposing several promising architectures in terms of achieving a balance between performance, fault tolerance, size, power, and cost. Chapter 6 describes our RapidIO system-level case studies and describes key insight developed into tradeoffs regarding the shared memory interface between FPGA-based processing elements and RapidIO. Finally, Chapter 7

provides a summary of this research as well as conclusions and contributions to the literature.

CHAPTER 2
BACKGROUND AND RELATED RESEARCH

In this chapter, we present background and related research on space-based payload processing and networks for embedded systems. We also justify the selection of RapidIO as the focus of this research and present detailed background on the RapidIO embedded systems interconnect.

Space-Based Payload Processing

Today's space-based payload processing systems typically consist of several blade-like cards residing in a compact chassis onboard a spacecraft. While the main processing boards are often custom solutions with Application-Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs) or Field-Programmable Gate Arrays (FPGAs), system control functionality is usually accomplished using a COTS Single-Board Computer (SBC) card based on an architecture such as the PowerPC [7]. Other cards may reside in the chassis as well for the purposes of power supply, network resources, or redundancy management.

Space is an extremely hostile radiation environment, and payload processing systems are asked to perform a number of complex mission tasks with limited resources. Spacecraft are frequently exposed to high-energy particles that can damage the processing system and dramatically affect the correctness of the results or the health of the system. Memory devices in the circuits of the processing system are subject to Single-Event Upsets (SEUs) that occur when these high-energy particles pass through the spacecraft and their resulting transient currents cause one of the memory cells of a chip in

the system to change state (i.e., a bit-flip). To mitigate the potential faults introduced by SEUs, the most common approach is to build parts that are fabricated using a special radiation-hardened process. Radiation hardening reduces the susceptibility to radiation, electrical, and magnetic interference, and many radiation-hardened chips are based off of COTS equivalents. However, the hardened variants tend to lag several technology generations behind the cutting-edge commercial products due to the extensive development and testing required in producing a radiation-tolerant design as well as limitations in the special processes used to fabricate the chips.

In addition to radiation-hardened fabrication processes, redundancy is also extensively used in space systems to provide fault tolerance at many levels. At the circuit level, a single output bit could be replaced by three separate bits with voting logic to determine the result. This type of technique is known as Triple-Modular Redundancy (TMR). TMR may also be accomplished at the chip level, where completely separate chips (possibly true COTS chips with limited or no radiation hardening) may work in lockstep to compute the same calculation and vote on the result. In addition to TMR-type redundancy for transient (soft) faults, other redundant components are usually integrated into a space system for use in the event of a hard fault. A hard fault has indefinite duration and generally reflects that an irreversible physical change has occurred in one of the components, potentially rendering it useless. In case of a hard fault, fault tolerance is often handled through the use of a cold spare. A cold spare is an un-powered hardware device that may be powered on in the event of a permanent failure of some system device. This technique is often accomplished in a space system at the card level, where, for example, a damaged processor card may be deactivated and a spare one may be

powered up to take its place. The same type of cold-spare mentality is also generally used in networks for space systems, where a completely separate backup network may be employed in the event there is a failure in the primary network.

Embedded Systems Interconnects

Prior to the arrival of VMEbus [1], most space systems were based around a custom interconnect designed on a program-by-program basis with little component reuse. The VMEbus was standardized by IEEE in 1987 and is an open architecture that became widely accepted in both the commercial and military communities. The bandwidth of this bus is modest by today's standards, at approximately 320 megabits per second (Mbps). Over the past two decades, many missions have been flown using VMEbus-based systems, giving the technology an impressive resume for space use.

In the last decade, space payload-processing architectures have begun to move from VMEbus architectures to CompactPCI (cPCI) architectures [2]. CompactPCI is based on a superset of the commercial PCI standard, and provides throughputs of approximately 1 Gbps for a standard 33 MHz/32-bit implementation and increasing performance for wider or faster (i.e., 64 MHz or 64-bit) versions of the PCI specification. The cPCI bus extends the PCI specification by providing a rugged mechanical form factor as well as support for additional peripheral cards. CompactPCI has also found widespread acceptance in commercial and military circles, and there are many missions that have flown or are scheduled to fly incorporating cPCI technology, including Mars Sample Return, Europa Orbiter, Pluta/Kuiper Express, and Deep Impact [8].

Over the last decade, several switched embedded systems interconnects have arrived that provide major performance advantages for space systems previously limited by bus-based networks. Mercury Computer Systems' RACE [3] architecture is based on

the RACEway Interlink standard for high-performance switched communication in real-time systems. RACE is a circuit-switched architecture providing approximately 1.3 Gbps of port-to-port bandwidth. Later, Mercury introduced their improved RACE++ architecture, which upgraded the performance to 2.1 Gbps while retaining backward compatibility with the original RACE architecture. While Mercury's RACE architectures did gain some industry and military acceptance, they are not as widely used as VMEbus and cPCI and are more commonly included in Mercury's COTS embedded system product lines. These products essentially are an "embedded system in a box," with combinations of General-Purpose Processors (GPPs), DSPs, or SBCs connected over a high-performance network. In the last several years, Mercury has been a major proponent of RapidIO and a driving force behind the standard, and some of Mercury's more recent systems have actually used RapidIO as the interconnect inside the box.

HyperTransport [9] and InfiniBand [10] are two other interconnects that have been introduced this decade and have received attention as potential interconnects for embedded processing systems. While each of these networks provides potentially higher throughput and major technology advances over the previously described networks, they have found their niche in other areas of computing. HyperTransport has become widely used as the main intra-system interconnect for newer AMD computer systems, and this use likely will remain the main application for HyperTransport. While InfiniBand was initially touted as an all-purpose open network, InfiniBand's main application realm is in High-Performance Computing (HPC) and it is one of the most popular interconnects for COTS clusters. Recently, InfiniBand standards have evolved to include backplane and packaging definitions for inside-the-box embedded applications. These embedded

flavors of InfiniBand use Low Voltage Differential Signaling (LVDS) [11], which is the same technology used by RapidIO and allows very high transfer rates over differential pairs of copper wires with low power consumption. However, InfiniBand is a serial specification, and lead times will be longer before this technology is ready to be used in space systems due to the high clock rates required for high-performance serial interconnects. RapidIO's specialization and head start in the embedded systems market are important factors in the selection of RapidIO for the focus of this study.

SpaceWire [4] is another technology to emerge early this decade and is a communications network specifically for use on spacecraft to connect high-data-rate sensors, memories, and processing units. The SpaceWire standard is defined by the European Cooperation for Space Standardization, and SpaceWire is currently being used on ESA and NASA missions. SpaceWire links are LVDS point-to-point serial links capable of up to 400 Mbps, bi-directional, full-duplex communication. SpaceWire is a packet-switched technology, and a SpaceWire network may be composed of links, nodes, and routers (switches). Preliminary versions of SpaceWire have been used on multiple missions including Mars Express, Rosetta, and STEREO, and SpaceWire is being designed into many current and future missions. With all of the similarities to RapidIO, SpaceWire may initially appear to be a competitive technology with lower data rates. However, SpaceWire links are commonly used in outside-the-box communication with spacecraft, and SpaceWire may actually be a complimentary technology that stands alongside RapidIO in future systems due to its ability to reach relatively high data rates over moderate transfer distances. For example, SpaceWire is capable of operating at

200 Mbps over distances up to 10 m, with slower implementations capable of reaching even longer distances.

The Advanced Switching Interconnect (ASI) [12] may be the biggest competitor to RapidIO's place in the embedded systems market in the coming years. ASI was originally a spinoff of PCI Express (PCIe) but has since evolved into its own specification with its own special interests group (the ASI-SIG). ASI shares the physical layer features of PCI Express but adapts the link layer and transport layer features specifically for embedded data processing systems. ASI is strictly an LVDS serial specification but has options for up to 32 lanes, providing extremely high-throughput serial configurations up to 80 Gbps. Because ASI is based upon PCI Express, it is a significantly more complicated specification than RapidIO and there are many more opportunities for overhead to be introduced in the protocol and the packet format. Because of its ties to PCI, ASI's most useful applications may be in systems already using PCI-based components. ASI products have not yet arrived in the embedded systems market, but will be arriving this year.

RapidIO Overview

RapidIO was chosen as the focus of this work due to its high performance and growing industry acceptance, with a mature specification and products that have been available since 2003. This section presents related work on RapidIO and provides an overview of the technology.

RapidIO Related Work

There has been very limited academic research on RapidIO performed outside the University of Florida. In 2001, researchers at Drexel University created C++ simulation models of RapidIO and used them to study throughput and latency performance for a

simple RapidIO network configuration [13]. The basic set of results obtained provided a very early look at the performance of RapidIO prior to actual hardware being available. In 2003, researchers at Strathclyde University in Glasgow, UK used the network simulator NS-2 to create discrete-event RapidIO network models to study the performance of traffic classes through a generic switch device that is composed of RapidIO components (e.g., using RapidIO devices to build an IP router) [14]. The work described in this study differs quite significantly from these two projects, as our modeling environment allows us to model and study complete RapidIO systems—including processing and memory elements—running real-world applications. This flexible approach enables us to collect a much wider and deeper variety of detailed results on the applications under study.

RapidIO Background

RapidIO is an open standard for a high-bandwidth, packet-switched interconnect that supports data rates up to approximately 60 Gbps. The main reference for RapidIO is the set of formal specifications published by the RapidIO Trade Association [15-23]. RapidIO is available in both parallel and serial versions, with the parallel versions of RapidIO supporting much maximum higher data rates and the serial versions offering a much lower pin-count but needing a higher clock rate to equal the throughput of a parallel implementation. RapidIO uses the Low-Voltage Differential Signaling (LVDS) technique to minimize power usage at high clock speeds, and therefore is appropriate for use in embedded systems.

One defining characteristic of the RapidIO protocol is that it is designed with simplicity in mind so that it should not take up an exorbitant amount of real estate on embedded devices. As shown in Figure 2-1 [24], the RapidIO protocol is comprised of

three layers: the logical layer, the transport layer, and the physical layer. The logical layer is responsible for end-to-end interaction between endpoints (it is not present in switches) and supports at least three different variants: a version suited to simple I/O operations, a message-passing version, and a version supporting cache-coherent shared memory. The logical I/O layer supports simple read/write operations from processor to processor, while the logical message-passing layer allows processing elements to communicate using a message-passing paradigm. A logical specification for globally-shared memory systems is also available that supports cache-coherent remote memory access, although we know of no current implementations for this specification. In this research we use both the logical message-passing and logical I/O layers in simulation and examine the performance of each, as well as discuss practical implementation issues associated with the two protocols. Our system validation experiments are performed using the logical I/O layer, as this is the logical layer present in our RapidIO hardware.

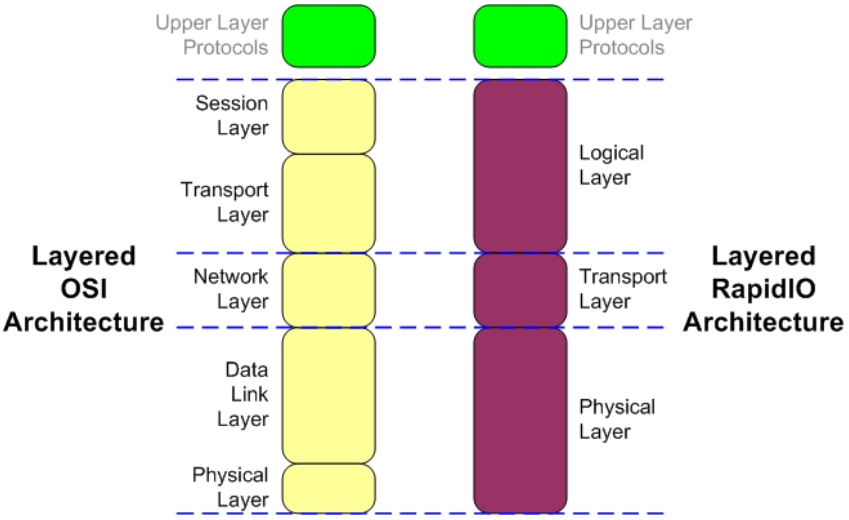


Figure 2-1. Layered RapidIO architecture versus OSI [24].

The logical I/O layer allows processing elements with RapidIO endpoints attached to read and write memory of other devices in the system with RapidIO endpoints. In

addition to standard read and write operations, there is also a low-overhead “streaming write” that can be used when the payload is a double-word multiple, as well as several additional operations such as atomic test-and-swap. The maximum payload that may be read with a single read packet or written with a single write packet is 256 bytes. By default, writes are responseless, although the option exists to have each write request packet confirmed with a logical-layer response packet.

The message-passing layer provides applications a traditional message-passing interface with mailbox-style delivery. The message-passing layer handles segmenting messages of up to 4096 bytes into packets (each RapidIO message-passing packet also has a maximum payload size of 256 bytes). Each message-passing transaction is composed of two events: a request and a response. Responses are required in the message-passing specification, and acknowledge the receipt of each message packet.

The RapidIO transport layer is a simple specification defining such rules as packet routing guidelines, maximum number of nodes in a system, and device ID format. This layer is a single common specification that may be used with any current or future logical layer or physical layer specifications.

The two primary physical layer specifications for RapidIO are the parallel and serial physical layers [16, 18]. The parallel version is geared towards applications that need very high amounts of bandwidth over short distances and allows link widths of 8 or 16 bits. The serial version of RapidIO is geared towards forming backplanes over somewhat longer distances than the parallel version and provides a much-reduced pin count. For example, an 8-bit parallel RapidIO endpoint requires 40 pins, while a 16-bit endpoint requires 76 pins. In contrast, a single-lane serial RapidIO endpoint requires

only 4 pins, and a four-lane serial version requires only 16 pins. However, in both cases the pin count requirement of a RapidIO endpoint is significantly less than that of a typical bus connection [5]. It is important to note that for RapidIO switches, each port will require at least as many pins as described here. Both the parallel and serial versions are bidirectional, and parallel RapidIO data is sampled on both rising and falling edges of the clock (DDR).

In RapidIO, there are two main types of packets: regular packets and special control symbols. Regular packets contain data payload and other application-specific information associated with the logical and transport layers. Control symbols exist in order to ensure the correct operation of the physical layer and support flow control and link maintenance. In the parallel versions of RapidIO, control symbols are embedded inside packets in the LVDS signal. This approach allows control symbols to be given the highest priority, and the simple protocol specification for control symbols reduces the amount of additional hardware needed to produce the control symbols and support their functions.

Error detection is supported directly in the physical layer through the use of cyclic redundancy checks (CRCs) in regular packets and inverted bitwise replication of control symbols. The physical layer specification has protocols designed for error detection and recovery, and error recovery is accomplished through the retransmission of damaged packets. As with the other elements of the RapidIO protocol, the error detection, correction, and recovery protocols are simple but comprehensive and require limited silicon real-estate.

The RapidIO specification for the physical layer provides two options for flow-control methods: transmitter-controlled flow control and receiver-controlled flow control. Both flow-control methods work with a “Go-Back-N” sliding window protocol for sending packets. In RapidIO, flow control is performed between each pair of electrically linked RapidIO devices. The specification requires receiver-controlled flow control to be available in all implementations of RapidIO devices, while transmitter-controlled flow control may be implemented optionally. In receiver-controlled flow control, a receiver of a packet notifies the sender of whether that packet has been accepted or rejected via control symbols. The sender makes no assumptions about the amount of buffer space available for incoming packets and relies on negative acknowledgements and retransmissions to ensure correct packet delivery. In transmitter-controlled flow control, the receiver notifies its link partner how much buffer space it has available by embedding this information into control symbols. The sender will never send more packets than the receiver has space. In this manner, the flow-control method will work more efficiently. Figure 2-2 illustrates the operation of each flow control method.

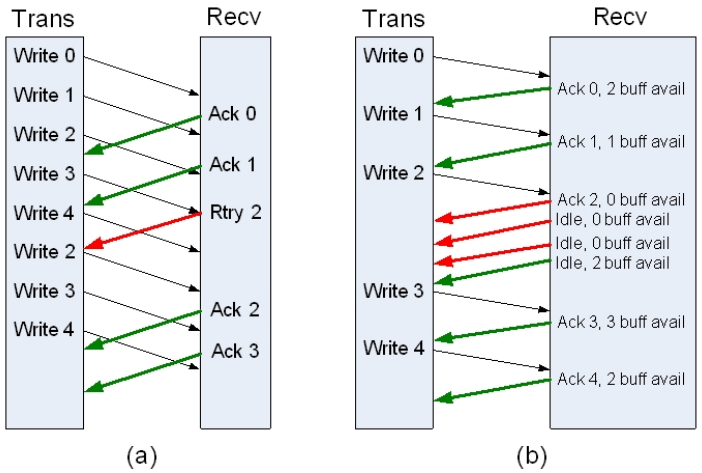


Figure 2-2. Types of RapidIO flow control. A) Receiver-controlled flow control. B) Transmitter-controlled flow control.

Since the physical layer supports four priority levels, the RapidIO specification suggests allocating buffer space to each priority using a simple static threshold scheme. Our RapidIO endpoint models use this method of buffer allocation (based on number of packets of a given priority in the buffer), while our central-memory switch models use a slightly more advanced scheme with per-port configurability. Other implementations may use dedicated physical buffers for each priority. The exact method of managing buffer space in a production system is left up to the implementer.

Xilinx, Tundra, Redswitch, Motorola, GDA Technologies, and Praesum are a few of the vendors that offer RapidIO products, and have made available a number of data sheets, whitepapers, and presentations. These documents provide basic performance figures, but more importantly topical discussions of some of the limitations and challenges of the RapidIO protocol. For our simulations, system parameters such as packet assembly time, packet disassembly time, and input/output queue lengths were initially taken from these data sheets [25-33] and then later tweaked during calibration to match our experimental RapidIO testbed.

CHAPTER 3
SIMULATION ENVIRONMENT OVERVIEW

In this chapter, we describe the simulation tool selected for this project, as well as the important aspects of the mid- and high-level models created for this project. Low-level models such as individual primitives designed for specific purposes (e.g., calculating a packet's priority) are generally considered beyond the scope of this study.

Modeling Tool

Mission-Level Designer (MLD) from MLDesign Technologies Inc. was selected as our primary modeling tool for this project [34]. While MLD is an integrated software package with a diverse set of modeling and simulation domains, the discrete-event domain was used in this study. The tool allows for a hierarchical, dataflow representation of hardware devices and networks with the ability to import finite-state machine diagrams and user-developed C/C++ code as functional primitives. The capability of MLD to model and simulate computer systems and networks in the discrete-event domain is based upon its precursor, the Block-Oriented Network Simulator (BONeS) Designer tool from Cadence Design Systems. BONeS was a discrete-event simulator developed to model and simulate the flow of information represented by bits, packets, messages, or any combination of these. An overview of BONeS can be found in [35].

MLD may be used from a top-down or bottom-up system design modeling approach. For this project, we used a combination of these two approaches. The RapidIO endpoint models were developed from the top down, first creating the logical, transport, and then the physical layer models. However, from a system-wide standpoint,

we took a bottom-up approach, first developing RapidIO endpoint and switch models, and then the processor, algorithm, backplane, and full-system models.

Model Library

In this section, we describe the model libraries developed for this research. We divide the models into three categories: RapidIO network models, end-node models, and high-level system components.

RapidIO Models

Our RapidIO models were created using a layered approach, corresponding to the three layers of the RapidIO protocol. The RapidIO logical layer is responsible for end-to-end communication between system nodes. Our logical layer RapidIO models are responsible for modeling packet assembly and disassembly, memory copy time, and other high-level aspects of the RapidIO protocol. The models include many adjustable parameters that can be tweaked to represent virtually any specific RapidIO setup. We have created models for both the RapidIO message-passing logical layer and RapidIO I/O logical layer. The message-passing layer allows communication between nodes using an explicit message-passing paradigm, while the I/O logical layer allows communication between RapidIO nodes using memory-mapped reads and writes. The two logical layer models may be easily swapped for one another, with no impact on the lower-layer models in the system. Of course, any layers above the logical layer (such as applications using the RapidIO network) will have to change the way they interact with the logical layer depending on which layer is used.

Some of the important parameters adjustable in the logical layer models are:

- Packet disassembly time: Time it takes to extract the payload from a received packet or process a response

- Response assembly time: Time it takes to assemble a response (when required) once a request is received and disassembled
- Response upgrade: In the RapidIO protocol, responses always have a higher priority than their corresponding requests (in order to free resources and prevent deadlocks). Determines how many priority levels a response is upgraded above its corresponding request

The RapidIO physical layer models are responsible for point-to-point behavior such as flow control, error detection and correction, and link-level signaling. These models carry the bulk of the functionality and complexity of our RapidIO model library. As the RapidIO transport layer is extremely simple, we have merged it with the physical layer models. We chose to use a packet-level granularity for our models (as opposed to a cycle-level granularity) in order to keep simulation times reasonable while still providing accurate results. Our models ignore signal propagation delay, because the short distances involved in an embedded space system would not have a noticeable impact on the overall results of the simulation. The parameters of our models allow us to tailor them to represent either the parallel or serial RapidIO physical layers.

Some important parameters adjustable in the physical/transport layer models are:

- Physical layer clock: Support for arbitrary clock speeds
- Physical layer link width: Selects the width of the RapidIO interface
- Input/output buffer sizes: Number of packets that an endpoint can buffer in its incoming and outgoing interfaces
- Physical layer priority thresholds (4): Correspond to the watermark levels used in a static priority threshold scheme, which determine if a packet of a certain priority may be accepted based on how much buffer space is currently occupied
- Flow-control method: Selects transmitter- or receiver-controlled flow control

Our RapidIO switch models are designed to represent a TDM-based central-memory switch, and are constructed in a very modular fashion, making it easy to create

switch models with an arbitrary number of ports. The switch central memory is composed of a small (8 KB to 16 KB), high-speed buffer capable of meeting non-blocking throughput requirements for all eight switch ports. The TDM bus for buffer memory access is modeled using near-cycle-level accuracy, but the TDM delay is obtained in a single calculation so as not to negatively impact simulation time. The switch central memory may be accessed by each port during its dedicated time slice of the TDM window. During this time slice, an input port may write data to the central memory or an output port may read data from the central memory. The choice of a central-memory switch was made due to the potential satellite designs we are using as the basis for our models, and the simplicity of the RapidIO protocol maps well to a central-memory-based design. However, other switch models can easily be developed as needed.

The switch models are created using a combination of existing physical and transport layer components, and switch-specific components. As mentioned previously, the logical layer is not present in RapidIO switches. Routing is accomplished based on a routing table file that is specific to each switch and routes each destination node ID to a specific switch output port. In addition, our RapidIO switch models support dynamic routing (not officially part of the RapidIO standard) in which a packet marked for a given destination ID may be routed to one of a group of possible output ports The dynamic routing feature may be selected to perform in round-robin or pure random fashion, or may pick the valid output port from the list with the least number of packets buffered for transmission.

Important parameters specific to our central-memory switch are listed below:

- TDM window size: Size of the complete TDM window (for all ports) in units of time

- TDM bytes copied per window: Number of bytes that may be copied to/from central memory during a port's TDM time slice of the TDM window

- TDM minimum delay: This value is the minimum amount of time to wait before starting to copy a packet to central memory. This parameter can be used to model time taken to examine packet headers or other fixed delays.

- Central memory size: Total number of bytes for packet buffer space in the switch central memory

- Cut-through routing: Cut-through routing may be turned on or off in the switch by adjusting this parameter. Cut-through routing eliminates most of the delay of copying a packet to switch memory by sending a packet immediately (subject to the memory read and write latencies) to the next "hop" if no other packets are currently queued to be sent out of the packet's corresponding output port. The alternative to cut-through routing is store-and-forward routing.

- Queue configuration file: This file allows the user to specify on a port-by-port basis the number of packets of each priority that may be buffered for each output port in the switch. It also allows specification of the maximum total number of packets that may be buffered for a port. If switch central memory is full, incoming packets are rejected regardless of the contents of the queue configuration file.

End-Node Models

The processing components in our system are modeled as scripted components in order to provide us the most flexibility for representing various algorithms. Each processor reads from a script file that contains the instructions for all the nodes in the system. These are simple instructions such as "compute 10000" telling the processor to compute for 10000 ns or "send 200 4 1" telling the processor to send 200 bytes to processor ID 4 at priority 1. We create scripts for the GMTI and SAR algorithms using custom Python code that takes algorithm settings as input parameters and generates simulation script files as outputs.

The scripted approach allows us to easily implement a wide range of applications, microbenchmarks, or kernel benchmarks to exercise our RapidIO system-level models.

In addition, this approach allows us to model any type of processing element we want as long as we have data related to the actual processing time of the application under study.

A complete list of the instructions used in our experiments is given below:

- Send: Send data to another node using RapidIO logical message-passing messages.
- Recv: Block until a given amount of data is received from a given node.
- RIO_Read: Read a specified amount of data from another specified node using RapidIO logical I/O read operations.
- RIO_Write: Write a specified amount of data to another specified node using RapidIO logical I/O write operations.
- Comp: Compute for a set amount of time (data may still be received while computing).
- Mark/Wait: When the Mark instruction is encountered, the script processor records the simulated time when the Mark was reached. When the corresponding Wait instruction is reached, the script processor then delays until the specified amount of simulated time has passed since the Mark was recorded.
- SDRAM_Read: Read a specified amount of data from local SDRAM into high-speed on-chip SRAM.
- SDRAM_Write: Write a specified amount of data to local SDRAM from high-speed on-chip SRAM.
- SDRAM_Read_Block: Block until a specified amount of data has been read from SDRAM.
- SDRAM_Write_Block: Block until a specified amount of data has been written to SDRAM.

The last four instructions listed (SDRAM instructions) are used exclusively in the third phase of this research, where our model set has been expanded to include modeling of the shared memory subsystem between a processor and RapidIO network. Chapter 6 will provide further details on the memory subsystem under study.

System Components

Other than the complete system-level models, the models discussed in this section are the highest-level models used in this work. These components are the direct building blocks from which our full systems are constructed and are composed of numerous additional blocks in underlying levels.

Our baseline processor board models consist of a RapidIO switch connected to several processing elements. For systems with eight-port switches, we connect four processing elements to the on-board switch, leaving four connections from the switch to the backplane. This approach allows full bandwidth to be simultaneously delivered from the backplane to each of the four processors simultaneously (provided there is enough bandwidth available on the backplane). The type of processing elements modeled may be determined based on the parameters of the processor script. Figure 3-1 shows a diagram of the four-processor board model with ASIC processing elements.

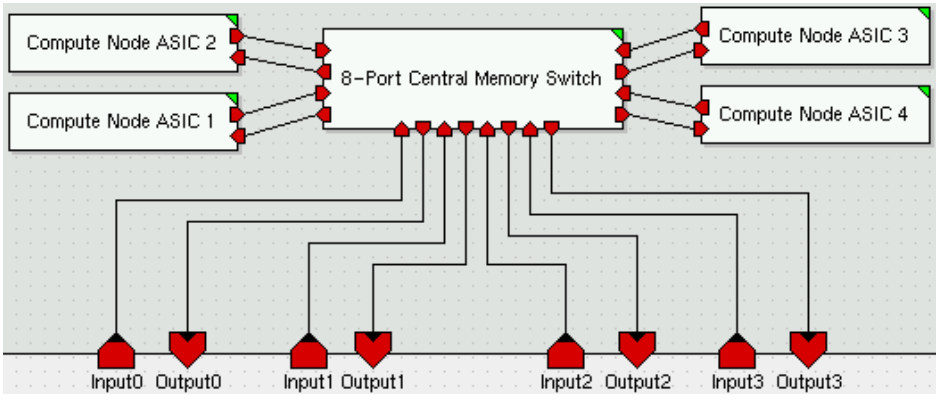


Figure 3-1. Four-processor board model.

The backplane models are comprised solely of eight-port RapidIO switches. In an embedded system (especially one for space), volume, weight, and power are at a premium, making it essential to build a high-performance backplane using as few switches as possible. Early backplane designs developed for this research attempted to

form a backplane with a small number of switches in a mesh, ring, or other regular and irregular topologies. However, these backplanes were not able to provide the necessary bi-section bandwidth and non-blocking communication for the GMTI algorithm. When run over these backplanes, the GMTI simulations would fail to complete, with large amounts of data being sourced into the system and insufficient bandwidth to get the data to the end nodes on the real-time schedule.

To meet the needs of GMTI and SAR, we created a Clos-based [36], non-blocking backplane architecture providing full bi-section bandwidth. This architecture allows any board to communicate with any other board through any single backplane switch. Figure 3-2 shows a four-switch backplane, created using four 8-port switches and supporting a total of eight boards. The Clos backplane serves as our baseline architecture throughout this research, and we examine several variations to the design depending on the phase of research and the experiment configuration.

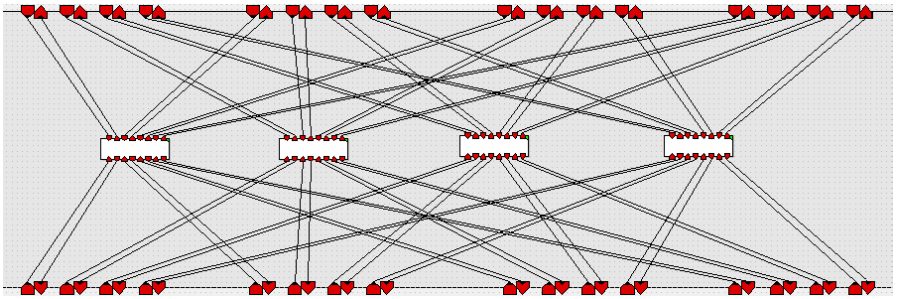


Figure 3-2. Four-switch backplane model.

The data source model is the source of all data in the system and is an important part of our design constraints. Spacecraft sensor arrives at the data source, which may also have a large global memory depending on the system configuration. Our data source models have four RapidIO endpoints, each connected to a model element that can service memory read and write requests to the global memory. In the case of a message-passing

implementation of an algorithm, the data source must also be equipped with a memory controller to know when to send data to specific nodes. The memory controller functionality is modeled through scripting that is similar to the scripting used for the processor models. Figure 3-3 displays a system-level view of our baseline system model with seven processor boards and a four-port data source connected.

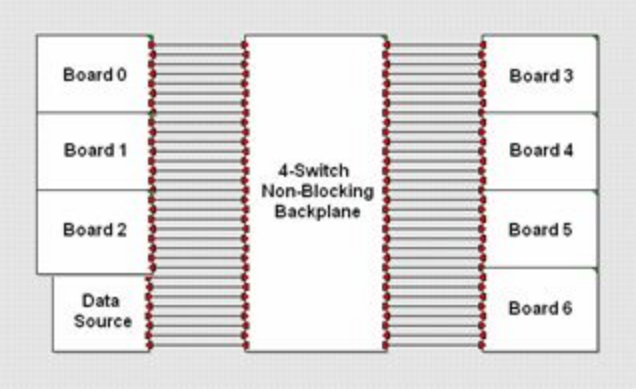


Figure 3-3. Baseline system architecture.

For the third phase of this work, we also developed a system controller component and integrated it into the baseline system architecture via a single RapidIO link to the backplane. In these systems, nine-port backplane switches are used rather than eight-port switches, and the system controller occupies one of the extra ports of one switch. The remaining three extra ports are left free for redundant components or additional external interfaces to the RapidIO network. The primary purpose of the system controller in our models is to generate small data (32-256 bytes) at regular intervals destined for each of the processing nodes. In a real-world system, this data may contain system health and status information or global timing information that must be delivered to end nodes within a latency-sensitive timeframe.

CHAPTER 4
RAPIDIO-BASED SPACE SYSTEMS FOR GROUND MOVING TARGET INDICATOR (GMTI) AND SYNTHETIC APERTURE RADAR (SAR) (PHASE 1)

In this chapter, we present the results of a study on the use of RapidIO-based space systems for real-time GMTI and SAR space-based radar algorithms. We provide background and related research on GMTI and SAR and then present a comprehensive set of simulation experiments and results for each of the algorithms. In addition, we use simulation and analysis to evaluate considerations for performing both GMTI and SAR on the same system.

Introduction

Space-based radar is a processor- and network-intensive set of applications that presents many unique system design challenges. In this phase of research, we consider the use of the RapidIO embedded systems interconnect to solve the problems associated with the high network bandwidth requirements of real-time Ground Moving Target Indicator (GMTI) and Synthetic Aperture Radar (SAR) applications in satellite systems. Using the simulation environment described in Chapter 3 of this study, we consider architectures designed to support GMTI and SAR by exploring various architecture options for the RapidIO interconnect, as well as algorithm structure and parallel decomposition methods for GMTI and SAR.

The GMTI and SAR algorithms are each composed of several subtasks that rely heavily on signal processing. GMTI is meant to detect and track movements of ground targets from air or space, while SAR is used to take a detailed picture of a target once it

has been identified. Due to the communication delays associated with space-to-ground downlinks and the large size of data associated with space-based radar, it is advantageous to process the incoming radar data onboard the satellite in a real-time manner. While previous systems lacked the processing and network capabilities to allow real-time payload processing, a RapidIO-based system with modern processing components can support these real-time algorithms. For the purposes of space-based radar, GMTI data sets are smaller and must be processed at more frequent real-time intervals (e.g., 800 MB of data to process in a real-time deadline of 256 ms) while SAR data sets are much larger and are processed with a more relaxed real-time deadline (e.g., 8 GB in 16 s). These differences lead to interesting tradeoff implications in systems designed to support both GMTI and SAR. These tradeoffs will be discussed further in this chapter.

With the high latencies and limited throughput available from satellite downlinks, the only way to perform space-based GMTI and SAR in a real-time manner is to handle the computation onboard the satellite that collects the data. This real-time processing presents a significant challenge not only due to power, size, and weight requirements, but because space-qualified processors and networks are several generations behind the most modern technology available. Therefore, it is extremely important to intelligently design systems to make the most of available throughput and processing capabilities. While current satellite systems have not yet achieved the goal of real-time space-based radar, this work hopes to show that a RapidIO network capable of moderate clock speeds provides an effective network for real-time space-based radar applications using technology likely available for flight systems by the end of this decade. Our virtual-prototyping approach uses simulation to design proof-of-concept systems and evaluate

their effectiveness, saving development costs and providing hardware designers with valuable information to help them design the system correctly the first time. All models used in this phase of research have been validated against a RapidIO hardware testbed in order to ensure accurate results, and processing times, data set sizes, and other relevant algorithm parameters for our GMTI and SAR algorithms come from information provided by our sponsors at Honeywell, Inc. which is based on data from MIT Lincoln Lab.

Related Research

Work on GMTI and SAR has received attention in recent years [37, 38]. Space-time adaptive processing (STAP) in particular, one of the main components of any GMTI-based system, has also received much attention [39, 40]. One interesting set of related papers on GMTI comes from work performed at the Air Force Research Lab in Rome, NY. In late May and June of 1996, a series of flights were performed in which a ruggedized version of Intel's Paragon supercomputer system was flown aboard a surveillance airplane and real-time processing of STAP was performed [41, 42]. Partitioning of processing was performed via staggering of incoming data between processing nodes of the Paragon. One drawback of this partitioning method is that, while it provided the necessary throughput to perform real-time STAP processing, each processing result had a latency of approximately 10 times the incoming data rate. A group of researchers at Northwestern University attempted to improve upon these latencies by examining a parallel-pipelined version of the STAP algorithm used, and the results are presented in [43, 44]. Previous work performed at the University of Florida in [45] examines the performance of the RapidIO interconnect from a networking standpoint using a simplified version of the GMTI algorithm as a case study, while [46]

examines the performance of three parallel partitionings (straightforward, staggered, and pipelined) of the GMTI algorithm over a RapidIO network and [47] explores considerations for systems capable of both GMTI and SAR. It is also important to note that our pipelined version of the GMTI algorithm is based on the work in [44]. Researchers at MIT Lincoln Lab in 2004 flew a Boeing 707 containing a ruggedized testbed for GMTI and SAR, and presented experimental results in [48]. The algorithm was not performed in real-time, however. Of course, one of the key challenges for moving airborne processing technologies to space-based systems is getting the necessary processing power into a self-contained system while meeting the strict size, weight, power, and radiation requirements imposed by the space environment. The work performed for this phase of research seeks to meet the networking-related aspects to these requirements through the use of RapidIO.

Background

In this section, we provide an overview of the basic structure of the GMTI and SAR algorithms studied in this work and briefly present performance results from a previous study on the mapping of three GMTI partitionings to a RapidIO-based system.

GMTI

GMTI is an application that is important to military operations, as it allows moving targets to be tracked and laid over a map of a battlefield for strategic planning during a conflict. GMTI works best when combined with some form of airborne or space-borne radar system. While GMTI is traditionally implemented in aircraft, the viewable area is limited due to the relatively low altitude of airborne radar. A satellite implementation of GMTI tracking targets from space would be ideal given the high altitude and viewable area, however, processing requirements pose formidable challenges for an SBR

implementation [49]. One of the main drawbacks of GMTI is that it requires a high degree of processing power, anywhere from 40 to 280 GFLOPs and more, depending upon the size and resolution of the data being collected. Meeting these processing requirements is made even more difficult when considering the power- and speed-limited nature of the radiation-hardened, embedded architecture of satellite payload processing systems. However, onboard GMTI processing capabilities may be necessary in order to perform the processing in real-time and avoid the delays and throughput limits associated with satellite downlinks. Such delays could render GMTI applications ineffective if target detection times are too high.

Our GMTI algorithm is composed of several sequential steps or stages, which are Pulse Compression, Doppler Processing, Space-Time Adaptive Processing (STAP), and Constant False-Alarm Rate detection (CFAR). A 3-dimensional data cube represents incoming data, where each data element in the cube consists of two 4-byte integers, one real and one imaginary. Typical values for data cubes range in the size from 10 MB for data cubes associated with airborne radar systems and to 6 GB or more for space-based radar systems due to an increase in the satellite's line of sight [38, 48].

The three dimensions of the data cube are the range dimension, the pulse dimension, and the beam dimension. For each stage of GMTI, the complete data cube is distributed over the memories of all nodes involved in the processing of that particular stage. Furthermore, this distribution of data is determined on a stage-by-stage basis, according to the ideal processing dimension for each stage. In some cases, the data cube needs to be reorganized in-between stages in order to optimally distribute the data for the next stage of processing [41]. This reorganization of data is referred to as a "corner

turn," and can be a costly operation to the system, requiring personalized all-to-all communication of the entire data cube among nodes involved in the processing. If consecutive stages perform computations along the same dimension, no data transfer between the stages is required. Figure 4-1 shows the processing flow of our GMTI algorithm, including the ideal partitioning for each stage.

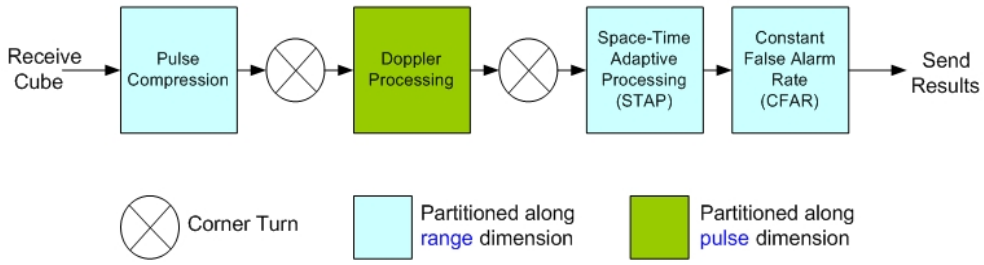


Figure 4-1. Processing flow diagram for GMTI.

Incoming data is grouped into Coherent Processing Intervals (CPIs), which are defined as the time interval that contains data that can be processed at a single time by the GMTI algorithm [38]. For the data cube sizes used in our studies, this interval is typically 256 milliseconds. Since data arrives from sensors as a stream, a new data set is ready every 256 milliseconds, making the CPI interval a real-time processing constraint for the system.

In [46] we studied three different partitioning methods for the GMTI algorithm. For each partitioning method, we examined the CPI completion latencies for a range of GMTI data cube sizes (256 pulses, 6 beams, and varied number of ranges). The systems under study contained anywhere from five to seven processor boards, and each of these processor boards contained four custom ASIC Digital Signal Processors (DSPs) connected via a RapidIO network. Results from these experiments are shown in Figure 4-2. The first approach used divided the system processors into sets of processing groups

and allowed each group to work on a data cube simultaneously. For efficient communication, we configured each four-processor board in the system to be a unique processing group. This "staggered" approach was very efficient for our system designs and required the smallest number of processors, but had very high CPI latencies on the order of the product of the total number of processor groups and the CPI. The next approach was a "straightforward" data parallel method in which each data cube was divided up among all the system processors. This brute force approach provided the lowest CPI latencies, but required the most processing boards due to communication inefficiencies associated with the large communication groups. The final method was a pipelined approach dividing the processors up into groups responsible for processing different stages of the GMTI algorithm. This method provided a compromise in terms of CPI latencies and efficiency. For the purposes of this study, we use the straightforward method as the baseline GMTI algorithm, since it provides the highest possible performance of the partitionings studied.

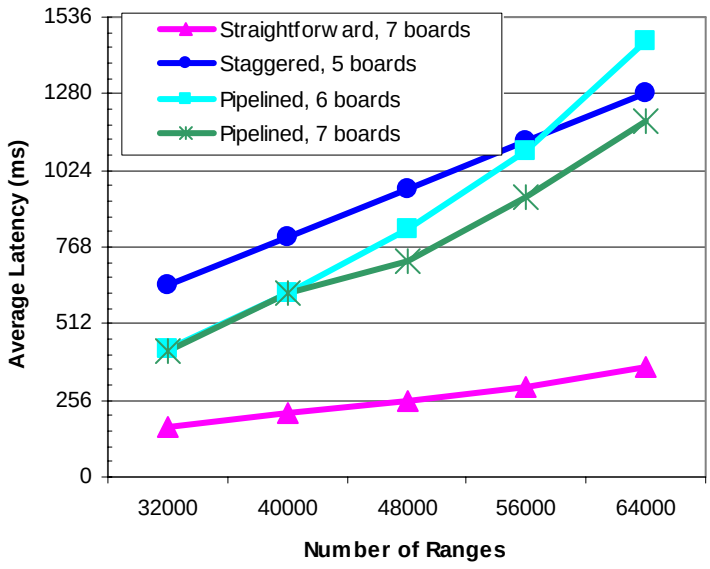


Figure 4-2. CPI latencies for three GMTI partitionings.

SAR

SAR is another radar application that is very important to military and intelligence operations. The purpose of SAR is to produce high-resolution images of the Earth's surface, and it can be used effectively even in the presence of inclement weather or the darkness of night. Similar to GMTI, SAR is typically implemented in aircraft for airborne reconnaissance. Also like GMTI, SAR would benefit from a satellite implementation taking images from space with a larger field of view [37]. However, the data size of each SAR image is much larger than that of the GMTI data cubes even for airborne implementations, and in turn there is more time allocated to the processing of each SAR image. Several variations of SAR exist, such as spotlight SAR and strip-map SAR. Spotlight SAR focuses the radar on a single patch of ground, taking several radar signals and forming a high-resolution image. Strip-map SAR takes signals over a wider area of ground [50]. The type of SAR selected for our simulative study is spotlight SAR.

Our SAR algorithm is composed of seven subtasks, including Range Pulse Compression, several FFTs, Polar Reformatting, Auto-focusing, and a Magnituding function. Range Pulse Compression is a technique used to increase the efficiency of the radar transmitter power usage, without degrading the resolution of the resulting images. Polar Reformatting converts the data set which has been collected on a polar grid to a data set in a Cartesian grid. The Auto-focusing stage goes through the whole image and corrects phase errors, such as those caused by platform (airplane, satellite) motion. Finally, Magnituding takes the complex data elements and converts them to real numbers. Figure 4-3 shows the processing flow and subtask ordering of the SAR algorithm studied in this research.

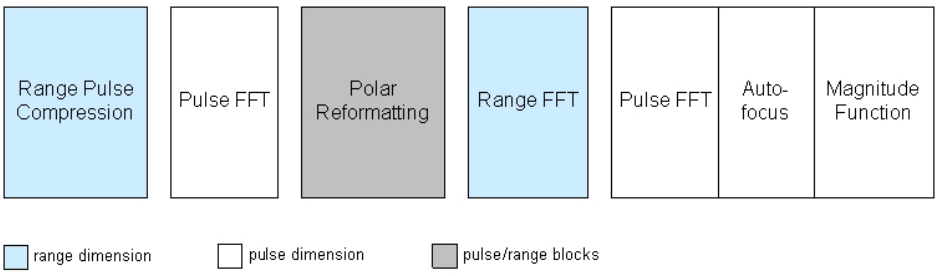


Figure 4-3. Processing flow diagram for SAR.

The input data for SAR is arranged as a 2-dimensional matrix, or image, with each data element consisting of two 32-bit integers representing a single complex number. Each subtask within SAR processes the image with a particular ideal partitioning of the data set for efficiency purposes. If consecutive stages do not use the same partitioning, reorganization of the distributed image data in between processing stages will be required. While most of the stages process the data along one of the two dimensions, either the range dimension or the pulse dimension, Polar Reformatting processes the data in blocks.

Even though there are several high-level similarities between SAR and GMTI, there exist enough differences to pose significant challenges to the design of a system intended to support both applications. The differences in terms of data size and processing deadline result in different system-level memory and throughput requirements. These requirements affect the system architecture design decisions regarding resource provisioning and interconnect topology, in turn affecting the way in which each application is able to be mapped onto the system. The differences between GMTI and SAR and their effects on mapping the algorithms to a single system architecture are explored later and are a key focus of this work.

Experiments and Results

This section describes the simulation parameters and experiment setup used for this research and then details the results of our GMTI and SAR simulations. The section concludes with a discussion of the important factors for developing RapidIO-based satellite systems capable of performing both real-time GMTI and SAR.

Simulation Parameters

Over the course of this research, we validated the accuracy of our RapidIO models by comparing results against real hardware as well as analytical results. Of course, one of the many benefits of simulation is that once models are validated, parameters can be adjusted to model hardware that is not yet available. Because all elements in a space system are typically radiation-hardened to be space-qualified, this puts the chip density and clock rates in such systems several generations behind the technology available for a terrestrial system. For these experiments, we set clock rate and other relevant parameters to values that we believe will be possible for radiation-hardened systems by the end of the decade when space-based radar systems are ready to fly. The parameters we use for our system are listed below and were derived with guidance from our sponsors at Honeywell Inc. as well as from information obtained from our comprehensive literature search and validation experiments. Unless otherwise noted, each experiment in this chapter uses the following parameter set:

- 16-bit parallel 250 MHz DDR RapidIO links
- 8-bit RapidIO device IDs (supports up to 256 RapidIO endpoints in the system)
- input and output queue length of 8 maximum-sized RapidIO packets on each endpoint
- 16 Kbytes of central-memory available in switch packet buffer

- 250 MHz ASIC processing elements (referred to as nodes) with 8 floating-point units per element
- The default system used for GMTI is a seven-board configuration with a four-port data source (as shown in Figure 3-3), running a straightforward parallel partitioning of the GMTI algorithm.
- The default system used for SAR is a four-board configuration with a four-port global-memory board, running a chunk-based parallel partitioning of the SAR algorithm (described later).

GMTI Experiments

Our first set of experiments was designed to study the GMTI algorithm. Because of the real-time nature of the algorithm, it is necessary to simulate several iterations (CPIs) of the algorithm in order to make sure that the system can keep up with the real-time requirement. Our baseline GMTI approach uses the RapidIO logical message-passing layer. A sensor board with four RapidIO ports is responsible for collecting data and sending to the processors. This sensor board model is similar to the global-memory board model previously described, but only needs a limited amount of memory for buffering detection results that are returned. As each CPI of data is collected, the sensors break up the data cube and split it across all processors in the system (the "straightforward" approach). The processors then begin to perform the GMTI algorithm on the data cube, using interprocessor communication (i.e., the corner turns) to redistribute data between stages as necessary. Once the processing is completed, the results are sent back to the sensor board, where they may be temporarily buffered in the small memory as they are transmitted down to the ground. The actual collection of data and transmission of detection results are outside the scope of our models, so the sensor board acts as a source and sink for traffic in the system.

The parameters used for the GMTI algorithm are as follows:

- 256 ms CPI
- 256 pulses
- 6 beams
- 8 CPIs simulated
- number of ranges varied from 32000 to 64000 in increments of 8000 for most experiments

Using 64-bit complex integers as the data type, and assuming 64000 ranges, these algorithm parameters lead to a data cube size of 750 Mbytes. Dividing this value by the 256 ms CPI leads us to the conclusion that approximately 3 Gbytes/s of bandwidth from source to sink is necessary in the network to ensure that the nodes are fed with data fast enough to meet the algorithm requirements.

Algorithm-synchronization experiment

Over the course of performing the GMTI experiments, we quickly learned that the massive data sizes we were dealing with had the potential to greatly degrade system performance by introducing large amounts of network contention if not managed properly. In particular, the corner turns were a frequent source of network contention. In the case of the straightforward parallelization method, the corner turns create an all-to-all communication between the processors in the system. If not synchronized properly, this pattern can cause scenarios where several nodes are trying to send very large messages to a single receiver at the same time. Of course, the RapidIO network is capable of handling this contention through physical layer flow control, but flow control should not be used as a substitute for efficiently-planned communication patterns. The physical layer flow-control mechanism of RapidIO—and in fact most networks—is much better equipped to

handle the congestion associated with a few sender nodes trying to contend for a single destination with some messages of a couple hundred bytes, rather than the much larger messages associated with GMTI corner turns.

We accomplish synchronization by inserting a barrier sync before each step of each corner turn. Nodes will be scheduled to send data such that there are never two nodes trying to send to a destination at the same time. For example, if there are four nodes in the system numbered 0, 1, 2, and 3, node 0's corner turn script will look similar to the following:

- Barrier
- Send to 1
- Barrier
- Send to 2
- Barrier
- Send to 3
- Receive from 1
- Receive from 2
- Receive from 3

Data may be received from any source at any time, and the processor model does not have to wait until it encounters the receive instruction to begin receiving data. The receive instruction simply prohibits the processor from advancing until it has received the data.

Figure 4-4 displays the average CPI completion latencies for the range of data cube sizes tested, with and without the synchronization methods described above.

Performance was clearly better for all cases except the smallest cube size, which did not experience enough network contention to greatly benefit from synchronization. As cube sizes increase, the opportunity for contention increases, increasing the need for synchronization. However, contention can also lead to extremely unpredictable results, so the size of the gap between the synchronized and unsynchronized versions tends to vary somewhat erratically as cube size increases.

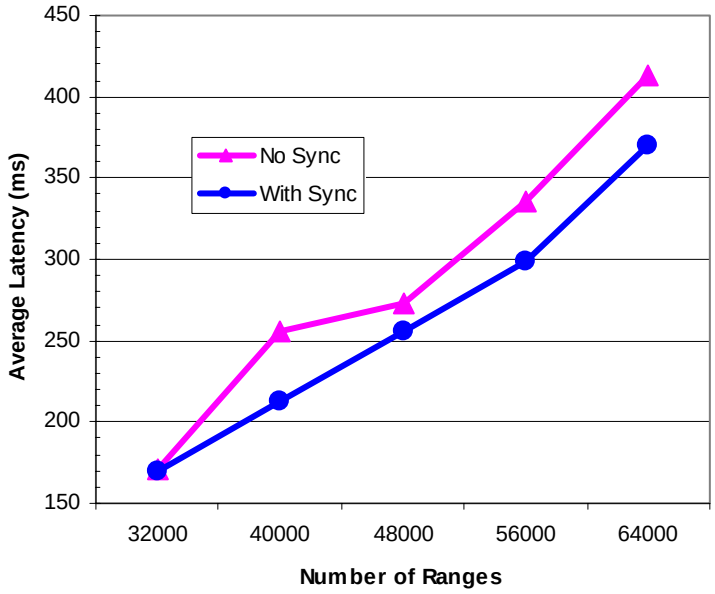


Figure 4-4. GMTI with synchronization versus no synchronization.

While CPI completion latency is the most important metric of our results, we studied several other aspects of these results in order to better understand the trends we observed. Figure 4-5 shows a histogram of the switch memory usage of one of the processor-board switches. This histogram is taken from the unsynchronized 40k ranges set of results. The switch memory histograms used in this section indicate the percentage of simulated time a switch spends with memory lying in certain ranges. The memory ranges are displayed as a percentage of the total available switch packet buffer memory

(16 KB). For these experiments we divide the switch memory into 20 segments, so each range represents five percent of the total switch memory. We arbitrarily chose the switch on processor board 5 as the switch observed, but due to the symmetric nature of the algorithm the switches of all boards exhibit nearly identical memory usage characteristics. High values on the left of the chart correspond to less free memory being present in the switches over the course of the simulation. The graph shown indicates about 60% of the simulation run is spent with very low switch memory. It is important to note that switch memory never dips much below 3000 bytes due to the configuration of the switch buffer priority thresholds. Allowing data packets to take more switch memory would not solve the problem and equal amounts of contention would occur. The only effect would be to slightly delay the time that it takes the switch buffers fill up.

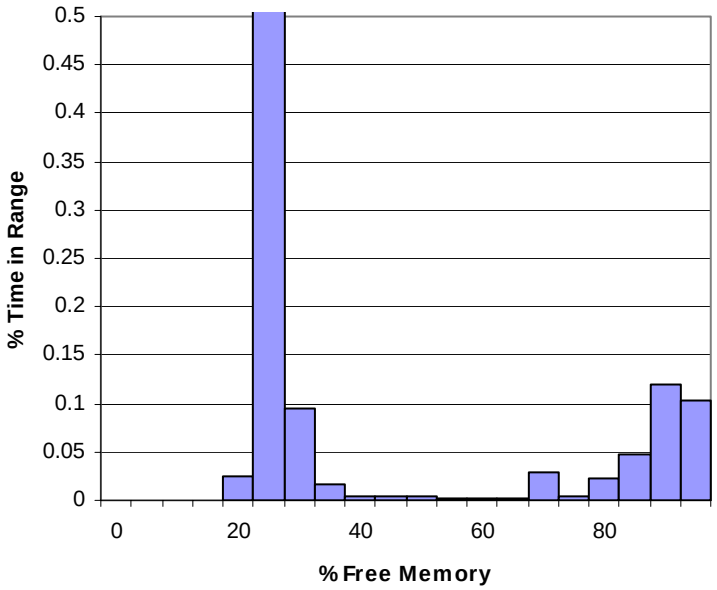


Figure 4-5. GMTI switch memory usage without synchronization.

Figure 4-6 shows the memory histogram for the 40k synchronized experiment. Because communication is much more efficient, switch memory is less utilized, and the

switch spends about 90% of its time with greater than 13 KB of the 16 KB total memory free.

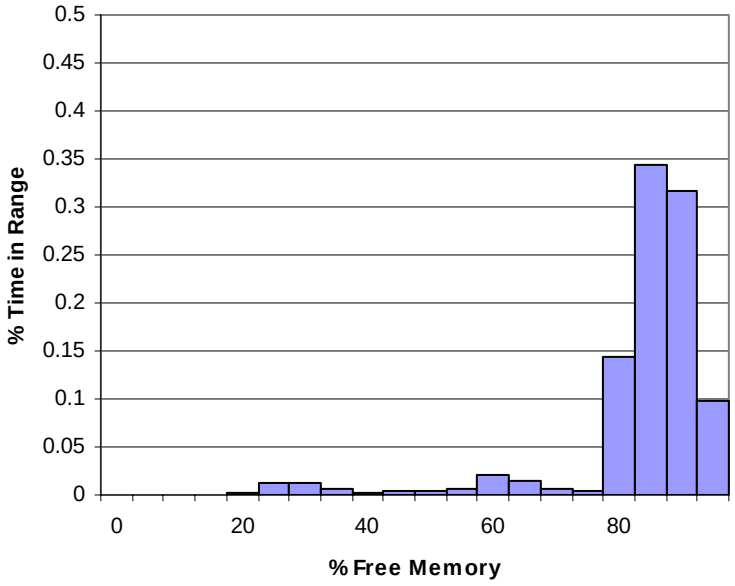


Figure 4-6. GMTI switch memory usage with synchronization.

Even when performed before every send operation, the latency associated with the barriers only accounts for about 0.003% of the total corner turn time. Because the synchronized version of the algorithm provided significant performance increases for larger cube sizes and no decrease in performance (actually a very small gain) for the smallest cube size, we select the synchronized version of the GMTI algorithm as our baseline for the remainder of this work.

It should be noted that the end-to-end flow control [21] of the new RapidIO RapidFabric protocol extensions [21-23] could help reduce network contention in the case of an unsynchronized system and allow nodes to use the network more efficiently. However, we still argue based on these results that the most efficient use of the network

is to intelligently plan the communication patterns the first time, and let the flow control handle isolated incidents of congestion.

While this experiment was GMTI-specific, the results and lessons learned can certainly be applied to other algorithms with similar communication parameters. Intelligently synchronized communication is important for many cases, and will help reduce stress on the RapidIO network. While the network has the capability to handle contention on its own, it is not necessarily wise to rely on it to do so if the level of contention is going to be high for an extended period of time.

Scalability experiment

In order to evaluate the scalability our system when running GMTI and discover the upper limits of the architecture's processing capabilities, we seek to determine the maximum data cube size supported by 5-board, 6-board, and 7-board configurations of our baseline system (which supports a maximum of seven boards and one four-port global-memory board). For each system, we perform tests across a range of cube sizes and evaluate the performance of each system. The range of cube sizes run for this experiment includes two extra points at 72k ranges and 80k ranges in order to see just how far each system will scale in terms of problem size.

The GMTI experiments in this study assume some amount of double buffering is possible. In other words, a node is allowed to receive parts of data cube N+1 while performing processing on data cube N. The use of double buffering increases node and board memory requirements by about a factor of two, depending on the system architecture and computational approach. Using a straightforward partitioning with no double buffering, any GMTI experiment in these sections yielding a CPI completion time of greater than 256 ms will not meet the real-time deadline. For example, the

synchronized baseline runs from the previous section would only be possible up to 48k ranges instead of the 64k case that is possible with double buffering. However, even using a double-buffered approach, the system must still be able to get the initial data from the source to sink in less than 256 ms, or the system will fall behind the real-time deadlines of the algorithm by not getting data to the nodes in time.

Assuming that full double buffering is possible, there are two key factors that dictate whether or not the system can meet the real-time deadline:

1. Can data get from source to sink within the 256 ms timeframe? If not, the traffic source will become congested with no chance for relief unless a CPI is skipped, which could be disastrous for some GMTI applications.
2. Once the data is received, can the processing elements process it and return the results within 256 ms? If not, the node will be forced to store more than two data cubes at once and will quickly fall behind the real-time deadline and run out of memory.

Figure 4-7 displays the CPI latencies measured for each system over the tested range of cube sizes. The dashed line in the figure also displays the 512 ms overall real-time deadline (the sum of the two deadlines described above). The results are fairly linear, with some deviations as the cube size gets larger and network contention increases. Each system has a sufficient amount of source-to-sink bandwidth to deliver data within 256 ms, so systems that fail to meet the deadline do so because the computation phase (including corner turns) takes more than 256 ms. The seven-board (i.e., 28-node) system was able to handle all cube sizes, just passing under the deadline for the largest cube size. The six-board system was able to handle all cases except the largest case, and the five-board system was able to process all of the cube sizes previously examined, but not the two larger sizes used for this experiment. These results mean that, given a specific algorithm parameter of 64k ranges, a five-board system could

potentially be used in place of the seven-board system, leaving two boards attached to the backplane as spares in the event one of the boards experiences a critical fault. We will further explore the issues related to fault-tolerant RapidIO-based space systems in Chapter 5.

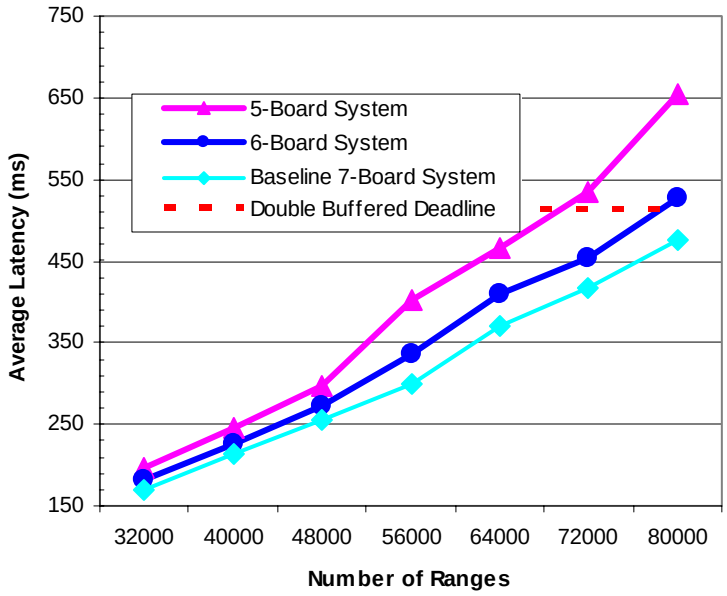


Figure 4-7. GMTI scalability.

The results in this section show that our system is scalable in terms of number of boards as well as algorithm requirements. Double buffering is a key factor in providing this scalability, as it enables much larger cube sizes to be processed than previously possible, as long as the data can be delivered to the processor nodes in time.

Clock-rate experiment

Our primary assumption regarding network clock rate is that it will be possible to build a space-qualified RapidIO network capable of 250 MHz DDR clock rates when space-based radar is ready to fly. This rate yields approximately 8 Gbps of throughput for a single RapidIO link in a system with 16-bit parallel RapidIO. However, in the event that this rate is not possible, we would like to evaluate systems designed for other

RapidIO link rates. Preliminary experiments indicated to us that backplane design was very important, especially in terms of data source-to-sink throughput. In fact, this source-to-sink throughput is the primary factor that dictates system design for one simple reason—if there is not enough throughput to get a single data cube to the nodes within the span of a single CPI, it is impossible for the system to keep up with the real-time requirement. In other words, the system can only process data as fast as it is provided with data. Therefore, a change in clock rate necessitates a change in system design or a change in algorithm parameters. In this case, we chose a 125 MHz DDR RapidIO system as the most likely feasible alternative to a 250 MHz system. In order to provide enough data to the nodes under the reduced data rate, we must double the number of data source ports supplying the data. All other aspects of the system are the same as the system in Figure 3-3, with the exception that a processor board must be removed in order to have an eight-port data source connected to the backplane. Figure 4-8 shows the average CPI completion latency of each configuration for a range of data cube sizes.

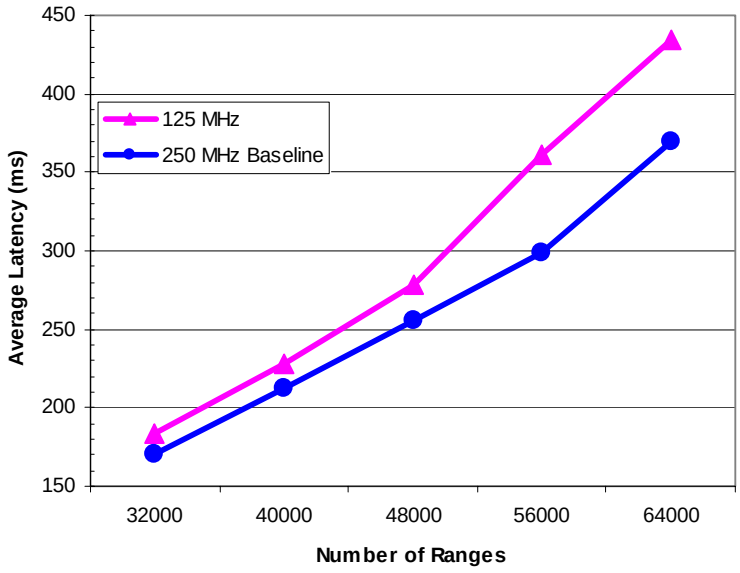


Figure 4-8. GMTI with 125 MHz and 250 MHz DDR RapidIO.

For smaller cube sizes, the 125 MHz system is able to compete quite favorably with the faster system. As cube sizes increase, the gap between the performance of the two systems widens. This gap can be partly attributed to the reduced processing power of the 125 MHz system, since it has one less processing board (four less processors) than the 250 MHz system. The backplane slot for the removed processing board is needed for the additional four data source ports. The results in this section show that the system architecture under study is flexible in terms of RapidIO clock rate and is capable of supporting the GMTI algorithm at rates slower than our baseline. The most important consideration in changing the clock rate is ensuring that adequate source-to-sink throughput is still provided so that data can be delivered to the processing nodes within the deadline. It should be noted that reducing the RapidIO link width by a factor of two is equivalent to reducing the RapidIO clock rate by the same factor, so these results also apply to the case of using 8-bit RapidIO rather than 16-bit RapidIO.

Flow-control experiment

As discussed in Chapter 2, RapidIO supports two different mechanisms for physical-layer flow control. Our baseline method of flow control is receiver-controlled flow control, in which the receiver of a packet is responsible for sending a negative acknowledgement if there is not enough space for a packet of its priority. The alternate method is transmitter-controlled flow control, in which a node's link partner is kept up to date on its buffer space through buffer status fields in idle packets and control symbols. We performed a set of experiments using both flow-control methods, and results showed that both methods performed nearly equally. In general, the transmitter-controlled method was slightly faster, but usually by less than a very small fraction of one percent of total execution time. A more significant potential benefit of transmitter-controlled flow

control is that there should never be any negative acknowledgements transmitted, because the transmitter always knows if the receiver has enough space for its packet. This flow-control method could potentially reduce power consumption if the RapidIO endpoints consume less power when transmitting idle packets (which contain buffer-space information) than data packets, since data packets may retried and retransmitted multiple times under receiver-controlled flow control. The power consumption parameters of a RapidIO endpoint are implementation-specific and outside the scope of this work.

RapidIO switch parameters experiment

A battery of GMTI experiments were run with total switch memory reduced by half (to 8 KB) in order to determine the system sensitivity to changing the switch memory size. Changing the switch memory size varies the amount of buffer space available in the RapidIO switches. For the most part, results were not greatly affected by the amount of memory available to the switches. Adding memory to a switch cannot get packets from one end of the network to the other any faster; it will only allow more packets to buffer up inside the network. Since the traffic patterns in GMTI are very regular, large packet buffers in the RapidIO switches are not necessary. Buffers are only needed during congested periods, which should be avoided at all costs in order to maintain the real-time algorithm performance. However, other applications with irregular or bursty traffic patterns could benefit from large switch memory buffers.

In [45], we performed studies to determine the effects of having switches that support cut-through versus store-and-forward routing in the system. The data indicated that there was virtually no difference in performance, as GMTI is heavily throughput-bound and not sensitive to the latency of individual packets.

SAR Experiments

Our second set of experiments focuses on the examination of system designs for the SAR algorithm and the study of their performance. Because of the very large memory requirements associated with SAR, a straightforward partitioning approach like the one used for GMTI is not possible within our design constraints. Our baseline SAR memory requirements are approximately 2.5 times those of the GMTI requirements, and necessitate the use of a large global memory bank. Future missions are anticipated to require processing of even larger SAR images (32 GB or more) and will also require a global-memory-based architecture unless the growth rate of radiation-hardened memory densities unexpectedly increases. While the global-memory-based approach clearly presents a potential bottleneck, it is currently the only option for space-based SAR processing of large images.

For our algorithm, the SAR image must be broken up into chunks that are processed out of the global memory. For each stage of the algorithm, nodes read a chunk from memory, process the chunk, and write the chunk back to global memory. The reads and writes are performed remotely over the RapidIO interconnect. This process continues for each of the remaining chunks until the processing for that stage is finished. One benefit of this approach is “free” corner turns, since the nodes can read out the data in any order without additional communication penalties incurred for the rearrangement of data.

The baseline parameters used for the SAR algorithm are as follows:

- 16 s CPI
- 16384 ranges
- 16384 pulses
- System-level chunk size varied from 256 KB to 16 MB for most experiments

System-level chunk size is the product of the per-processor chunk size and the number of processors in the system. It should be noted that total image size is the product of the number of ranges, number of pulses, and number of bytes per element (8 bytes per element is assumed in our experiments). Our baseline image size is approximately 2 GB, which is a typical expected value for a first-generation space-based SAR system.

Because GMTI has more stringent network requirements than SAR, we start with the GMTI baseline system when determining our preliminary system needs for SAR. Compared to GMTI, SAR has a very long CPI and therefore has less strict processing requirements for meeting the real-time deadline. A four-board (16-node) system meets the SAR processing requirements and was selected as our SAR baseline. Additional system sizes and dataset sizes were also examined; however, the scalability of this particular system configuration was limited by the throughput of the RapidIO interface to the global memory. Future systems are expected to require even larger SAR images, but will also have the advantage of increased network and memory bandwidth.

The backplane used for SAR is identical to the backplane used for our baseline GMTI system. This system design potentially leaves three boards as spares that may be turned off, left idle, or used for other purposes. Of course, one or more of the three spares may be completely omitted from the system to minimize weight if GMTI processing is not also desired.

Because of the global-memory approach taken for SAR, we chose the RapidIO logical I/O layer as our baseline logical layer. The logical I/O layer allows nodes to issue “read” and “write” packets to global memory, rather than using a message-passing

paradigm to communicate. Later in this section we examine the effects of varying the logical layer.

Algorithm-synchronization experiment

As we observed with GMTI, careful use of the network is essential in obtaining optimal performance. For GMTI, large amounts of all-to-all communication with large messages were the limiting factor in network performance. For global-memory-based SAR, all-to-all communication is no longer an issue since corner turns are performed in place by reading and writing the global memory bank. However, the main networking issue now becomes contention for the four global-memory ports between the 16 processors in the system. Our initial experiments showed that as chunk sizes increased, network contention caused CPI latencies to increase unexpectedly. To understand the cause of this contention, it is important to understand the steps a processor follows to complete the processing of a chunk. The normal sequence of operations for each processor to process a chunk is

1. read chunk from global memory;
2. process chunk;
3. write chunk to global memory.

At the very beginning of the simulation and many times thereafter, 16 processors all try to read data from only four global-memory ports. Immediately, these reads create a significant amount of network contention that gets worse as chunk size increases. Over time, this is a very inefficient behavior to maintain; therefore, it is important to control the access to global memory. Several methods were attempted to synchronize access to global memory, with varying impacts on performance. We label the first method “sync level 1.” This method focuses on synchronization of memory reads. In order to read

from global memory, a node must have the “read token.” This token is a simple small packet, not to be confused with a token-ring network. Once a node is finished reading, the node passes its token to a neighbor. With a four-board system, there is always one token per board, being held by one of the four nodes on the board. The next method, called “sync level 2,” uses a “write token” in addition to the read token. The tokens are independent, and one node may hold the read token while another holds the write token. The last method is “sync level 3,” and it uses a single token for reads and writes. A node can read or write global memory as long as it possesses the token. The token is passed around such that each node gets a chance to read (and then compute), and then each node gets a chance to write back its results. Of the three synchronization levels, level 1 has the most concurrency available, followed by level 2, followed by level 3 having the least concurrency and the most serialization.

The results of our SAR algorithm-synchronization experiments are shown in Figure 4-9. The performance of the unsynchronized version of the algorithm does not scale well with increasing chunk size. Adding the first synchronization level (synchronized reads) greatly improves performance, but adding level 2 decreases performance slightly and adding level 3 decreases it tremendously. Adding the final synchronization level did prove an interesting point, however. By totally synchronizing both reads and writes against each other, virtually all potential for network contention is removed. However, almost all of the bi-directional capabilities of the RapidIO network are negated, and the algorithm is excessively serialized. The serialization reverses the previous trend of decreasing performance with increasing chunk size, but also makes

performance so slow that it is not practical to use the excessively synchronized version of the algorithm.

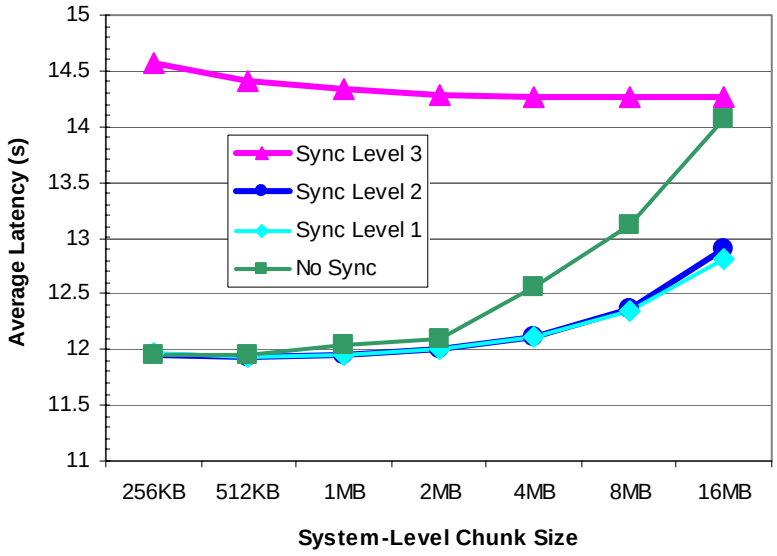


Figure 4-9. SAR performance under varying levels of synchronization.

It is important to note that our models used in this phase of work do not account for potential processing inefficiencies that may occur due to processing small chunks of data—only networking inefficiencies are reflected in our results. However, an in-depth understanding of the networking issues allows a designer with knowledge of the actual processing hardware to make an informed decision on the appropriate chunk size for the system. Our work in Chapter 6 will use simulation to examine the processing activities of GMTI and SAR in more detail and attempt to balance optimal processing chunk size with optimal network chunk size.

As sync level 1 had the best performance and was the simplest of the three synchronization levels tested, it will be the baseline SAR configuration for the remainder of this chapter.

Logical I/O versus message passing experiment

While the RapidIO logical I/O layer was chosen as our SAR baseline due to its appropriateness for a distributed-memory system, it is not the only option. The chunk-based SAR algorithm can also be implemented using the RapidIO logical message-passing layer, which is the same layer used for our GMTI experiments. Use of this method assumes that the memory controllers on the global-memory board have knowledge about the algorithms being performed by the processor nodes. The memory controllers must be able to get the data to the nodes partitioned along the correct dimension for each algorithm stage using "send" packets (a "push" model) rather than having the nodes issue "read" packets to acquire the data (a "pull" model). Likewise, when nodes write results to memory, they issue "send" packets to the global memory that the controller must translate to a memory write. Overhead associated with this process is not considered in our models. A comparison of the logical message-passing versus the logical I/O implementations of SAR is shown in Figure 4-10.

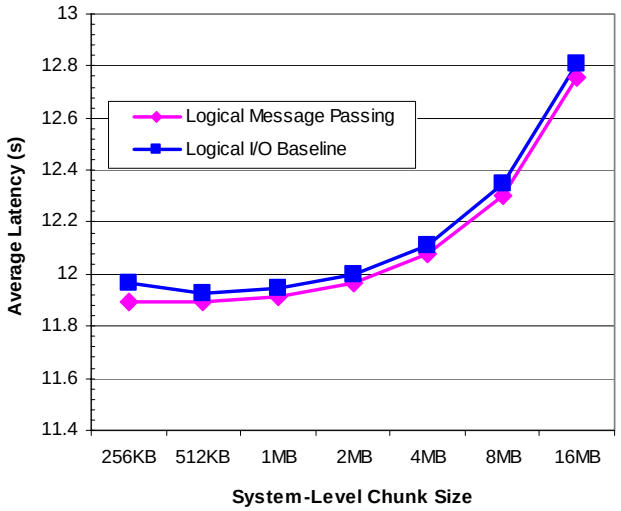


Figure 4-10. SAR performance under logical I/O and message passing.

The message-passing layer (i.e., “push” model) slightly outperformed the logical I/O layer (i.e., “pull” model) in all cases. Most likely, the performance difference is due to slightly reduced levels of contention and the implicit rather than explicit synchronization that occurs with the message-passing version. Instead of having to wait for a token in order to send read requests to the global memory, nodes are able to simply wait for the data to be sent to them from the global-memory board. However, it is still our belief that the logical I/O layer is a more appropriate fit for the global-memory-based SAR processing in order to reduce memory-board complexity.

It is important to note that the logical layer is the uppermost layer of the RapidIO protocol stack, and this is the layer with which applications must interface. In some RapidIO implementations, the overhead associated with message passing rather than directly accessing memory via logical I/O may impose a latency penalty on individual packets. However, GMTI and SAR are throughput-bound and not sensitive to latency of individual packets. RapidIO throughput is primarily dictated by physical-layer characteristics, making the choice of logical layer for SAR (and GMTI) primarily dependent upon implementation complexity or other requirements such as power or gate-count.

Double-buffering experiment

For our GMTI experiments, we used the phrase “double buffering” to describe the ability to process data from one data cube while receiving data from the next data cube. In this section we examine the effects of double buffering on SAR at the chunk level rather than buffering a complete CPI’s worth of data. We created several scripts to simulate a double-buffered system, where a node would be allowed to process a chunk while receiving the next chunk. In order to accomplish the double buffering,

synchronization was removed and nodes were permitted to ask for data from global memory whenever they were ready for it.

Much like GMTI, double buffering for chunk-based SAR requires some amount of extra on-node or on-board memory. However, the added memory is less of a concern for SAR systems, since our SAR parallel algorithm partitioning requires only a small amount of node memory. Memory requirements per node are on the order of the per-node chunk size, which is very small compared to the GMTI requirements for on-node memory or compared to the size of the SAR global-memory bank.

The results of our double-buffering SAR experiments are shown in Figure 4-11. For the smallest chunk size, double buffering was found to improve performance by nearly 1 full second. As chunk size increases, however, potential for contention in the system increases even more heavily than in the standard synchronized logical I/O implementation. For medium and small chunk sizes, the additional contention is offset by the ability to double buffer and overlap communication with computation. However, double buffering with large chunk sizes can lead to even heavier contention and worse performance than the baseline implementation.

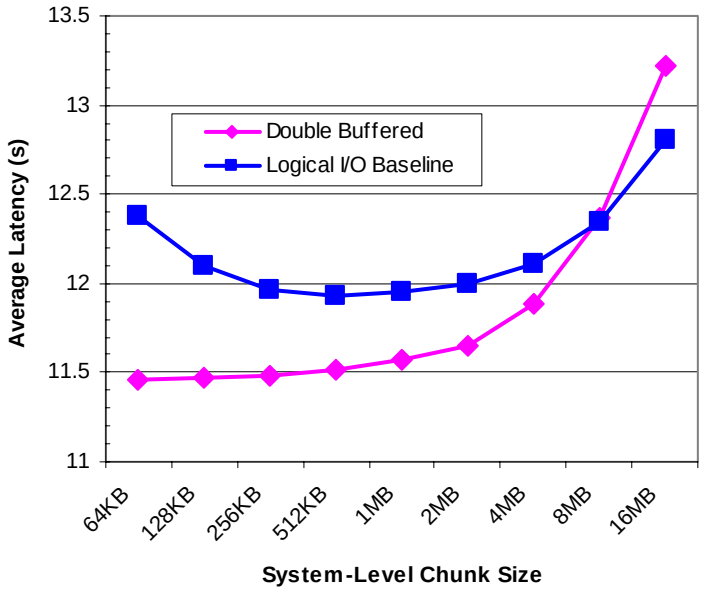


Figure 4-11. SAR performance with double buffering.

By incorporating lessons learned from the synchronization and double-buffering experiments, we created an algorithm script that incorporated the “read token” concept as well as double buffering. However, this method performed worse than the standard double-buffering method. We can conclude from this result that double buffering is an effective method to increase performance of the SAR algorithm, but it is only effective for small chunk sizes and inherently introduces network contention into the system that cannot be helped.

Clock-rate experiment

For this experiment, we examined the effects of reducing the RapidIO DDR clock frequency to 125 MHz. Because SAR has less stringent network requirements than GMTI, we did not anticipate that a 125 MHz SAR system would require additional global-memory ports in order to meet source-to-sink throughput requirements.

Therefore, we study performance of versions of the 125 MHz system and 250 MHz system with four-port and eight-port global memories, with results shown in Figure 4-12.

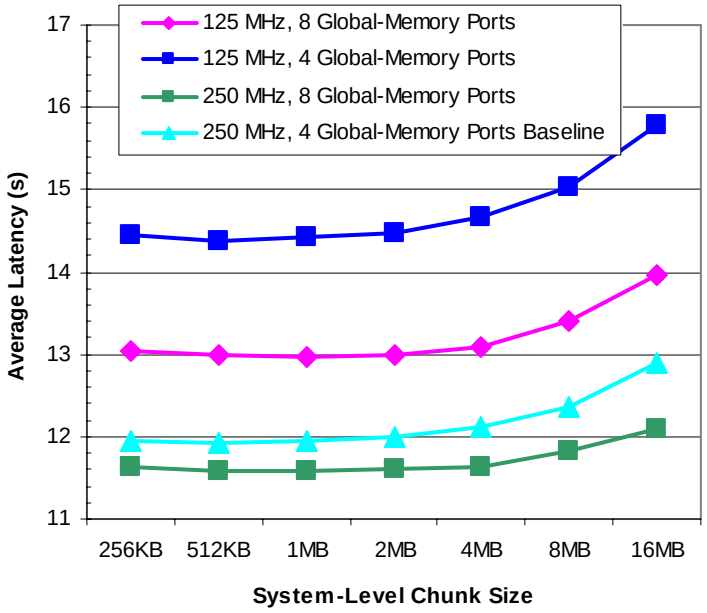


Figure 4-12. SAR with 125 MHz and 250 MHz DDR RapidIO.

All configurations examined were able to meet the 16 s real-time deadline, including the 125 MHz system with only four global-memory ports. The addition of four extra global-memory ports improved the performance of both systems, but it had the most dramatic effect on the slower system since the processors were likely starved for data due to the slower links. In both cases, the extra ports reduce contention and increase aggregate system throughput between the processing nodes and global memory.

Flow-control experiment

The SAR flow-control experiments followed the same trend as the GMTI experiments, with the transmitter-controlled flow control being slightly faster than the receiver-controlled flow control. For the most part, the differences are negligible.

RapidIO switch parameters experiment

Similarly to our GMTI results, changing the total amount of switch memory or using cut-through routing instead of store-and-forward routing for SAR did not have a significant effect on performance.

Logical-layer responses experiment

By default, the logical I/O RapidIO write is a responseless operation. However, the RapidIO logical I/O layer allows the option for writes to require a response be sent from the receiver back to the originator of the write request. Our RapidIO models support this option, and we ran a set of SAR simulations using "write with response" packets rather than conventional write packets. However, results were virtually identical for both cases. SAR is throughput-bound and we allow endpoints to issue as many requests as allowed by the flow-control protocol without waiting for a response. Since SAR is not sensitive to the latency of individual packets (or their responses), the only effect of the responses is to take up a very small amount of extra system throughput in the opposite direction of the main traffic flow (most write packets in our configuration are 272 bytes and their responses are only 8 bytes).

Dual-Function System: GMTI and SAR

While GMTI and SAR have many key differences in terms of processor, memory, and network requirements, it is desired in many cases to perform both algorithms on a single system. Such a system would perform a number of iterations (CPIs) of one algorithm, then switch and perform another number of iterations of the next algorithm, and continue to alternate in this manner. Table 4-1 gives a high-level outline of the key similarities and differences between GMTI and SAR, and Figure 4-13 shows the percentage of total execution time spent in one GMTI and one SAR experiment for

corner turns, data processing, and data and results delivery. Baseline synchronized configurations are used for both GMTI and SAR, with no double buffering used in order to most accurately differentiate computation time from communication time. Note that SAR's corner turns are performed "for free" as data is read and written from global memory as part of data and results delivery, and time spent transmitting synchronization messages is negligible and not shown for either algorithm.

Table 4-1. GMTI requirements versus SAR requirements.

	GMTI	SAR
Throughput Requirement	Multiple Gbytes/s source-to-sink throughput	More relaxed demands due to larger CPI
Packet Latency Requirement	Not sensitive to individual packet latencies	Not sensitive to individual packet latencies
Node Memory Requirement	High memory capacity (tens of MB)	Low memory capacity (tens of KB)
Global Memory Requirement	No global memory necessary	High memory capacity (multiple GB)
Processing Requirement	Very high demands due to small CPI	More relaxed demands due to larger CPI

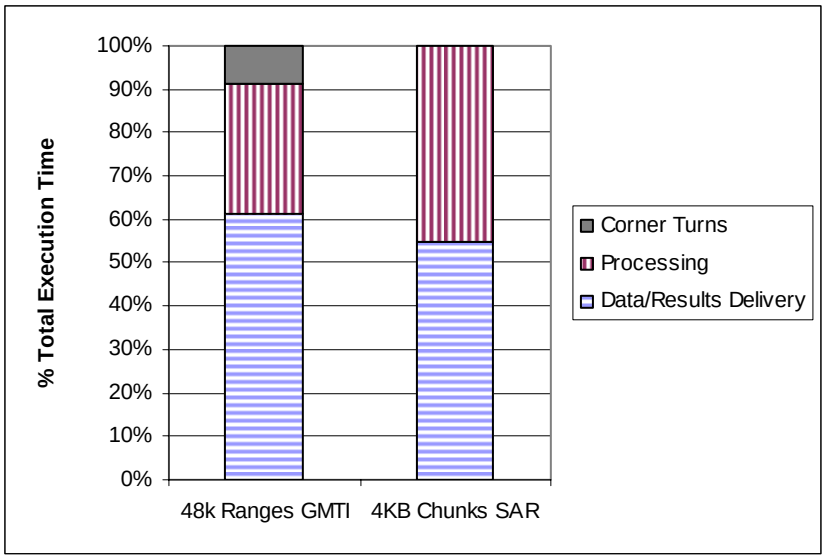


Figure 4-13. Performance breakdown of GMTI and SAR.

In order to design a system that is effective for both GMTI and SAR, we must build the system to meet the most demanding constraints of each category shown in Table 4-1.

We have already designed the network in this manner, with SAR running over a network with throughput capable of supporting GMTI, while also showing that SAR can be performed on a network with smaller throughput capabilities as long as sufficient bandwidth is provided to global memory. Figure 4-13 supports our earlier results by showing that network communication occupies more than 50% of both GMTI and SAR's execution times, and optimization of this communication has been shown to have a significant effect on execution time for both algorithms. Global memory is required in our space-based SAR system, and therefore required for a dual-function system, because it is not possible to put sufficient memory on a node to accommodate its complete fraction of a SAR image in an embedded system with our design constraints. Even though we have assumed data is streamed directly to the nodes for GMTI, it is equally feasible for data to be written to a global memory first, and then read out by the processing nodes and processed in the straightforward manner described in this work (without writing intermediate results back to memory as done with SAR). The processing power of each node must be sufficient for meeting the GMTI real-time deadline, which will in turn be more than sufficient for SAR. The amount of on-node memory required is constrained by the GMTI implementation selected.

If it is desirable to make the GMTI and SAR systems and requirements as similar as possible, there is one additional option for GMTI partitioning. GMTI may be performed in a global-memory-based manner by breaking the data cube up into chunks, just as we have done with SAR. The immediate advantage of this approach is that it would require much less memory on each node, since each node would no longer have to be able to hold its complete fraction of the GMTI data cube at one time. The system

designers could place just enough memory on each board to accommodate the optimal chunk size (compromising between GMTI and SAR and adding cushion if necessary for double buffering or other uses) and configure the GMTI and SAR algorithms to use this chunk size. To gain further insight into systems for both GMTI and SAR, we created a global-memory, chunk-based version of the GMTI algorithm. Communication patterns for the chunk-based GMTI algorithm are essentially identical to the SAR algorithm already studied. We performed an experiment similar to our SAR experiments by varying the GMTI chunk size for data cubes with 32k ranges and 64k ranges. Results are shown in Figure 4-14, with a comparison to the straightforward GMTI parallel partitioning of the data cubes with 32k and 64k ranges.

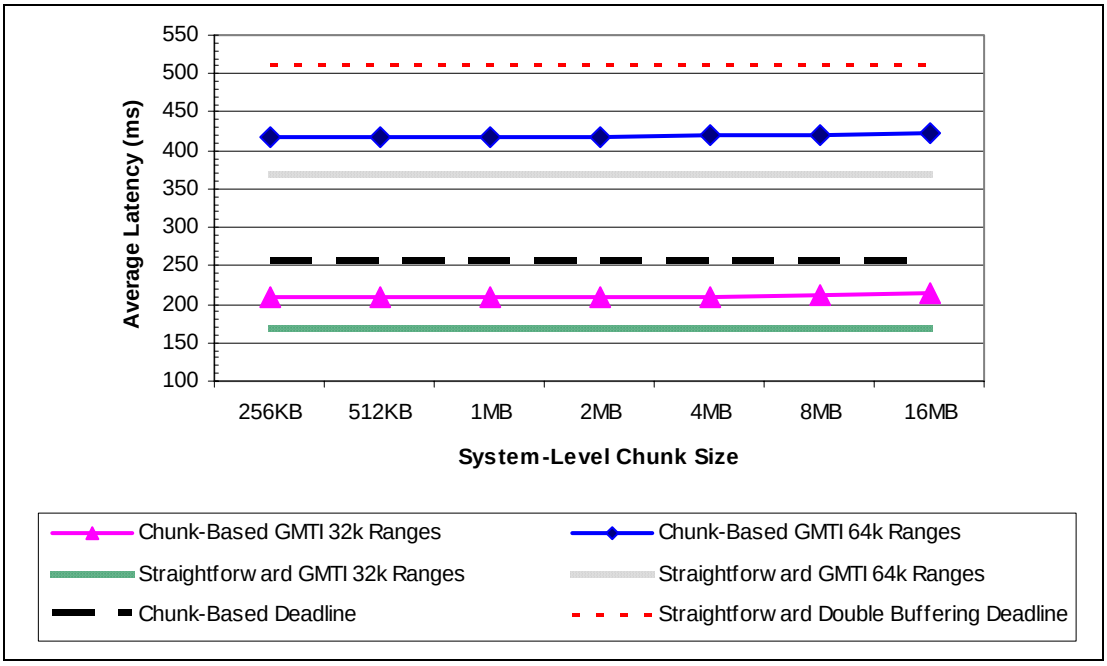


Figure 4-14. Chunk-based GMTI performance versus baseline performance.

The deadline for the chunk-based algorithms is 256 ms, because it is impossible to double buffer at the CPI level when a CPI is broken down into chunks. While the system running the 32k-ranges algorithm was able to perform within the real-time constraints,

the system with 64k ranges failed to meet the deadline for several reasons. Most importantly, the original system for 64k ranges required over 350 ms, which was adequate when allowing double buffering of an entire CPI but not acceptable for the chunk-based algorithm. Also, the chunk-based algorithm requires large amounts of redundant data transfer when compared to the original version. In the straightforward version, data is sent directly to the appropriate node during data redistribution in between stages. In the chunk-based version, data must be re-written to global memory and then read out of memory in the appropriate order by the correct node for the next phase of the algorithm. This process happens repeatedly for each data chunk, greatly reducing the overall performance of the application.

In general, the trends observed in GMTI and SAR chunk-based processing are very similar. One difference appears to be that GMTI suffers much less as chunk sizes are increased, leading to much more consistent performance than chunk-based SAR. A contributing factor is GMTI's shorter CPI, which does not amplify performance differences due to contention as much as SAR's longer CPI. The primary factor, however, appears to be that the size of each chunk of GMTI data is reduced after each stage, so data being written back to memory may be much smaller than the data that was read out. This phenomenon generally leads to less full-size chunks in flight in the system at any one time, reducing the effects of contention as chunk size grows.

Based on the GMTI and SAR experiments performed in this section, results suggest that the optimal system-level chunk size is either 512 KB or 1 MB. These chunk sizes fall in the "sweet spot" for most of our experiments for GMTI and SAR—large enough to make the network and processor usage as efficient as possible, but not so large that they

create excessive contention or require too much on-node memory. As mentioned previously, our modeling approach for this phase of research is network-centric, and processing factors may also significantly play into the final choice of an optimum chunk size.

Because a chunk-based approach to GMTI suffers greatly in performance compared to the original straightforward approach, the chunk-based implementation is only recommended for systems where it is desirable to reduce the amount of memory needed on each processor board. Otherwise, the system should be constructed with the "greatest common denominators" needed for straightforward GMTI (or staggered GMTI or pipelined GMTI as explored in [46]) and chunk-based SAR.

We have seen that the trends examined for the chunk-based SAR processing apply strongly to chunk-based GMTI. These results should also be easily generalizable to other processing applications that have similar communication patterns.

Summary

In this phase of research, we have used validated simulation to evaluate and optimize GMTI and SAR space-based radar algorithms for a satellite system with a RapidIO network. After performing a comprehensive battery of both GMTI and SAR experiments, we discovered many interesting results, the most important of which are outlined here.

We found that the performance of the GMTI algorithm was heavily throughput-constrained, and it was very important to intelligently plan communications patterns to reduce contention as much as possible. Source-to-sink throughput was a major factor in insuring that real-time deadlines were met, so the system had to be carefully designed to allow high-throughput data flow to the end nodes. A 250 MHz DDR RapidIO system

was able to meet the real-time deadlines for data cube sizes up to 80k ranges, but we also found that a specially-constructed 125 MHz system could also meet the real-time deadlines for most data cube sizes under study when the data source was provided with additional RapidIO ports to meet source-to-sink throughput requirements. Double buffering was of significant benefit for GMTI and helped to allow for scalability in algorithm parameters and system size, in exchange for greater on-node memory requirements.

Our SAR experiments found that the algorithm was less processor- and network-intensive than GMTI, due to its relatively long processing interval (CPI). However, the memory requirements for SAR are much greater than GMTI, forcing the use of a chunk-based approach in which intermediate results are read and written from global memory in between stages. Synchronization and network efficiency were still important issues for this type of parallel partitioning, and performance was found to decline when chunk sizes were made too large or too small. Therefore, it is important to choose a chunk size in the "sweet spot" for minimum network contention and maximum performance. From our SAR experiments, we also determined that the choice of RapidIO logical layer should be based on ease-of-implementation, as the logical I/O and the message-passing layers exhibited nearly equal performance, which can likely be generalized towards most other applications. Double buffering of chunks improved the performance of SAR, but not to the degree that double buffering of entire CPIs was able to improve the performance and double the real-time deadline of GMTI. Double buffering for SAR worked best for small chunk sizes, but as chunk sizes increase network contention begins to negate the benefits

of double buffering. RapidIO switch parameters, method of flow control, and use of logical-layer responses did not significantly affect the performance of SAR.

Finally, we discussed considerations for building systems capable of alternating between performing GMTI and SAR. Most importantly, the systems must be built around the "greatest common denominator" requirements for each application. However, in order to keep system requirements as similar as possible, the GMTI algorithm can be employed with a chunk-based, global-memory approach nearly identical to the one we used for SAR. This approach sacrifices GMTI performance due to redundant data transfer, in exchange for much lower on-node memory requirements. By taking this approach, the GMTI and SAR algorithms and system parameters can be optimized for the same global chunk size. While the work described in this chapter used GMTI and SAR as specific experiments, many of the results may be generalized to other applications with similar communication patterns. Results may also apply to similar systems or networks running GMTI or SAR, even though the work is tailored towards RapidIO-based systems.

CHAPTER 5
FAULT-TOLERANT RAPIDIO-BASED SPACE SYSTEM ARCHITECUTRES (PHASE 2)

In this chapter, we present a research study focusing on fault-tolerant RapidIO architectures for space systems. We provide background and related research on multi-stage interconnection networks, fault-tolerant embedded systems, local- and system-area networks, fault-tolerant RapidIO, and fault models. We propose several novel fault-tolerant architectures for RapidIO-based space systems and quantitatively evaluate the architectures through a unique combination of analytical metrics and simulation studies. The results of this research show several promising architectures in terms of achieving a balance between performance, fault tolerance, size, power, and cost.

Introduction

By nature, the paradigm shift from bus-based designs to switched network designs in space systems introduces numerous design variables related to network architecture and fault tolerance. While typical bus-based systems simply add redundant buses for fault tolerance, it is unclear how to best design a fault-tolerant switched network for space. At one end of the spectrum the entire network might be duplicated, while at the other end it might be possible to simply build a few redundant paths into the network—possibly not requiring even a single extra switch.

For mission-critical systems operating in space's harsh environment, it is paramount that fault tolerance be built into the system at many levels. This fault tolerance must not only be able to mitigate the effects of transient faults due to SEUs and

other anomalies, but also must be able to overcome permanent hardware faults since space systems are relatively inaccessible for repair. Some of the methods for adding fault tolerance include choosing high-reliability components, providing redundant features at the chip-level such as TMR voting, and using software monitoring to detect and recover from faults. However, even the most robust components may eventually experience a critical hard fault that renders the component permanently inoperable. It is for this purpose that component-level redundancy is also necessary in the form of redundant processor boards, system controllers, network interfaces, and other critical components.

From a bus-based networking perspective, component-level redundancy implies having redundant buses with separate physical interfaces connecting each potential bus-user to each of the buses. This method has been traditionally used in networks for space systems and provides fault isolation by literally giving the choice of two separate networks—a primary network and a redundant network. Depending on the system, the network that is not in use (as well as the interfaces to the network) may be deactivated to save power. The analogous approach for switched networks is to duplicate the entire network fabric and provide each network client with an interface to the primary fabric and to the redundant fabric. Figure 5-1 shows a simplified example of a generic space system using redundancy for fault tolerance. Redundant elements in the figure are indicated with white font.

As mentioned previously, the technology level of space-qualified components typically lags several generations behind non-radiation-hardened counterparts for terrestrial applications. Due to this technology gap, standard switched network technologies such as RapidIO are only just starting to be employed in true flight systems.

Therefore, the level of understanding of the tradeoffs in building high-performance, fault-tolerant switched networks for space systems is quite limited. While the basic approach of duplicating the entire network may be a feasible one in some cases, it is likely not the most efficient use of switched technology that can provide redundant bandwidth through additional switches and paths within the same fabric. This phase of research will explore the relevant tradeoffs in building fault-tolerant RapidIO networks for space, including performance, size, power, and cost.

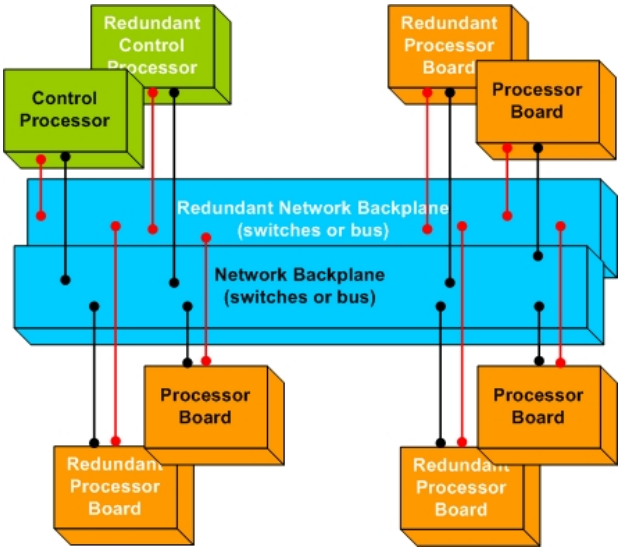


Figure 5-1. Generic payload-processing space system.

Background and Related Research

This section discusses literature from several topics related to building fault-tolerant embedded systems. While no available literature covers all of the topics within this research scope, our research seeks to leverage many ideas covered in the literature on Multi-stage Interconnection Networks (MINs), fault-tolerant embedded systems, Local-Area Networks (LANs), and System-Area Networks (SANs). We also discuss literature related to RapidIO with a focus on fault tolerance. A discussion of fault models that relate to our research concludes the section.

Multi-Stage Interconnection Networks

One important category of literature on fault-tolerant systems relates to Multi-stage Interconnection Networks (MINs). Traditionally, the term "MIN" was used to refer to networks made up of small switching elements (2- to 4-port) arranged in multiple stages. MINs were a very popular research topic through the 80s and 90s, though most of this literature is now outdated because switching elements of today's networks often contain a larger number of ports, which can drastically alter topology. However, in a more general sense, RapidIO networks for space may be considered MINs as long as they employ multiple stages of switches. There is a subset of literature on Clos networks that directly relates to our work. The network design used in the first phase of this research is based on the Clos network [36] in order to provide full bisection bandwidth and non-blocking connectivity. The Fault-Tolerant Clos (FTC) network [51, 52] was proposed by researchers at the New Jersey Institute of Technology in the early 90s and studied throughout the decade. An FTC network is formed by inserting an extra switch in each stage and a set of link multiplexers before two or more of the stages of the Clos network. In the event a switch fails, the multiplexers allow traffic to be routed to a replacement switch that may initially be an unpowered cold spare. This topology allows for the failure of one switch per stage while still retaining equivalent performance to the original Clos topology upon which the particular FTC network was based. Figure 5-2 [51] depicts a Clos network (a) alongside an equivalent FTC network (b). While the figure shows four stages of multiplexers, later variants of the FTC network only use multiplexers before the two stages that directly connect to the system endpoints (i.e., the first-level switches). The reduction in multiplexers is accomplished at the cost of one extra switch port on first-level switches and two extra switch ports on second-level

switches. In the event of a fault, multiplexers as well as switch routing tables may be adjusted, depending on the location of the fault. Our work with FTC networks is based on the later iteration of the FTC network in order to reduce the number of multiplexer components necessary.

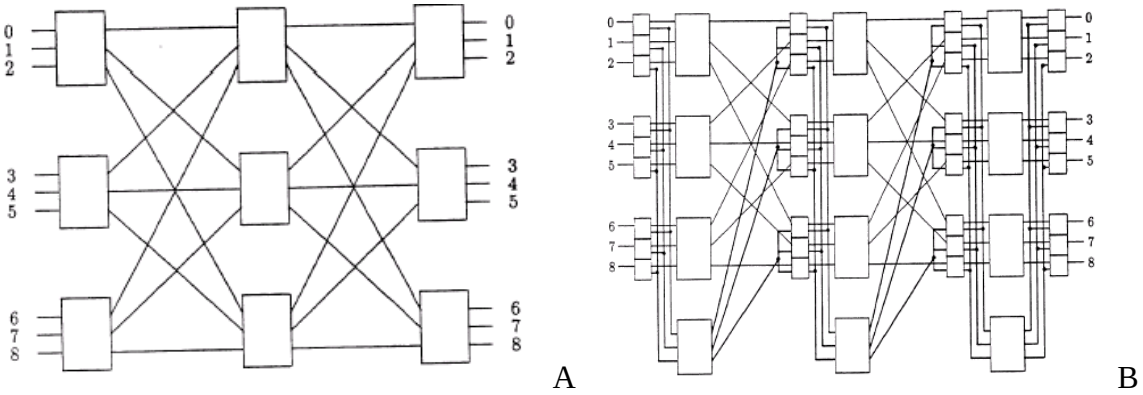


Figure 5-2. Comparison of Clos network variants. A) Clos network. B) Equivalent Fault-Tolerant Clos network [51].

Fault-Tolerant Embedded Systems

There is also a variety of literature related to building fault-tolerant embedded systems. The x2000 Project at the Jet Propulsion Laboratory [53] employs the novel approach of using two different COTS network types (IEEE 1394 and SpaceWire) in an avionics network. Both networks are actively used for unique purposes during normal operating conditions, but in the event of a failure, the system is able to operate with only one of the networks at a degraded performance level. In [54], researchers at the Nottingham Trent University explore ideas with group adaptive routing in embedded systems, which uses alternate paths in order to avoid faults that occur. In [55], IBM and the Loral Federal Systems Group explore their ALLNODE-RT architecture, which is a real-time, fault-tolerant, circuit-switched network. This network accomplishes fault tolerance by defining alternate paths for delivery of deterministic “hot” traffic and

random “cold” traffic. Decisions on which paths to use are influenced by the real-time status of the algorithm. The notion of deterministic traffic versus random traffic is very important in our research and has a significant impact on the various network architectures and techniques under study. Other relevant work is contained in [56], in which Walker from 4Links discusses building fault-tolerant Clos network configurations out of SpaceWire switches. This paper stresses the point that Clos networks have the capability of providing fault tolerance through multiple paths. One main focus is the use of many smaller switching elements to form Clos networks that will experience a very limited performance reduction if a single switch is rendered inoperable. However, the idea of many powered switches in a backplane opposes our desire to keep active switch-count low to conserve power.

Local-Area Networks (LANs) and System-Area Networks (SANs)

Literature on LANs and SANs from the High-Performance Computing (HPC) community can apply to fault-tolerant embedded systems. A SAN is similar to a LAN, with the exception that a SAN generally delivers high bandwidth and low latency, while a LAN may or may not have those attributes. Techniques such as redundancy, link aggregation, and alternate-paths routing have been used in HPC systems for many years, with such applications highlighted in Extreme Networks’ whitepaper [57] for building fault-tolerant Ethernet networks. In addition, research at the Universidad Politecnica de Valencia explores fault tolerance extensions for InfiniBand [58]. Along with the fault tolerance benefits gained by adaptive routing, this work also explores the use of adaptive routing for dynamic load balancing [59]. Of particular interest for our research are the adaptive routing techniques employed in literature for both fault tolerance and performance reasons. Some of these methods include selection from a group of possible

ports via round robin or random methods, or making the decision based on the status of the output buffers for each port [59, 60].

Fault-Tolerant RapidIO

All RapidIO-specific fault tolerance literature comes in the form of whitepapers and product briefs. There is a heavy focus on providing mechanisms for fault tolerance in the RapidIO specification, and there is an additional RapidIO Error Management Extensions Specification [19] that allows extensions to provide additional fault monitoring and recovery capabilities. These capabilities include techniques such as allowing the user to set thresholds for reporting unsuccessful packet transmissions to software via an interrupt. A RapidIO Trade Association application note for building fault-tolerant systems [61] describes the six key elements of fault tolerance for RapidIO as:

1. no single point of failure;
2. no single point of repair;
3. fault recovery;
4. 100% fault detection;
5. 100% fault isolation;
6. fault containment.

The document also describes the ways in which RapidIO helps to provide each of these six key elements through electrical, physical, and logical specifications.

Fault Models

A fault model is used to define the deficiency of a network element that is not functioning correctly, and several fault models have been studied through previous research. Many of the models seen in literature are specifically oriented towards crossbar switches, such as the “stuck-at” fault model [62], or the “losing contact” fault model [63]. The stuck-at model states that a crossbar element is stuck, causing certain switch input

ports to only be routable to a subset of the switch's output ports. The losing contact fault model is similar, stating that two crossbar elements cannot make contact, leading essentially to a stuck-at-open condition. Other more useful fault models for our work include the "link fault" and the "switch fault" [64]. A link fault assumes a link is unusable, while a switch fault assumes the entire switch is unusable. In order to keep this work highly generalizable and independent of the internal architecture of the RapidIO switches, we will use the switch fault as our main fault model. The switch fault model is also the most restrictive of the models, and using this model should lead to the design of architectures with the highest degree of network-level fault tolerance. The link fault model also applies, as the link fault condition is a subset of the switch fault condition.

Overview of Proposed Architectures

This section outlines each of the architectures evaluated as part of this research. We present discussion of each architecture, as well as analytical results describing the basic characteristics of each network design. In order to ensure a fair comparison between architectures and provide a reasonable starting point, we adhere to the following constraints in creating candidate network designs for this research:

- Network design must support non-blocking connectivity for at least 32 RapidIO endpoints under no-fault conditions
- Any system element that interfaces with the RapidIO network must possess separate primary and redundant physical interfaces to the RapidIO network
- 250 MHz Double-Data Rate (DDR), 8-bit parallel RapidIO links for parallel RapidIO systems (4 Gbps)
- 1.25 GHz, 4-lane serial RapidIO links for serial RapidIO systems (4 Gbps)
- Fault model selected is the switch fault
- Network must be able to withstand complete loss of any one RapidIO switch and maintain connectivity to all RapidIO devices

- Upon a switch fault, the routing tables of the remaining switches must be capable of being reconfigured to adjust to the failure

In order to more deeply explore the core issues in building fault-tolerant switched RapidIO networks, this phase of work does not focus around a specific board architecture as we did in the first phase of this research. Instead, the network analysis is performed independently of the mapping of processors and switches to boards and backplanes. The backplane in this study is then defined as the set of all RapidIO switches and multiplexers in the network.

Clos Network Baseline

Our baseline network design is based upon the Clos network and is shown in Figure 5-3. While the basic Clos architecture contains many redundant paths between nodes connected to different first-level switches, it contains no explicit switch-level redundancy, and the loss of any first-level (edge) switch disconnects the four devices directly connected to that switch from the rest of the network. Upon the loss of a second-level (core) switch, all RapidIO devices maintain full connectivity to each other (assuming routing tables are altered to avoid the fault), but the network bi-section bandwidth is reduced and the network no longer can maintain non-blocking connectivity between all RapidIO devices. In order to provide fault tolerance for the baseline design, a completely redundant network must be present in the system and connected to the redundant interfaces of each networked system element. Upon any failure in the primary network, the primary interfaces and network must then be powered down and the redundant interfaces and network must be powered up. Upon a failure in the activated redundant network, the system is then forced to operate in a degraded condition, if possible (using either the primary or redundant network). The type of degraded operation

will depend on the faulty switch in the active network, as described above. The redundant network is not shown in Figure 5-3, but is identical to the primary network shown.

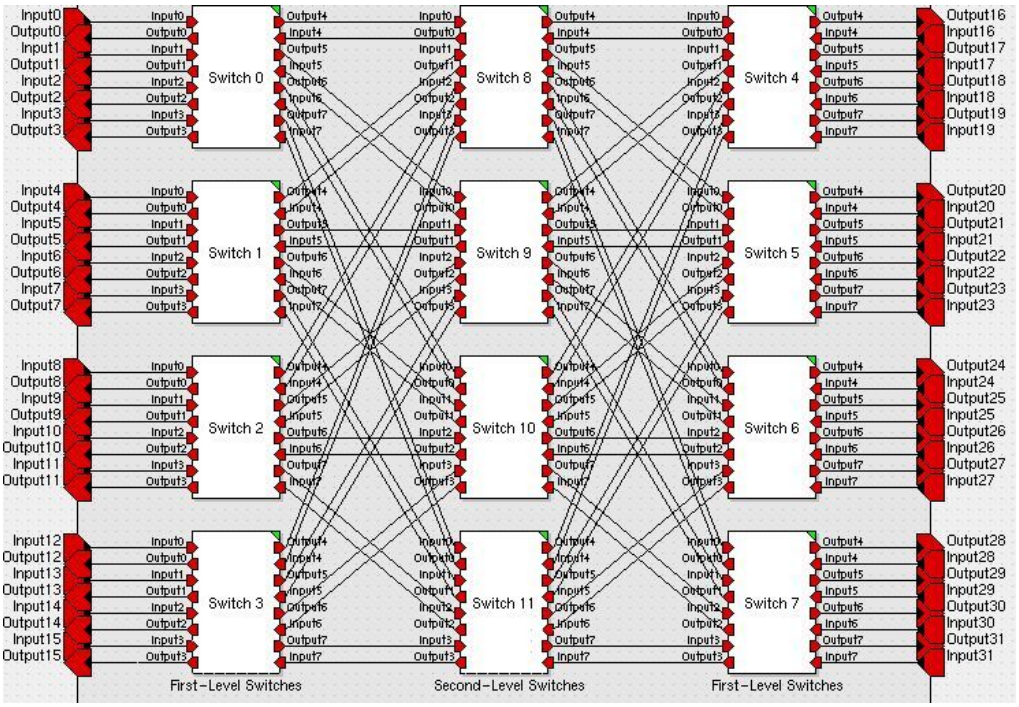


Figure 5-3. Baseline Clos network architecture.

Table 5-1 provides an overview of the basic characteristics of our baseline architecture. A similar table will be provided for each architecture introduced in this chapter. The "Active Switches" and "Standby Switches" table entries tell how many powered and unpowered switches, respectively, are used in the system prior to any faults occurring, while "Total Switches" is the sum of these two metrics. "Active Ports per Switch" is an average of how many ports are active on each switch that is active. Inactive switches do not contribute towards this average. All of the switch-count and port-count metrics are useful in determining the size and cost of an architecture. In addition, the "Active Switches" and "Active Ports per Switch" metrics are critical in determining the power consumption of an architecture. "Mux Count" indicates the

number of multiplexers that are needed for the network design. We will provide more detail on this metric when we examine network designs using multiplexers. The "Number Switches to Reroute-1" and "Number Switches to Reroute-2" metrics describe the number of switches that will require routing tables to be changed in the event of a fault in the first or second level, respectively. Note that an inactive switch that must be powered up is not counted as a rerouted switch unless its routing table is not known prior to the occurrence of the fault. These metrics help provide a measure of fault isolation, as a fault in one part of a system should ideally not affect other parts of the system. A fully-redundant network approach such as the one described here provides ideal fault isolation at the cost of many wasted resources. The metrics described here will be explored in more detail and their implications will be discussed later in this chapter when we draw comparisons between all of the architectures under consideration.

Table 5-1. Characteristics of baseline Clos network.

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	12	24	8	192	0	0	0

Redundant First Stage Network

Figure 5-4 shows a Clos-like network design with a redundant first stage. Under no-fault conditions, the network is functionally identical to our Clos baseline. However, upon failure of any first-stage switch, a corresponding standby replacement switch will then be powered on and used as a replacement. This replacement switch directly connects to four redundant RapidIO endpoints of system end nodes; therefore, the redundant RapidIO endpoints of the appropriate nodes will also be activated and the primary RapidIO endpoints of those same nodes will be deactivated. Each multiplexer device shown in Figure 5-4 allows eight RapidIO links to be multiplexed down to four

RapidIO links (i.e., an 8:4 multiplexer) and swaps connectivity between a primary or redundant first-level switch and a second-level switch. Note that LVDS multiplexer devices such as the ones described here are currently commercially available [65] and would simply need to be integrated into a custom, radiation-hardened Application-Specific Integrated Circuit (ASIC) for a RapidIO production network.

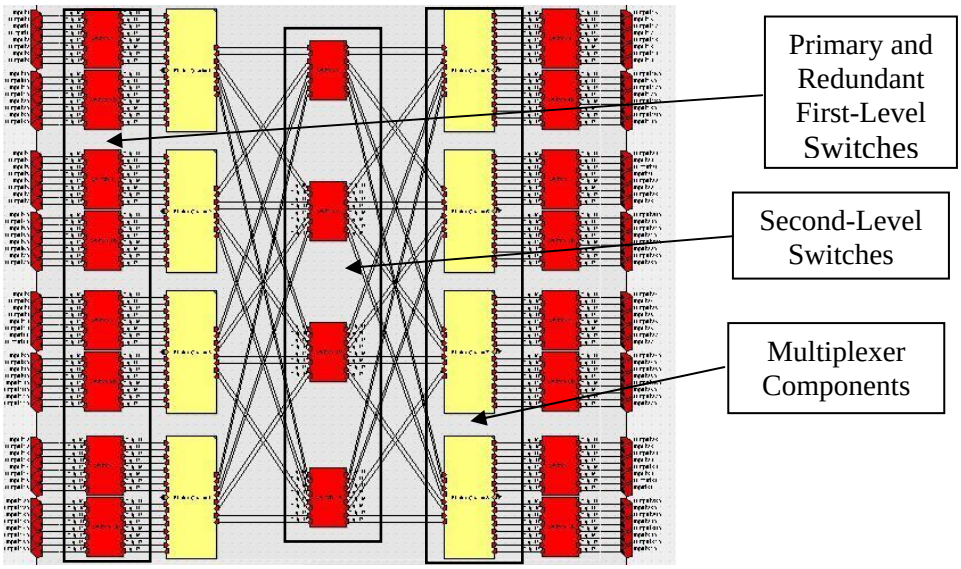


Figure 5-4. Redundant first stage network.

Table 5-2 shows a summary of the important characteristics of the network in Figure 5-4. Note that a fault in the second level requires all of the active first-level switches to be rerouted to avoid sending packets to the faulty switch. While no connectivity is lost, the network suffers a 25% loss of core bandwidth, and fully non-blocking connectivity is no longer possible. A fault in the first level requires no switches to be rerouted, but does require the reconfiguration of a multiplexer device to establish a connection to the backup switch.

Table 5-2. Characteristics of redundant first stage network.

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	8	20	8	160	8 (8:4)	0	8

Redundant First Stage Network with Extra-Switch Core

In this section, we take a unique approach to fault tolerance in that we explicitly segment the network into stages and handle fault tolerance differently in different stages of the network. Variations on this approach will also be used in several other network designs under study in this chapter. Figure 5-5 depicts a network with a redundant first stage, but with an additional second-level (core) switch. To enable the fifth core switch, an additional port is needed on each first-level switch, for a total of nine ports per first-level switch. Each of the eight multiplexer components in the system also must now multiplex 10 RapidIO links down to 5 (10:5) rather than 8 down to 4 (8:4—as in the network shown in Figure 5-4). The fifth switch in the second level provides redundant bandwidth and may be powered or unpowered under no-fault conditions. Second-level switches still require only eight ports.

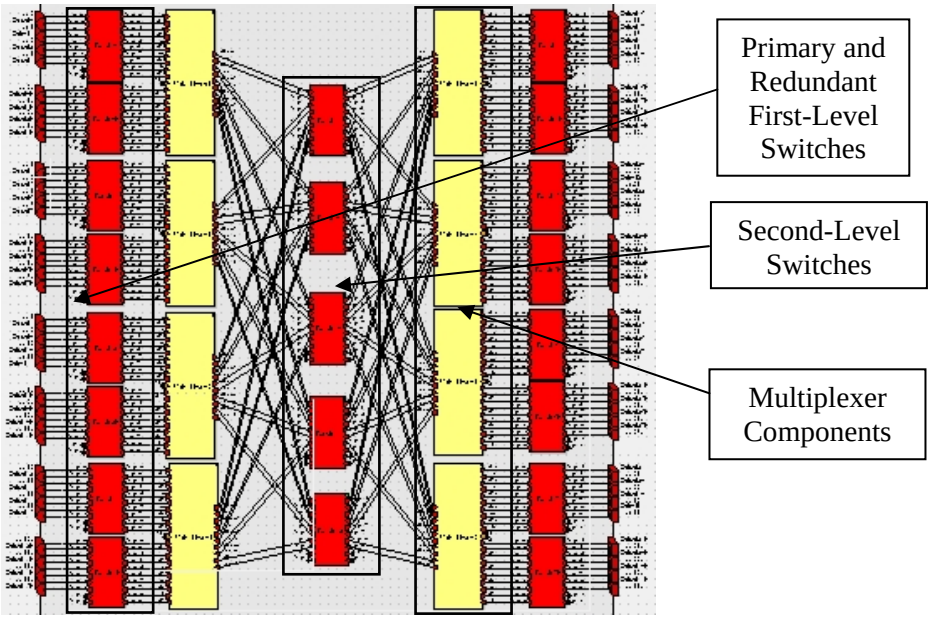


Figure 5-5. Redundant first stage network with extra-switch core.

Table 5-3 shows the important characteristics of the redundant first stage network with extra-switch core. The table assumes one of the five core switches is an unpowered spare and only used in the event of a failure of another second-level switch. Later in this chapter, we will use simulation results to help justify this use of the extra switch, as well as use simulation to explore the implications of using the additional switch as a powered spare. In theory, under this configuration this network should be able to withstand the loss of at least a single first-level switch and a single second-stage switch with no sacrifice in performance. Again, in the case of a first-level switch fault no switches must be rerouted, but a multiplexer device's configuration must be altered to allow connection to the redundant first-level switch.

Table 5-3. Characteristics of redundant first stage network with extra-switch core.

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	9	21	8	184	8 (10:5)	0	8

Redundant First Stage Network (Serial RapidIO)

One factor that may introduce complexity into the previously-examined networks is the introduction of the multiplexer devices for first-level switch failover. However, a similar architecture can be created and the use of multiplexers can be avoided if switches with larger port-counts can be used in the second level. While parallel RapidIO switches will not scale far above eight or ten ports due to pin-count issues, a serial RapidIO switch can have many more ports since each port uses a much smaller number of pins. For this research, we use a conservative estimate and assume that the first generation of space-qualified serial RapidIO switches will be capable of reaching a port-count of 16 when each serial port is a four-lane port (16 pins). The novel network architecture shown in Figure 5-6 uses 8-port serial switches in the first level and 16-port serial switches in the

second level. Only eight of the ports on the second-level switches need to be active at any one time. These eight active ports connect each switch to the eight active first-level switches.

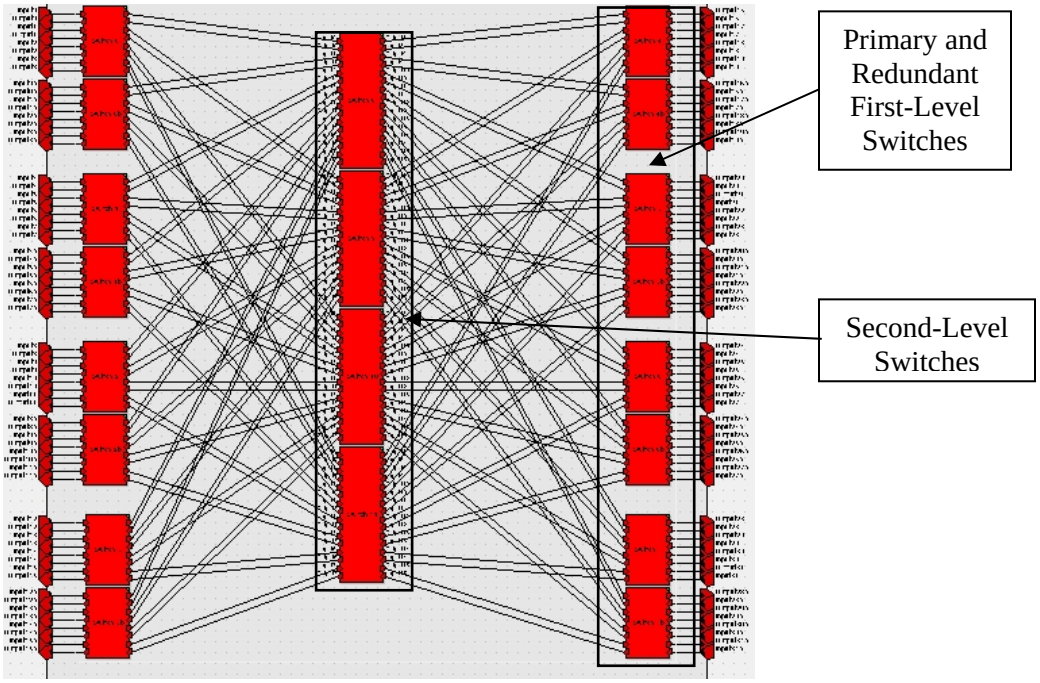


Figure 5-6. Redundant first stage network (serial RapidIO).

Table 5-4 shows a summary of the important characteristics of this network architecture. In the event of a first-level switch failure, the correct standby switch must be powered on and the routing tables of each second-level switch must be reconfigured to use the replacement switch rather than the faulty one. This activity represents a slight tradeoff in fault isolation from the similar multiplexer-based architectures previously examined in this chapter, since rather than simple multiplexer reconfiguration (between one of two options), four switch routing tables must instead be changed.

Table 5-4. Characteristics of redundant first stage network (serial RapidIO).

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	8	20	8	192	0	4	8

Redundant First Stage Network with Extra-Switch Core (Serial RapidIO)

The network shown in Figure 5-7 combines elements from the previous two networks discussed in this section. Sixteen-port serial RapidIO switches form the second level, in order to facilitate the inclusion of a redundant first stage without the use of multiplexer devices. However, this design also includes an extra switch in the second level in order to provide redundant throughput for use in the case of a second-level fault. The addition of the extra switch requires the inclusion of an extra port on each first-level switch, raising the total port requirement for each of those switches up to nine ports. The extra second-level switch also has potential to be used on an active basis even in zero-fault cases to provide additional routes through the network to avoid congestion.

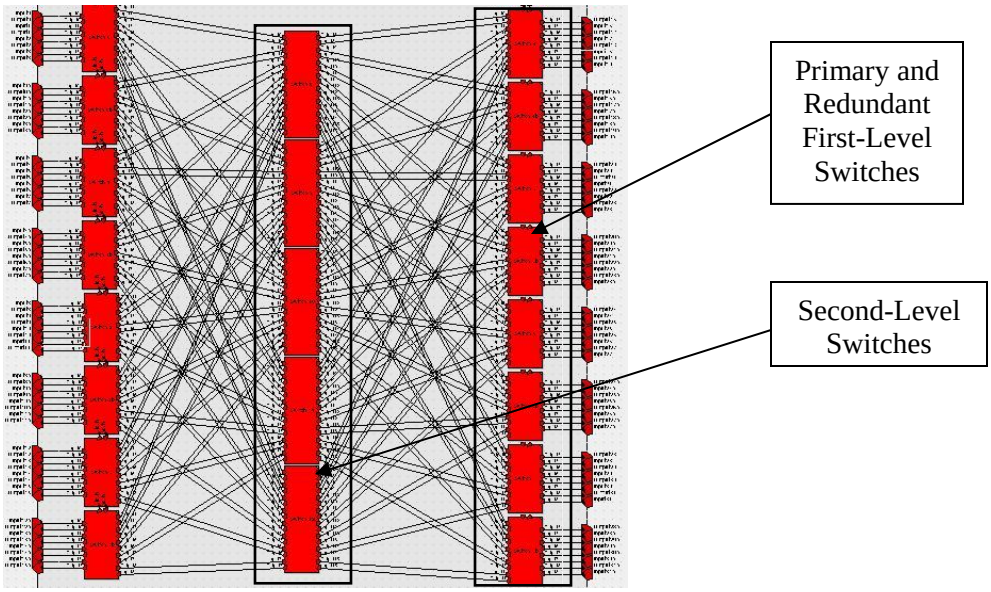


Figure 5-7. Redundant first stage network with extra-switch core (serial RapidIO).

Table 5-5 shows a summary of the basic characteristics of this architecture, assuming that one second-level switch is used as an unpowered spare. As with our other serial RapidIO-based architecture, the active second-level switches must be rerouted in the event of a first-level switch fault.

Table 5-5. Characteristics of redundant first stage network with extra-switch core (serial RapidIO).

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	9	21	8	224	0	4	8

Fault-Tolerant Clos Network

While the Fault-Tolerant Clos (FTC) network was originally designed for circuit-switched systems, we have adapted this network for the packet-switched, bi-directional communication associated with a RapidIO network. Figure 5-8 depicts a RapidIO-based FTC architecture that under no-fault conditions is equivalent to the previously-studied networks in terms of number of endpoints supported and bi-section bandwidth. This architecture can be constructed using a combination of parallel RapidIO switches and multiplexer devices. First-stage switches must have at least 9 ports, and second-stage switches require 10 ports. For each of the two groups of four first-stage switches, a single unpowered first-stage switch serves as a redundant spare. This redundant switch connects to the redundant network interfaces of 4 of the 16 the RapidIO endpoints on that side of the network through a set of four 4:1 LVDS multiplexer devices. The multiplexers allow a single redundant switch to serve as a potential backup to up to four primary switches, and this particular system could theoretically suffer two first-level switch faults with no drop in performance, provided the first-level switch faults were not in the same group of four switches. In order to support the redundant first-level switches, this network inherently provides an extra-switch core. As with the previous extra-switch core designs, this extra switch may be powered or unpowered under no-fault conditions, depending on system and application requirements.

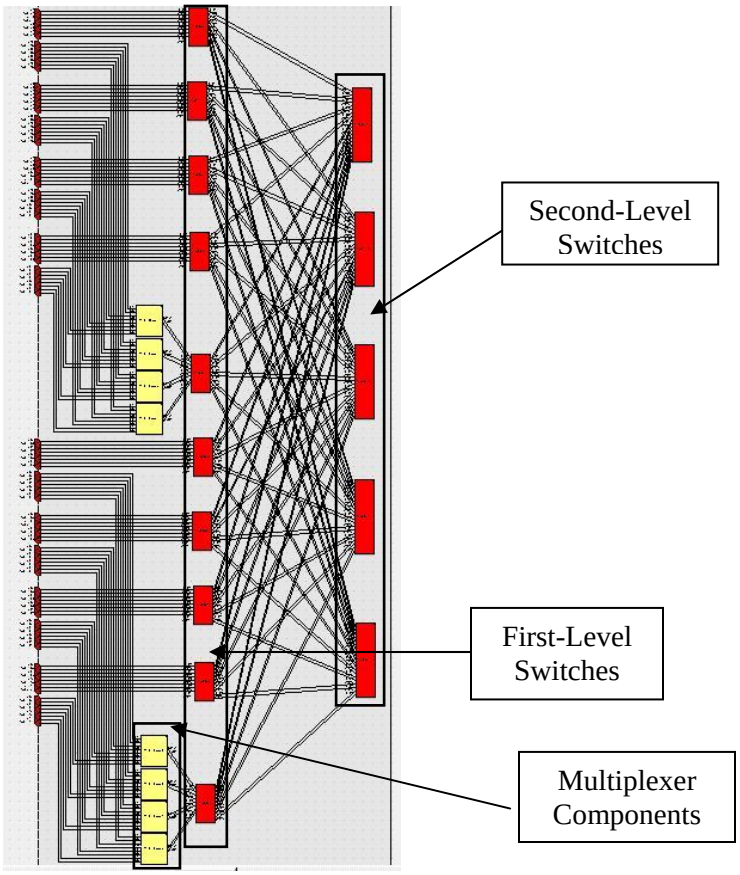


Figure 5-8. Fault-Tolerant Clos network.

Table 5-6 displays a summary of the important characteristics of our RapidIO FTC network, assuming that one second-level switch is used as an unpowered spare. One major advantage of this architecture over the other multiplexer-based architectures studied in this chapter is that the multiplexers in this system do not represent a single point of failure under no-fault conditions. While the multiplexer devices are much simpler than RapidIO switches, it is likely that there is still some risk of failure of one of these devices. For the other systems, the failure of a multiplexer device implies that four primary RapidIO endpoints and their four redundant endpoints will no longer be able to connect to the network. Therefore, these other systems must possess at least some tolerance for the loss of four devices through redundancy or graceful performance

degradation. In this FTC-based network, a failed multiplexer has absolutely no effect on the system unless a first-level switch has failed. The tradeoff for these advantages of the FTC network is the reduced level of fault isolation, indicated in Table 5-6 by the relatively high number of switches whose routing tables must be altered in the event of a first-level fault. A first-level fault affects each of the second-level switches, and also requires a routing table to be determined for the standby switch that is being activated.

Table 5-6. Characteristics of Fault-Tolerant Clos network.

Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
12	3	15	8	140	8 (4:1)	5	8

Simulation Experiments

In this section we describe the simulation experiments used to aid in the evaluation of the fault-tolerant RapidIO architectures we have presented. We first describe the benchmarks selected to stress these architectures and justify the selection of each benchmark. We then present simulation results and discussion comparing and contrasting the architectures. In addition, we present results of a study contrasting a variety of methods that may be employed to provide adaptive routing in RapidIO systems for load balancing or fault tolerance.

Overview of Benchmarks and Experiments

In order to draw practical conclusions about the performance of the architectures under study, we have selected a set of microbenchmarks and benchmark kernels to stimulate our simulation models. Each benchmark is mapped to the simulation environment using a scripted approach. Using this approach, processor models follow a set of instructions read from a script file at simulation runtime. This file gives the processing elements in the system a sequence of events to follow, and most instructions

are geared towards generating the correct communication pattern for a given application. Computation modeling is handled through an instruction that delays the processing element from performing other functions for a specific amount of time.

Depending upon the experiment, one or more benchmarks are used to stress the simulated network and help provide insight into the performance of the network when running a real-world application. The complete list of benchmarks used is given below, along with a description of each benchmark.

- Matrix multiply: We use a 28-processor matrix multiply benchmark, performed out of a global memory with four RapidIO ports. The four global memory ports connected to the network fabric leave the maximum of 28 processors with RapidIO endpoints to be used for computation. At simulation start, each processing element reads a portion of each matrix from the global memory, performs computation (i.e., delays), and then writes results back to global memory. The primary communication pattern associated with this benchmark is many-to-few, and this pattern is common among parallel and distributed master-worker type tasks as well as global-memory-based processing for tasks such as SAR.

- Synchronized corner turn: Our corner turn benchmarks simulate personalized all-to-all communication between each active processing element, where a unique data payload of adjustable size (100 KB default) is sent from each processor to every other processor in the system. We have 16-, 24- and 32-processor versions of this benchmark, with the active processors in each case selected such that utilization is balanced throughout the network as much as possible. For the synchronized corner turn, each node executes a barrier synchronization after each data transfer of each corner turn. This synchronization helps reduce contention by ensuring the processors remain on a schedule such that no two processors are ever trying to send a large amount of data to a common third processor.

- Unsynchronized corner turn: This benchmark is very similar to the synchronized corner turn, with the exception that there are no barrier synchronizations in between the data transfers composing each corner turn. This benchmark may be prone to network contention especially when data transfer sizes are large and the network does not provide full bi-section bandwidth for all processing elements.

- Random sends: This benchmark causes each processing element to send a specified number of data transfers of a specified size to random destinations using the RapidIO message-passing logical protocol. After each data transfer, each processing element delays long enough so that the next transfers begin with no other traffic currently in the network. Therefore, traffic generation is on a synchronous basis, with each processing element creating a single transaction at the

same time. The random destinations are chosen on a per-transaction basis at the time the benchmark script is generated.

- Random reads: This benchmark causes each processing element to issue a specified number of RapidIO read requests of a specified size to random destinations using the RapidIO logical I/O protocol. For each processing element, each read request must be filled with a read response before the next read request can be generated. This causes a moderate amount of network traffic to be generated, but the traffic generated by each node is not synchronized with respect to the traffic generated by other nodes. The random destinations are chosen on a per-read basis at the time the benchmark script is generated.

Matrix Multiply Results

Our first simulation experiment uses the matrix multiply benchmark to examine the capabilities of each network under study. The benchmark simulates a 512 by 512 matrix multiply, with 8 bytes per matrix element. We use a 28-processor version of the algorithm, performed out of a global memory with four RapidIO endpoints. We examine the performance of each architecture in 0-fault, 1-fault, and 2-fault cases, with each fault occurring in the second level (i.e., core level). In all fault cases explored in this research, faults are injected statically prior to simulation runtime by disabling one or more switches and loading adjusted routing tables to affected switches (if any). First-level fault cases are trivial due to first-stage component redundancy in all architectures, although the architectures provide the redundancy in different ways. The merits of each approach will be explored in a later section combining our analytical and simulative results.

The results of the matrix multiply experiment are shown in Figure 5-9. Assuming four active core switches in each network, all networks in this study are essentially the same under no-fault conditions, and thus exhibit equivalent performance. With the occurrence of one fault, the baseline architecture is able to maintain this original level of performance by switching to a completely redundant network that is free of faults. In addition, the networks with an extra-switch core—including the Fault-Tolerant Clos

network—are able to maintain a nearly equal level of performance by activating a standby core switch to replace the faulty switch. The networks without redundant core bandwidth experience a drop in performance with one fault, and all networks experience a drop in performance with a second fault.

In the two-fault baseline case, the system has two isolated networks, each with one fault in the second level. Either network may then be used in a degraded mode of operation. The result shown indicates the performance of the baseline Clos network architecture using only three active second-level network switches. All two-fault scenarios experience some degraded level of performance in this experiment, but the architectures with an extra-switch core—as well as the baseline architecture—showed the best performance under two-fault conditions.

It should be noted that the results for the matrix multiply benchmark exhibit clear trends, but the overall execution time for the benchmark does not vary greatly even in the presence of multiple faults for any of the architectures examined. Further examination of this benchmark reveals that performance is largely dictated by contention for the four-port global memory. The removal of a core switch does not greatly impact the overall execution time, since the four global memory ports can not supply data fast enough to place a significant strain on the network backplane. These results show that for applications where full network bi-section bandwidth is not necessary, even RapidIO networks with a modest level of fault-tolerance can likely continue to meet performance needs as long as that fault tolerance is provided in an effective manner.

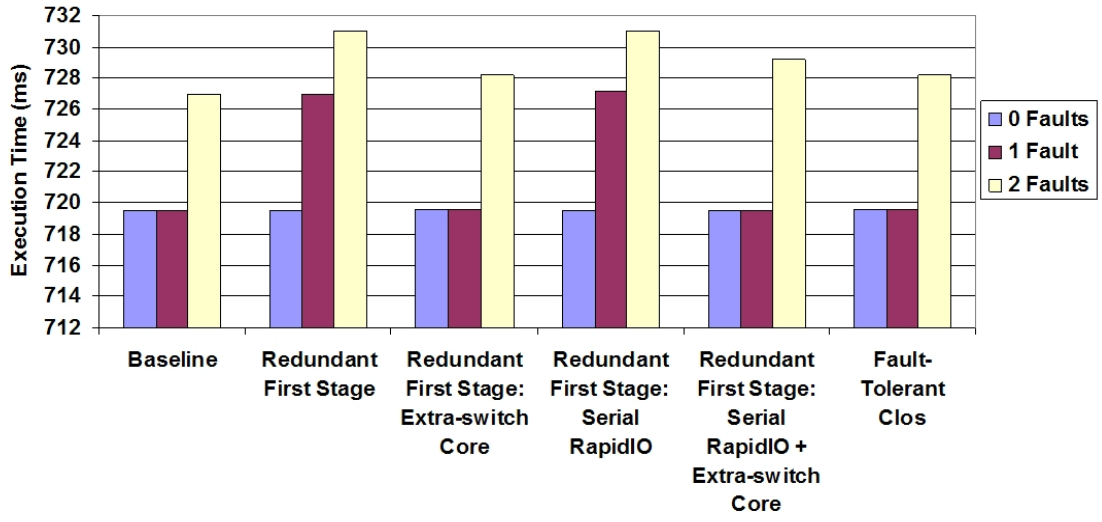


Figure 5-9. 28-processor matrix multiply results.

Corner Turn Results

Our corner turn experiments are designed to generate a high level of stress on the network fabric through the use of personalized all-to-all communication. Figure 5-10 shows the results of the 16-processor synchronized corner turn. In this benchmark, each processor sends 100 KB of data to each other processor. Barrier syncs occur after each transfer in order to keep the scheduling such that no node is receiving from more than one other node at any given time. The results show a vastly wider spread in performance of the different configurations when compared to the matrix multiply results. However, many of the overall trends observed in the matrix multiply results do apply to this experiment. Essentially, these trends have simply been amplified by the much larger amount of traffic causing greater stress on the network. The extra-switch core designs (as well as the fully-redundant baseline) excel in the 1-fault case, while architectures without built-in core redundancy suffer greatly as the corner turn requires non-blocking connectivity to operate efficiently. Figure 5-11 shows the results of the 32-processor

synchronized corner turn. Again, the previously-noted trends continue, but this time with an even greater disparity between the architectures with and without adequate core bandwidth in the presence of faults.

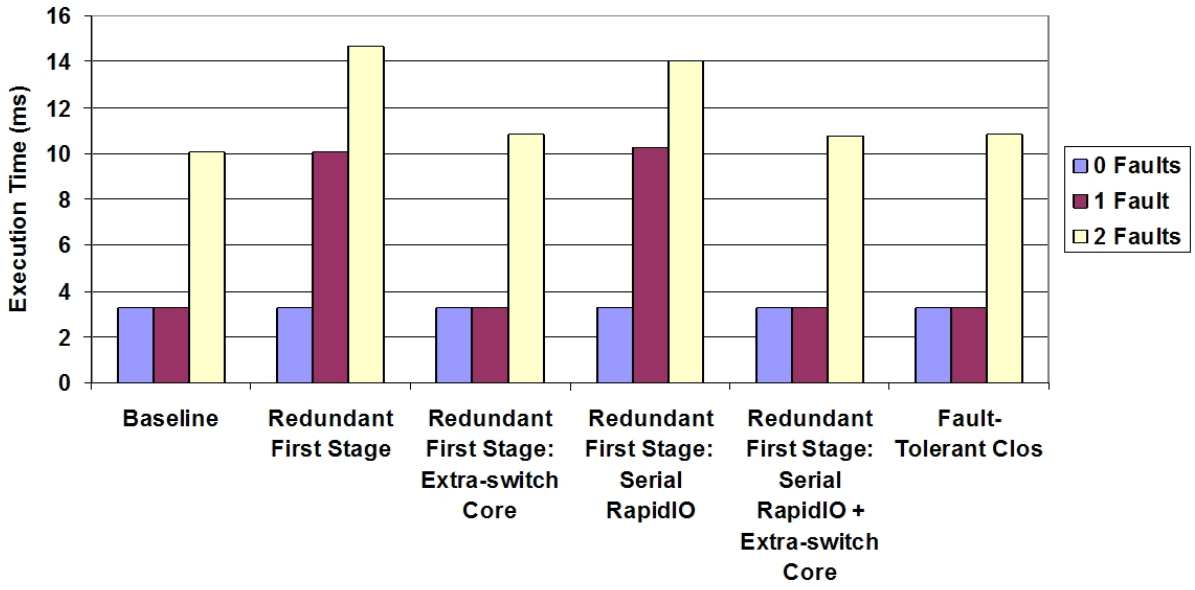


Figure 5-10. Synchronized 16-processor corner turn results.

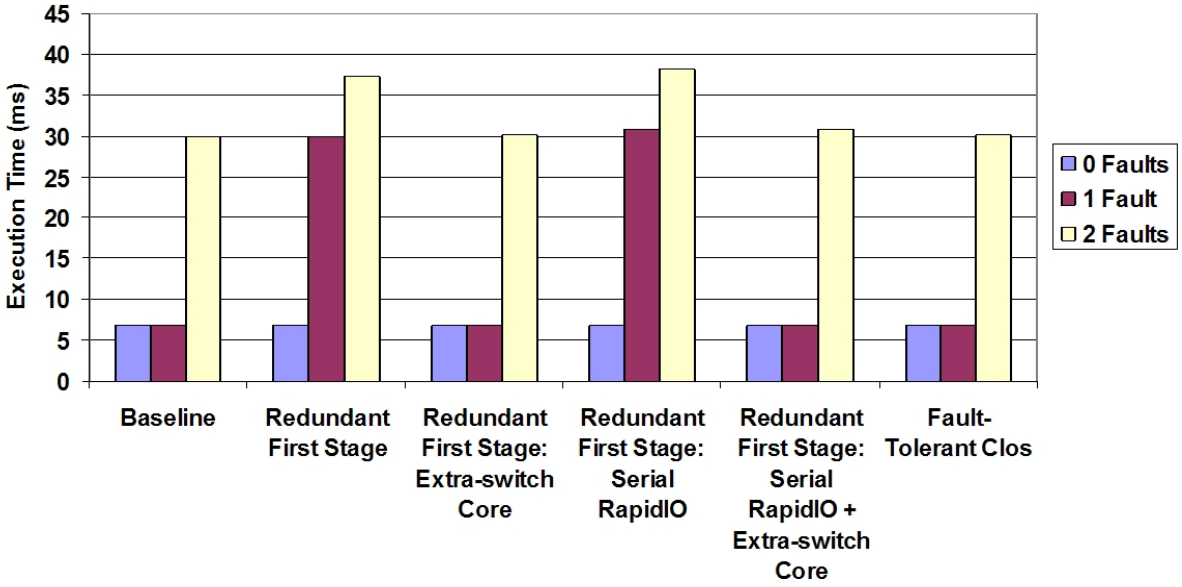


Figure 5-11. Synchronized 32-processor corner turn results.

While the baseline network architecture has performed well in the 1-fault cases examined thus far, it is also important to note that the baseline network does not have any switch-level redundancy in the first network stage. This lack of first-level redundancy may present problems during certain multiple-fault combinations. For example, a single fault in a first-level switch of the primary network and a single fault in a second-level switch in the redundant network could present a serious problem if no board-level redundancy has been considered in the system's design. Scenarios such as this will be further accounted for in the combined simulation/analytical results in a later section of this study.

Figure 5-12 displays the results of a comparison between the synchronized and unsynchronized corner turn benchmarks. This comparison was performed using the standard redundant first stage network with the same 100 KB per-processor data transfer size as the previous corner turn experiments. We performed the benchmark for systems of 16, 24, and 32 processors. Under no-fault conditions, the synchronized and unsynchronized corner turns perform very similarly, with a slight edge in performance actually belonging to the unsynchronized case. In the absence of faults in the unsynchronized case, the corner turns are able to maintain their schedule without the need for intervening barrier synchronizations between corner turn data transfers. This efficiency of the unsynchronized case can be attributed to the fact that the only type of traffic present in the network is the corner turn traffic, and the traffic is symmetric in its use of the network and able to stay on schedule without frequent synchronization since there is no network contention. The absence of contention in the no-fault cases is evidenced in Figure 5-12 by the unity slope of the no-fault data lines as the number of

processors increase—with no contention, the 32-processor systems take twice as long to perform a corner turn as the 16-processor systems, since each processor must send to approximately twice as many other processors, and each processor performs its operations concurrently. However, previous experiments with RapidIO networks for GMTI space-based radar [45] have shown that the corner turn traffic benefits greatly from synchronization when other traffic flows are present in the network, and the corner turn traffic can not stay perfectly synchronized without intervention. Similarly, the 1-fault and 2-fault cases in this experiment introduce high amounts of contention into the network, and the efficient use of the under-provisioned network becomes very important to achieve maximum performance from the corner turn benchmark. As the number of processors increases, the importance of carefully using the RapidIO network also increases. In fact, for 24- and 32-processor cases, the synchronized 2-fault system actually performed the corner turn faster than the 1-fault system with no synchronization. We performed similar comparisons between synchronized and unsynchronized corner turns using the other systems in this study and observed similar trends, but those results are omitted for conciseness.

Our final experiment with the corner turn benchmark examines the performance of the extra-switch core networks when using the fifth core switch on an active basis rather than using it as an unpowered standby. Figure 5-13 displays the results of the synchronized corner turn benchmark performed on Fault-Tolerant Clos networks with four and five active core switches. In this case, adding the fifth core switch actually hurts performance because the algorithm had been scheduled and synchronized for the four-switch configuration. For applications such as the corner turn that may be effectively

statically scheduled, the active five-switch core presents limited benefits. Even a version optimized for the five-switch core could at best only hope to achieve equal performance to the four-switch version. Therefore, the extra switch should be left inactive in most cases in order to save power and simplify scheduling. This experiment has also been performed on the other extra-switch core systems with similar results that are omitted for conciseness.

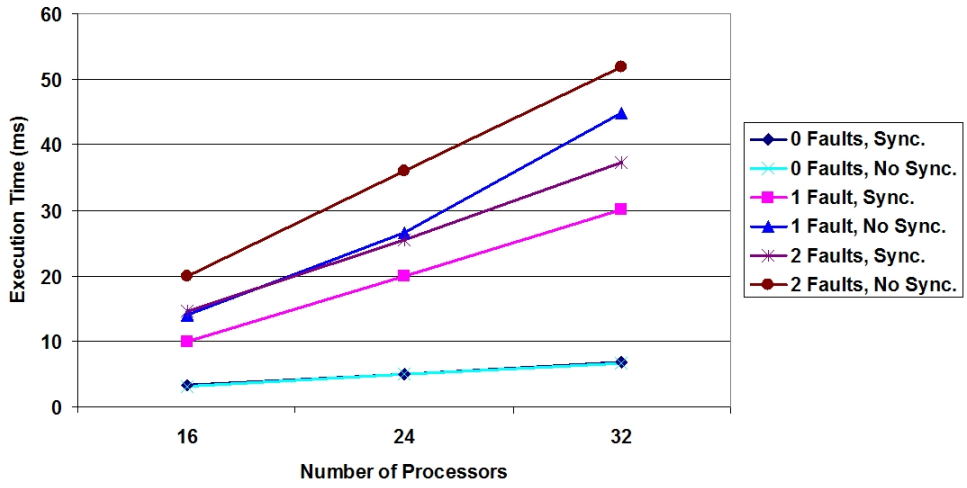


Figure 5-12. Synchronized versus unsynchronized corner turn results (redundant first stage network).

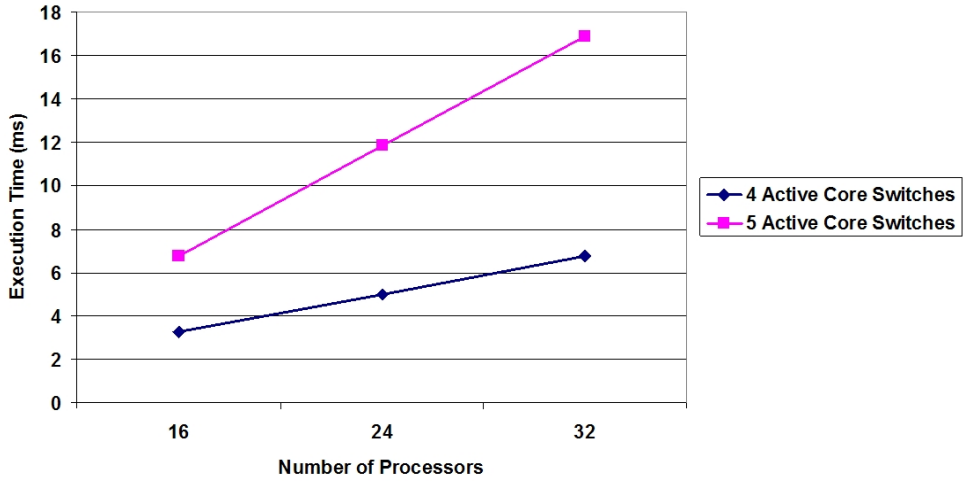


Figure 5-13. Synchronized corner turn with four- and five-switch core (Fault-Tolerant Clos network).

Adaptive Routing Results

In order to attempt to gain maximum performance and load balance from the networks under study, we expanded the RapidIO switch model's capabilities to support adaptive routing. Adaptive routing [54, 59, 60] allows a packet destined for a specific node to leave out of one of a set of possible output ports. A typical switched RapidIO implementation handles routing through the use of routing tables that map a packet's destination ID to a single output port on the switch. However, adaptive routing allows a packet with a given destination ID to leave from one of several ports on a switch (a one-to-many mapping). The RapidIO specification [16] does allow for adaptive routing to improve fault tolerance or load balancing, as long as routing decisions are made by the switches and take the destination ID of the packet into consideration. In addition, switches may consider internal factors, such as the depth of various buffers, when making routing decisions. One negative aspect of adaptive routing is that in-order delivery of a flow of traffic from one source to one destination can not be guaranteed. If in-order delivery for a series of transactions is required, additional destination IDs can assigned to a node and static paths can be assigned to those IDs in each switch's routing table [59]. In other words, one physical node in the system may have multiple destination IDs—one for in-order traffic that is always assigned a one-to-one mapping in switch routing tables, and one for packets that may be delivered out of order due to adaptive routing.

In order to keep our solution compliant with the RapidIO protocol, as well as simple enough to be implemented in an embedded system where chip real-estate is at a premium, we restrict the adaptive routing methods tested to the isolated adaptive routing type [60]. Isolated adaptive routing uses only information locally available at each switch to make routing decisions, and does not require the exchange of information

between switches. We incorporated three tactics for isolated adaptive routing into our RapidIO switch models, and refer to them as “shortest-buffer routing”, “round-robin routing”, and “random routing.” These tactics are similar to methods studied in literature [59, 60], but have been adapted to embedded systems by combining routing tables and adaptive selection techniques. These adaptive routing methods all use routing tables, but rather than a standard mapping of each destination ID to a single port, the adaptive routing tables map each destination ID to one or more ports. Because our Clos-based networks are all regular, symmetric topologies with a relatively small switch-count, there is not a large need for more complicated methods such as shortest-path distributed adaptive routing techniques. An explanation of each adaptive routing method used for this work is as follows:

- Shortest-buffer routing: When the RapidIO switch receives a packet, it examines the destination ID of the packet and looks up the ID in its routing table. The routing table returns a list of possible ports the packet may exit through, and the switch determines the current number of packets buffered to exit through each of the ports. The port with the shortest buffer depth is then chosen for the packet’s path. In the event one or more ports are tied for the shortest buffer depth, the port is chosen at random. This tactic may be considered a subset of the port bias and buffer depth technique [60] that chooses an output port based on the smallest sum of a pre-assigned weight and the depth of the port’s queue. In our case, the biases of each port in the list are equal, and the other ports have an infinite bias. Biases are not necessary in our regular, Clos-based networks due to their symmetry and small size.

- Round-robin routing: Upon packet arrival at a RapidIO switch, the list of possible destination ports for the packet is determined as previously described. The destination port is chosen in strict round-robin order from each distinct list of destination ports in the routing table.

- Random routing: Random routing is similar to round-robin routing, except that the destination port is chosen at random from the list.

In terms of the shortest-buffer routing tactic, it is important to note that preliminary experiments showed that deterministic selection of the first shortest-buffer port identified

tends to create an unbalanced load on the network and degraded performance; hence, the shortest-buffer tactic used in this study will always randomly select a port from the list of shortest-buffer ports when two or more ports are tied for the shortest-buffer depth.

Our adaptive-routing experiments will consider the Fault-Tolerant Clos (FTC) network configuration, which previous experiments have shown is representative of any of the extra-switch core networks under study in this research. We will consider FTC networks with three-, four-, and five-switch second-level network stages. Note that each system is structured as shown in Figure 5-8, with the only difference between the systems being the number of second-level switches that are powered and active (with the maximum number being five). Unpowered switches may be considered faulty, or may be considered to be inactive for purposes of cold-sparing or power saving. The five-switch system technically is over-provisioned and has more available bandwidth than the 32 RapidIO endpoints in the system are able to consume at once, while the four-switch system is evenly provisioned, and the three-switch system is under-provisioned. Full connectivity between all endpoints is possible in all cases. However, the redundant network bandwidth of the five-switch system provides extra paths that may be useful in cases where traffic can not be easily statically scheduled to use a system that is evenly provisioned or under-provisioned. In cases where adaptive routing is used, adaptive routing decisions are only made at the first-level network switch that initially receives a packet from a source endpoint. For a packet that must travel to an endpoint connected to a different switch, this first-level switch may then choose from up to five possible output ports (one per second-level switch), which effectively fixes a packet's path through the network. The second-level switch must relay the packet directly to the first-level switch

that connects to the destination endpoint. In the case where the source and destination endpoint are connected to the same first-level switch, no adaptive routing decisions are made and the packet travels through only one hop to reach its destination. Other routing methods (such as purely random routing with no routing table) may use more than three hops for a packet to reach its destination, but these methods are more suited for large networks than a tightly-coupled embedded system where most shortest-hop paths should be more efficient than all other paths.

Our first experiment uses our original synchronized corner turn benchmark on an FTC network configuration with various adaptive routing techniques compared to our fixed-routed baseline. Results of this experiment are shown in Figure 5-14. Note that that the five-switch result for the fixed-routed system actually uses four active switches and is presented for comparison to the adaptive methods, since we have shown in a previous experiment that the five-switch core does not benefit a fixed-routed corner turn. In the five- and four-switch cases, fixed routing actually outperformed the adaptive cases. The good performance of fixed routing for these systems is due to the relatively efficient static scheduling of the corner turn application that minimizes network contention. Adaptive routing performed nearly identically to the static routing in the five-switch case, but performance was slightly degraded in the four-switch case.

The three-switch case, however, creates much contention due to the severe under-provisioning of the backplane for the all-to-all communication of the corner turn. For this configuration, static routing is unable to provide the simple efficiency seen in the five- and four-switch cases. The adaptively-routed systems all outperform the fixed-routed configuration in the three-switch case, with random and round-robin routing

providing the best performance. The shortest-buffer technique, however, is actually slightly detrimental to performance compared to the other adaptive routing techniques for this benchmark. One problem with isolated adaptive routing techniques is that a given network switch does not take the status of other switches or endpoints into account. In our FTC network configuration—as well as other Clos configurations—adaptive routing decisions may only be made by first-level switches if a packet is traversing a three-hop path through the network. Therefore, the decision made at the first switch may not be the best overall decision to get a packet to its destination, especially when the network is under extreme contention introduced by the corner turns and bandwidth under-provisioning.

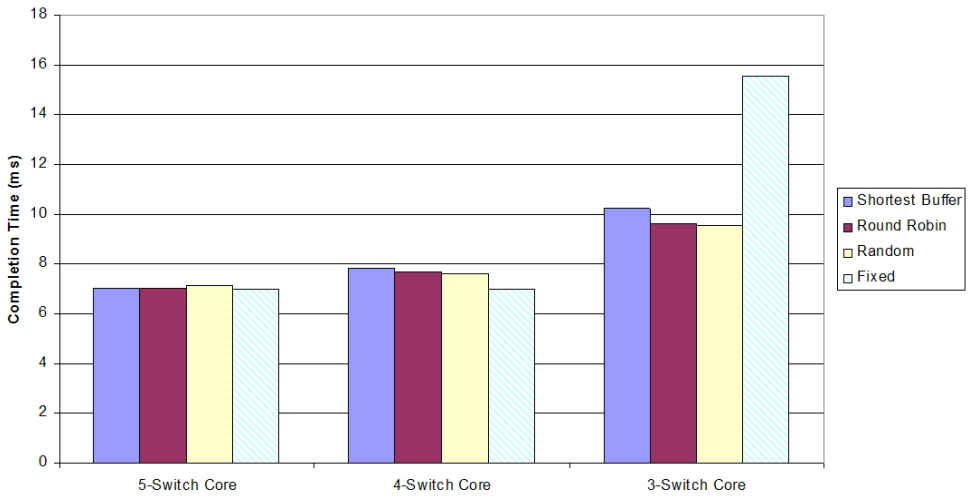


Figure 5-14. Corner turn results: adaptive versus fixed routing.

Since the adaptive routing was not effective for the corner turn benchmark (except under high-contention conditions) due to the ability to statically schedule the corner turn, we seek to examine the performance of adaptive routing under nondeterministic traffic conditions. Figure 5-15 shows the results of our random reads benchmark for each architecture and routing configuration. The benchmark is configured to have each

processing element send 1000 read requests of 256 bytes, each to an individually-selected random destination. Each processing element issues its requests sequentially, with a new request being issued after each subsequent request is filled with a RapidIO response. We choose total completion time for all read requests as the metric to evaluate overall performance. The random reads results show significant promise for the adaptive routing techniques. The shortest-buffer technique provided the best performance in all cases, with round-robin routing performing almost as well due to its extremely fair load balance over the course of the simulation. Random routing performed worse than the other adaptive routing techniques, but still was slightly better than the fixed-routed case. In this benchmark, trends between the routing methods did not change as we varied the number of active second-level switches; the only change is reduced performance across all routing methods as the number of second-level switches is reduced. Also note that the active extra-switch core improved performance of the nondeterministic traffic even when using static routing. This result is significantly different from our corner turn results where the extra-switch core was of limited benefit.

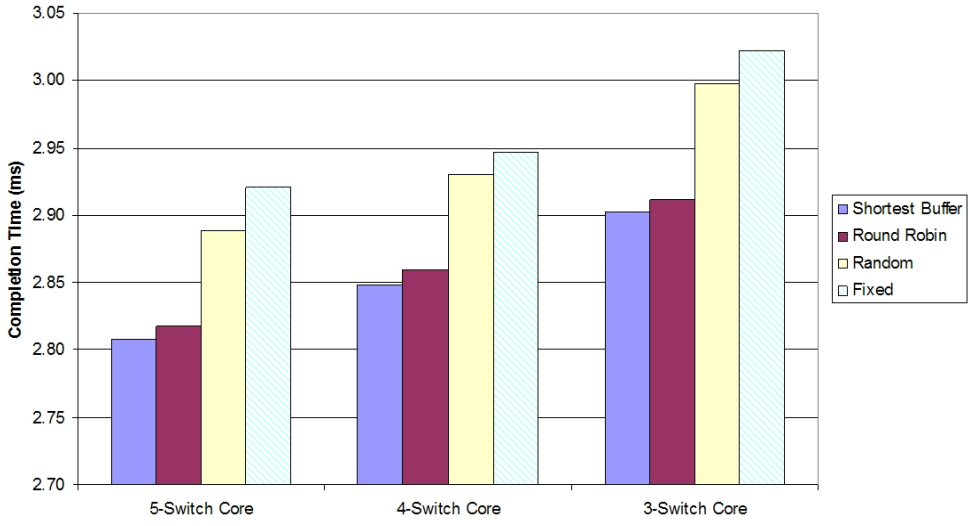


Figure 5-15. Random reads 256-byte results: adaptive versus fixed routing.

Figure 5-16 shows results of the random sends benchmark with 256-byte requests. Each processing element issues a request in parallel that is sent to a randomly-selected destination, and then all processors wait until all requests have been filled. We issue 1000 total requests per processor and measure the average latency for each request. This benchmark generates relatively light, synchronized traffic, and our five-switch and four-switch results show the fixed-routed case performs very well when compared to the adaptive methods. Low contention and efficient load balance cause the fixed-routed, shortest-buffer and round-robin tactics to perform relatively evenly in the five-switch and four-switch cases , while random routing has about 75 us higher packet latency on average. In the three-switch case, once again the network under-provisioning and higher contention lowers the efficiency of the fixed-routed case, and causes the shortest-buffer and round-robin routing to exhibit better performance. Also note that once again the five-switch core system performed best for each given routing tactic, including fixed routing.

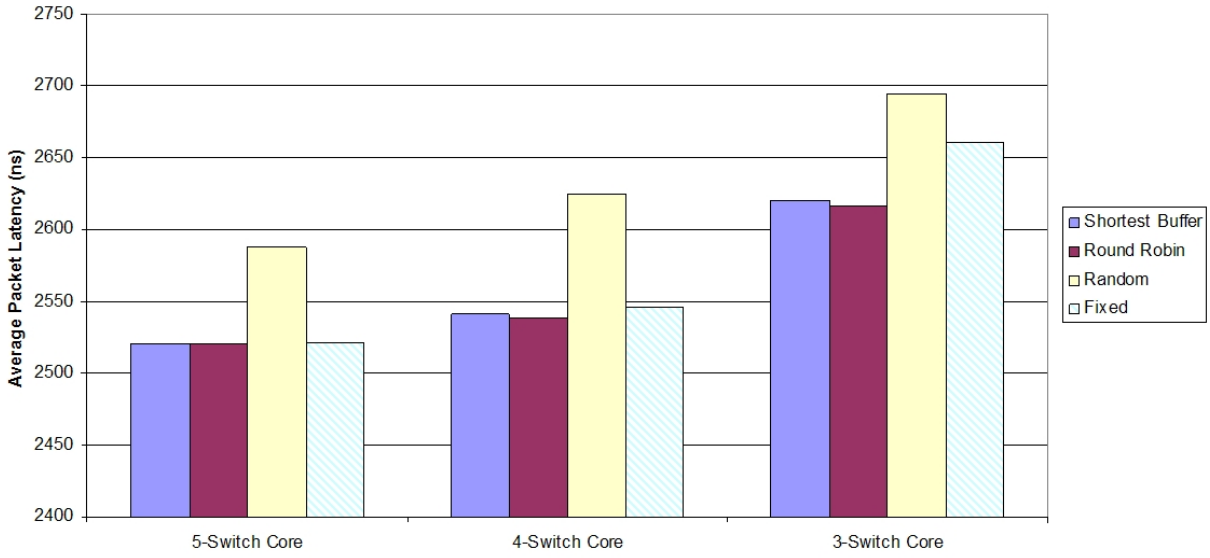


Figure 5-16. Random sends 256-byte results: adaptive versus fixed routing.

Our final test for the adaptively routed systems uses the same random sends benchmark, with 4096-byte RapidIO messages. This size is the maximum allowed by the RapidIO logical message-passing protocol, with each 4096-byte message consisting of 16 RapidIO packets, each with a 256-byte payload. Each processing element again issues 1000 messages in total, and we report the average latency of each individual packet in each message. Under this configuration, the benchmark has significantly higher network activity and creates much more potential for contention and frequent network saturation. Figure 5-17 shows the results of the 4096-byte random sends experiment. For the five-switch and four-switch cases, the shortest-buffer and round-robin routing tactics offer a small improvement over random and fixed routing. These are moderate-contention cases, but most contention present is not the fault of the network routing; instead, contention tends to occur when two or more sources randomly attempt to send to the same destination, and no routing tactic can overcome this event. However, in the case of extreme contention introduced by the under-provisioned three-switch core, much of the network contention may be caused when two or more 4096-byte messages for different destinations enter the same switch through different input ports, but the fixed routing dictates that they both leave a switch through the same output port. Any of the adaptive methods will spread the messages across all of the available output ports, while the fixed method is stuck using the single entry in the routing table. For this reason, even random routing outperforms fixed routing by a large margin in the three-switch case. As we have seen with the other random-traffic experiments, the five-switch network core was a benefit for systems using any of the routing tactics.

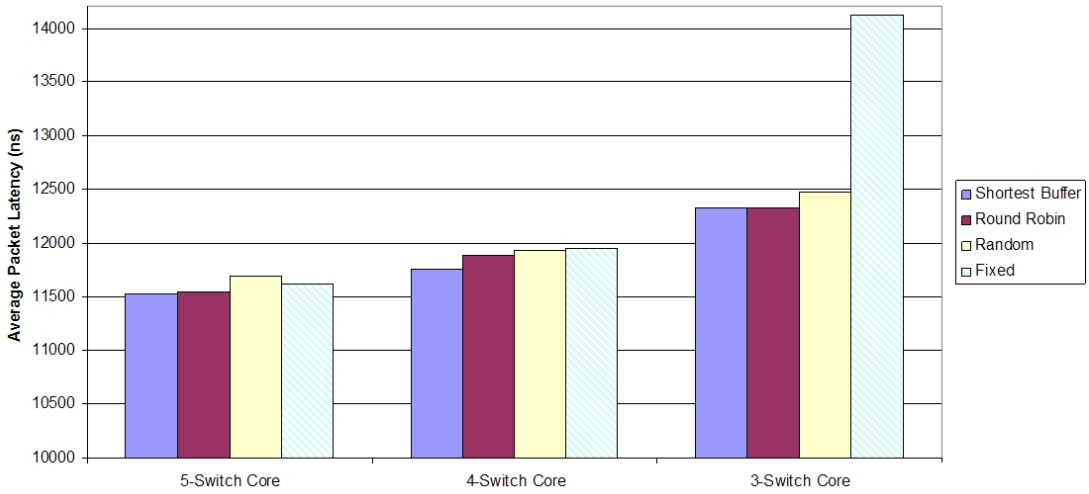


Figure 5-17. Random sends 4096-byte results: adaptive versus fixed routing.

In general, our experiments have shown that adaptive routing can be very effective in RapidIO networks in cases of high contention, where traffic cannot be statically scheduled. However, statically-schedulable applications with deterministic traffic over a high-bandwidth network backplane likely should be handled with fixed routing. Under-provisioned networks, such as those that are suffering degraded performance in the presence of faults, had the greatest benefit from adaptive routing, as expected. Of the adaptive routing tactics, the round-robin and shortest-buffer routing had the best overall performance, with a slight edge usually going to the shortest-buffer routing. However, routing based on status of a switch's output ports certainly will use more complex logic than simple round-robin routing, and the benefit gained from shortest-buffer routing was not extremely significant in most cases. One downfall of the shortest-buffer tactic is the fact that a switch may only make decisions based on the status of its local buffers, and switches in our RapidIO network have no knowledge of the status of other switches' buffers. Additionally, in a Clos network, adaptive routing decisions may only be made in the first stage if a shortest-hop path is to be taken through the network. With other

network topologies, the effectiveness of our adaptive routing tactics may vary, but we focus here on Clos networks due to their performance and fault tolerance characteristics. While distributed adaptive routing tactics could potentially introduce more effective decision-making into an intelligent adaptive routing approach, the level of complexity required would certainly not be within the spirit of the simple RapidIO protocol for embedded systems. The adaptive routing methods proposed in this research may all be implemented in a RapidIO-compliant system and require no special modifications to the data that is transferred in the network.

Quantitative System Evaluations

In this section, we seek to tie together our simulation results with additional analytical results derived from the basic characteristics of each system outlined earlier. For the evaluations in this section, we assume an application with deterministic communication patterns using fixed routing. We also assume that extra-switch core networks use four active second-level switches, with a fifth switch reserved as a cold spare. The fault model we consider is the switch fault, where a failure renders an entire switch unusable.

Since this is a quantitative evaluation, we try to use simple evaluation criteria that we feel reflect the needs of a typical embedded system for space-based payload processing. However, since there is really no such thing as a “typical” system, there will obviously be a level of subjectivity in the selection of evaluation factors. Therefore, we provide evaluations based on three different sets of weights, and, if desired, the reader may modify our weights or rating system in order to evaluate these system architectures for other specific needs. For this evaluation, we divide the evaluation factors into four main categories:

- Power consumption: Power is of paramount importance in nearly every embedded system, and is an important metric in our evaluations. We judge power largely based on number of active RapidIO switch ports in the entire system under no-fault conditions. A multiplexer device functions as a multiplexing LVDS repeater, which requires much less logic and uses much less power than a full RapidIO switch. Therefore, multiplexer ports are conservatively assumed to use 50% of the power of an active RapidIO switch port.

- Size/cost: For our purposes, we consider size and cost to be determined by the total number of network-port pins on all chips in the network fabric. In this manner, we try to treat serial versus parallel RapidIO considerations as fairly as possible, and also treat multiplexer devices as fairly as possible without introducing excessive complexity into a metric that clearly may be determined any number of ways. Note that by performing the evaluation this way, we allow that multiplexer devices need not be constructed exactly as shown in our models. For example, if an 8:4 parallel RapidIO multiplexer cannot be constructed due to size and pin-count issues, two 4:2 multiplexers could be used instead, and the pin count total would be the same for either configuration. For reference, we use 8-bit parallel RapidIO endpoints with 40 pins each, and 4-lane serial RapidIO endpoints with 16 pins each.

- Fault isolation: For this metric, we calculate the average number of switches that will be rerouted in the case of the first switch fault, assuming that any switch (first-level or second-level) in the network may fail with equal probability.

- Fault tolerance: For this category, we calculate the expected value of the number of switches that may fail in a given system before a performance loss occurs greater than 5% in our synchronized corner turn application. This application was selected since it highly stresses the network and represents a common operation in signal processing, which is a common use for space-based systems. This metric considers the loss of each switch in the network to be equally likely, with the loss of connectivity to a processing element's primary and redundant endpoints due to first-level switch loss considered to be a system failure. Note that for our purposes, multiplexer failure is not considered because in order to make a correct assessment of the probability and effect of device failure, the device would need to be constructed. It is an accurate statement to say that the probability of failure of any multiplexer device would be significantly lower than a full RapidIO switch. However, any real-life system design using multiplexer devices that may fail must consider the impact of device failure, especially if the devices represent a single point of failure that may disconnect processing elements from the system (as they do in all our multiplexer-based architectures except for the FTC network). While outside the scope of this work, most systems will have processor-level redundancy as well, and extra processor boards in a system will help provide an allowance for multiplexer faults.

Table 5-7 is provided for reference and contains a consolidation of the previously-reported basic characteristics of all architectures under study.

Table 5-7. Summary of fault-tolerant RapidIO architectures.

	Active Switches	Standby Switches	Total Switches	Active Ports per Switch	Total Switch Ports	Mux Count	Number Switches to Reroute-1	Number Switches to Reroute-2
Baseline Clos Network	12	12	24	8	192	0	0	0
Redundant First Stage Network	12	8	20	8	160	8 (8:4)	0	8
Redundant First Stage Network with Extra-switch Core	12	9	21	8	184	8 (10:5)	0	8
Redundant First Stage Network (Serial RIO)	12	8	20	8	192	0	4	8
Redundant First Stage Network with Extra-switch Core (Serial RIO)	12	9	21	8	224	0	4	8
RapidIO Fault-Tolerant Clos Network	12	3	15	8	140	8 (4:1)	5	8

Table 5-8 contains the data and results of our quantitative evaluation. Our evaluation uses three sets of weights, labeled Case 1, Case 2, and Case 3, that represent scenarios in which power, size/cost, and fault tolerance are the most important factors, respectively. In each case, two of the three factors are weighted 1.0 and the important factor is weighted 2.0. Fault isolation in each case is weighted 0.5 since it is based on a simple metric (routing tables to update) that was only a small focus of our investigation. Prior to weighting, the score for each category is normalized to 1.0, with the best-rated system(s) in each category having a score of 1.0 and other systems scored in proportion to the best-rated system(s). The fault isolation metric is a special case, because the baseline system has a score of 0, which cannot be used to normalize data. Therefore, we

normalize the data to the next-best value, and allow the baseline system to have a raw score of 0 (i.e., "perfect") in that category due to its superior fault isolation. In all cases, once the values have been normalized, lower scores are considered better in our evaluation.

While most calculations for the table entries are trivial (e.g., number of total pins), we briefly explain here the methodology used to calculate the fault tolerance metric. Equation 5-1 is used to calculate the expected number of switch failures, *F*, that may be tolerated before a loss of connectivity to any endpoint or a 5% drop in performance of our corner turn application (which always happens when the number of active core switches is less than four):

$$\overline{F} = \sum_{i=1}^{N} i \times P_i \times \left(1 - S_{(i-1)}\right) \tag{5-1}$$

The term S_i is defined using Equation 5-2 and represents the probability that a system failure occurred with any number of faults up to and including *n*:

$$S_n = \sum_{i=1}^{n} P_i \tag{5-2}$$

Where:

N = number of switches in the system

P_i = probability of a system failure after exactly *i* faults

Equation 5-1 is derived from the classical definition of an expected value, where the probability of system failure with a given number of faults is equal to the probability of system failure with exactly that number of faults (P_i), multiplied by the probability that the system has not previously failed with any smaller number of faults ($1\text{-}S_{(i-1)}$). In order to give the best scores to the architectures that can withstand the greatest expected

number of faults while still preserving the integrity of our results, the reciprocal of the expected number of faults is taken prior to normalization (reciprocal is not shown in Table 5-8). Note that inactive switches are considered to have a zero failure rate until activated, at which point we assume they are equally likely to fail as any other switch in the system.

As an example calculation, the redundant first stage network score for Case 1 is calculated as described here. To obtain the power score, the total number of active ports is counted (96) and added to ½ of the total number of active multiplexer ports (½ × 8 muxes × 8 ports per mux = 32). The value obtained (128) is normalized by dividing by the score of the architecture with the lowest (best) power score (four architectures tied at 96). The normalized value (1.33) is indicated in Table 5-8. The size/cost score is calculated in a similar fashion, by counting the total number of network pins (10240, active and inactive total) and normalizing (10240 is 3.33 times the number of pins of the "best" system in this category). The fault tolerance score is calculated as explained previously. The redundant first stage network can withstand an expected value of 2.67 switch faults before a 5% drop in performance occurs in our corner turn benchmark, which normalizes to a value of 1.67 when compared to the architectures that may withstand up to 3.95 faults (reciprocal must also be taken prior to normalization since lower scores are better). In the event of a fault, an average of 2.67 switches in the redundant first stage network will require their routing tables to be reconfigured. Since the baseline Clos network is exempt from this calculation for reasons already described, the redundant first stage network is then tied for the new "best" fault isolation (i.e., normalized score of 1.0) and all other networks are normalized against this network for

the fault isolation category. Finally, the redundant first stage network's weighted total score for Case 1 is calculated as: $2.0 \times 1.33 + 1.0 \times 3.33 + 1.0 \times 1.67 + 0.5 \times 1.00 = 8.17$.

Table 5-8. Evaluation of fault-tolerant RapidIO architectures.

Category	Power (number of active ports)		Size/Cost (number of total network pins)		Fault Tolerance (average switch faults tolerated)		Fault Isolation (average rerouted switches)		Total Scores (weighted sum of normalized values)		
Weights (Case 1/2/3)	2.0/1.0/1.0		1.0/2.0/1.0		1.0/1.0/2.0		0.5/0.5/0.5		Case 1 Power	Case 2 Size/ Cost	Case 3 FT
	Raw	Norm	Raw	Norm	Raw	Norm	Raw	Norm	Total	Total	Total
Baseline Clos Network	96	1.00	7680	2.50	2.00	1.98	0	N/A	6.48	7.98	7.45
Redundant First Stage Network	128	1.33	10240	3.33	2.37	1.67	2.67	1.00	8.17	10.17	8.51
Redundant First Stage Network with Extra-switch Core	128	1.33	12160	3.96	3.95	1.00	2.67	1.00	8.12	10.75	7.79
Redundant First Stage Network (Serial RIO)	96	1.00	3072	1.00	2.37	1.67	5.33	2.0	5.67	5.67	6.34
Redundant First Stage Network with Extra-switch Core (Serial RIO)	96	1.00	3584	1.17	3.95	1.00	5.33	2.0	5.17	5.33	5.17
RapidIO Fault-Tolerant Clos Network	96	1.00	7200	2.34	2.89	1.37	6	2.25	6.84	8.18	7.20

Our quantitative evaluation reveals several interesting aspects of the architectures under study. In terms of our power metric, the architectures are all very similar, with the multiplexer-based architectures suffering a penalty due to the additional ports required. Most networks will use the same amount of power under non-degraded, one-fault conditions as the no-fault conditions we have examined. However, it is also worth noting that the power consumption of the FTC network will increase under conditions where a

first-level switch fault has occurred, because the four 4:1 multiplexer devices on that side of the network will need to be activated in order to allow replacement of the faulty switch.

In terms of total size/cost based on the number of network pins in the system, there are large differences between the architectures. Reduced pin-count and lack of multiplexer devices in the serial RapidIO architectures give these architectures a significant edge over other the other architectures under study. The FTC architecture provides an interesting compromise, with less size/cost than the baseline and a greatly reduced total switch count at the expense of eight 4:1 multiplexer devices.

The baseline network clearly provides the best fault isolation, with two completely separate networks. The serial RapidIO and FTC architectures had the worst fault isolation, as most switch faults in these architectures require the reconfiguration of several switches in the system. Note that multiplexer devices must also be reconfigured in many fault cases, but this reconfiguration is not considered in our calculations since the configuration of these devices is trivial.

Our fault tolerance metric, along with our previous simulation results, indicates that the most fault-tolerant architectures to both first- and second-level switch faults were those with an extra-switch core. One interesting result can be observed by examining the expected number of faults to cause failure in the redundant first stage case (2.67, parallel or serial) as well as the equivalent systems with an extra-switch core (3.95). Note that the addition of a single core switch increases the expected number of tolerated faults by more than 1, due to the 1:1 switch redundancy that is also present in the first level of these networks working with the extra-switch core to provide an increased benefit.

While the serial networks faired the best in our evaluation under all three weighting schemes, the implementation of serial RapidIO technology for space systems presents a challenge that makes other alternatives worth considering for systems with an immediate path to flight. For Cases 1 and 2, the best parallel RapidIO system was the baseline due to its low power consumption and superior fault isolation. However, the FTC network was the best of the parallel alternatives for Case 3 because of its high level of fault tolerance when compared to the baseline and its reasonable size/cost due to unique use of multiplexer devices. Overall, the FTC network provides a unique compromise between the various alternatives shown, and this compromise is reflected in its total score in all three cases. This architecture may present a viable alternative to current fault-tolerant networks for space systems while the transition to serial networking technology for space takes place. In addition, future, larger RapidIO network configurations for space may make use of concepts from the FTC network as well as serial technology. The other two multiplexer-based architectures provide fault-tolerance increases over the baseline in terms of fault tolerance metric, but the extra size and cost potentially required by multiplexer devices as well as the potential fault of multiplexer devices may be unable to outweigh the switch-level fault tolerance benefits of these architectures (especially emphasized by their poor score in Case 2 where size/cost is most heavily weighted). It is important to note that our quantitative evaluation of these architectures simply presents one method of evaluating the networks under study, and these architectures and metrics should be tailored to meet the requirements of a specific mission when actually employing a fault-tolerant RapidIO network.

Summary

With the emergence of switched networks such as RapidIO for space-based payload processing systems, new architectural designs are needed in order to efficiently provide fault tolerance without resorting to duplication of the entire switched network fabric. For this phase of research, we have proposed a set of six architectures based on Clos network topologies that we adapted to RapidIO. In most cases, the architectures are unique from conventional packet-switched architectures in that they provide fault tolerance in different methods depending upon the network stage in question and the attributes of the particular network. We provided an overview of each architecture and presented a set of basic numerical characteristics for each one such as number of passive and active switches. We then presented a set of simulation benchmarks and experiments using these benchmarks to stress the performance of the architectures under no-fault, one-fault, and two-fault conditions. We also presented an analysis on adaptive routing and the effectiveness of extra-switch backplanes in our candidate networks. Finally, we concluded with a quantitative analysis using both simulative and analytical results in order to effectively compare the overall merit of each network.

Our matrix multiply simulation experiments found that the over-provisioned extra-switch core networks provided better performance under fault conditions than the other networks. However, the overall execution time difference between the networks was not very large for this benchmark, even when comparing no-fault to two-fault conditions. This benchmark demonstrated that even RapidIO networks with a modest level of fault tolerance can often continue to provide adequate performance in the presence of faults when an application does not require full bi-section bandwidth and non-blocking network connectivity.

Our corner turn experiments use much more network bandwidth than the matrix multiply and require large all-to-all data transfers. The trends noticed in the matrix multiply experiments continued for the corner turn benchmark, but these trends were greatly amplified, especially for the 32-processor corner turn case. Our synchronization experiments with the corner turn benchmark showed that careful synchronization of communication in RapidIO networks becomes very important as contention is introduced into the network, especially in the presence of faults. Under no-fault conditions, the two corner turn benchmarks had similar performance, but the presence of faults introduces contention and gives a large edge to the synchronized version of the corner turn benchmark. Our final corner turn experiment examined the performance of the extra-switch core networks when actively using the extra switch rather than employing it as a cold spare. The performance of the five-switch core case actually did not improve, as the corner turn communication was already efficiently scheduled and synchronized for the four-switch case. For cases where traffic may be statically scheduled to use the baseline four core switches without contention, this experiment suggests power may be saved by leaving the extra switch inactive unless faults arise.

In order to determine the effectiveness of our network architectures under random traffic patterns that cannot be statically scheduled, we integrated adaptive routing capabilities into our RapidIO switch models and used several benchmarks with traffic generated to random destinations. In all random-traffic cases, the extra-switch core networks provided a significant performance boost when the extra switch was used in a powered mode. Adaptive routing also improved performance in most cases, especially in under-provisioned conditions with high levels of network contention. Of the adaptive

routing methods studied, the best-performing tactics were round-robin routing and shortest-buffer routing. Round-robin routing, however, should require considerably less logic to implement and provided performance very comparable with the shortest-buffer tactic, which requires information about the status of a switch's buffers to make decisions about packet routing.

To help draw overall conclusions on each network architecture under study, we performed a unique quantitative analysis combining our analytical metrics and simulation results. This analysis indicated that serial RapidIO-based architectures based on Clos networks with an extra-switch core should provide an excellent balance between performance, fault tolerance, size, power, and cost. The Fault-Tolerant Clos (FTC) architecture also provides a very good alternative for parallel RapidIO-based architectures. The FTC architecture requires the use of multiplexer components, but these components do not represent a single point of system failure in this architecture. Other multiplexer-based architectures provided favorable increases in fault tolerance over our baseline with a reduced number of total switches, but the price paid in system size and cost due to the multiplexer devices may be prohibitive. Similar to the work performed in the first phase of this research, we have chosen to focus strictly on RapidIO, but the insight gained from this study may also apply to other embedded packet-switched networks such as InfiniBand or the Advanced Switching Interconnect.

CHAPTER 6
RAPIDIO SYSTEM-LEVEL CASE STUDIES (PHASE 3)

In this chapter, we present three detailed system-level case studies with an expanded simulation environment that includes detailed models of the processing and memory subsystems in addition to RapidIO network models. This phase of research expands on the two network-centric previous phases by studying architecture tradeoffs at the processor-level as well as system-wide considerations for a complete space system architecture. Using validated simulation based on an actual RapidIO hardware testbed, we develop key insight into the optimal configuration of these high-performance systems and suggest additional architecture modifications to further improve processor and network performance.

Introduction

While the previous two phases of this research have focused on a network-centric view of the applications and systems under study, our virtual-prototyping approach allows much more detailed modeling of an entire system, including processors, memory devices, and external sensor data interfaces. Our third phase of research will take a deeper look at some realistic, complete RapidIO system configurations running simulated applications. While RapidIO provides very high data rates and the ability to transfer the data directly into the memory of a remote processing node, the memory interface is often shared between one or more host devices that need access to the incoming RapidIO data for processing. This interface may often present a bottleneck in a real-time system, and it is important to understand all relevant tradeoffs regarding this interface. For example,

burst length, access latency, and interface clock rate are all important factors that can drastically affect the performance of a real-time system. In addition, memory bottlenecks are compounded by the high-speed processing possible with FPGA or ASIC hardware, where data elements may often be processed as fast as they may be obtained from memory, greatly increasing the difficulty in performing memory access hiding via overlap of I/O and processing (i.e., double buffering).

This phase of research focuses on the integration of RapidIO networks with high-performance FPGA-based processing engines for GMTI and SAR space systems. We use discrete-event simulation to perform several case studies based on our RapidIO testbed architecture, with a focus on gaining insight into the optimal configuration of the memory interface and reconfigurable hardware resources for our SBR application kernels and additional case study requirements. Our testbed was constructed with guidance from our sponsors at Honeywell Inc., and is representative of a typical first-generation RapidIO-based architecture that would be present in an actual flight system. The RapidIO testbed is a critical part of this work, and allows us to study actual application and network performance. However, it is cost prohibitive and inefficient to use actual hardware (even reconfigurable hardware) to study all the relevant tradeoffs of interest for this research. Instead, we use results from our hardware testbed to validate and calibrate our custom-built RapidIO simulation models, and then upwardly scale system sizes and adjust parameters within our models to gain a breadth and depth of insight into architectural tradeoffs not possible using only the testbed.

Testbed Architecture Overview

Our RapidIO testbed was constructed using two Xilinx development boards, Virtex-II Pro FPGAs, and RapidIO IP cores, as well as a collection of custom-designed

circuits to enhance the capabilities of the hardware and mimic an actual RapidIO-based flight system. The actual testbed development and algorithm implementation is outside the scope of this work, but background is provided in order to give detail on the architecture being studied through modeling and simulation. The Virtex-II Pro FPGAs used in the testbed are XC2VP40-FF1152-6 devices, providing sufficient resources for debug, experimentation, and expansion with over 40,000 logic cells. Each FPGA is enhanced with 128 MB of off-chip DDR SDRAM operating at 125 MHz with a 32-bit data bus, providing a theoretical main memory throughput of 8 Gbps at each node. Furthermore, each FPGA also contains an 8-bit parallel RapidIO endpoint (logical I/O logical layer) running at 250 MHz that provides a theoretical network throughout of 4 Gbps per node (8 Gbps full-duplex). The internal data path width is 64 bits, except inside the processing engines themselves where the data path width is 32 bits.

The internal design of each node is a custom architecture that seeks to maximize memory efficiency and data throughput by leveraging the parallelism available in specialized hardware components such as FPGAs and ASICs. This architecture has been designed with guidance from our sponsors at Honeywell Inc. and is representative of a typical first-generation RapidIO-based system for space. Each node consists of an external memory controller, a network interface controller, an on-chip memory (OCM) controller for internal dual-port SRAM memories, and a variety of processing engines. These processing engines can include both embedded PowerPCs as well as custom hardware co-processors designed using the reconfigurable resources of the FPGA. Our processing engines are all custom hardware designs, with a PowerPC used to control

DMA transfers. Figure 6-1 illustrates the internal node architecture, and shows both the control and data paths between the various sections of the node.

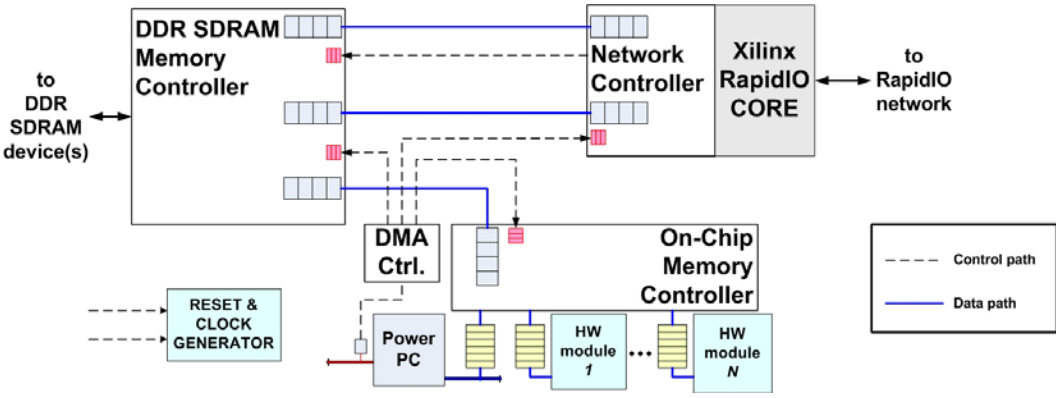


Figure 6-1. Conceptual single-node testbed architecture.

The baseline design is SDRAM-centric, meaning that all memory transfers involve reading or writing from/to external SDRAM. By examining the data paths indicated in Figure 6-1, it can be seen that the node architecture consists of 3 main sections: (1) external memory, (2) network interface, and (3) OCM. These main sections all pass data to one-another through parallel FIFOs. Processing engines are loaded with data from SDRAM, and write processed results back. Network transfers also always occur between the external memory controller and the network interface. Multiple command queues are interfaced to the actual external memory controller core, with arbitrated service for concurrent requests to avoid starvation. The purpose of having multiple command queues, with sufficient control logic, is to provide a high degree of parallelism for internal transfers. Incoming transfer requests over the network can be serviced in parallel with locally-initiated memory transfers, both local and remote. Furthermore, isolation across FIFOs provides flexible clock domains, allowing each peripheral to operate at its ideal frequency without depending on other peripherals.

Baseline System Architecture Overview

The baseline system architecture used for our case studies is shown in Figure 6-2. This is similar to the architecture shown in Figure 3-3, except it also contains a system controller. This controller connects to an additional port on one of the backplane switches as described in Chapter 3. Each of the seven (maximum) total processor boards in the system contains four FPGA-based processors and an eight-port RapidIO switch, and each processor has the architectural features described in the previous section, including an interface to its own bank of off-chip DDR SDRAM. Refer to Chapter 3 for further details on the modeling environment and system architecture.

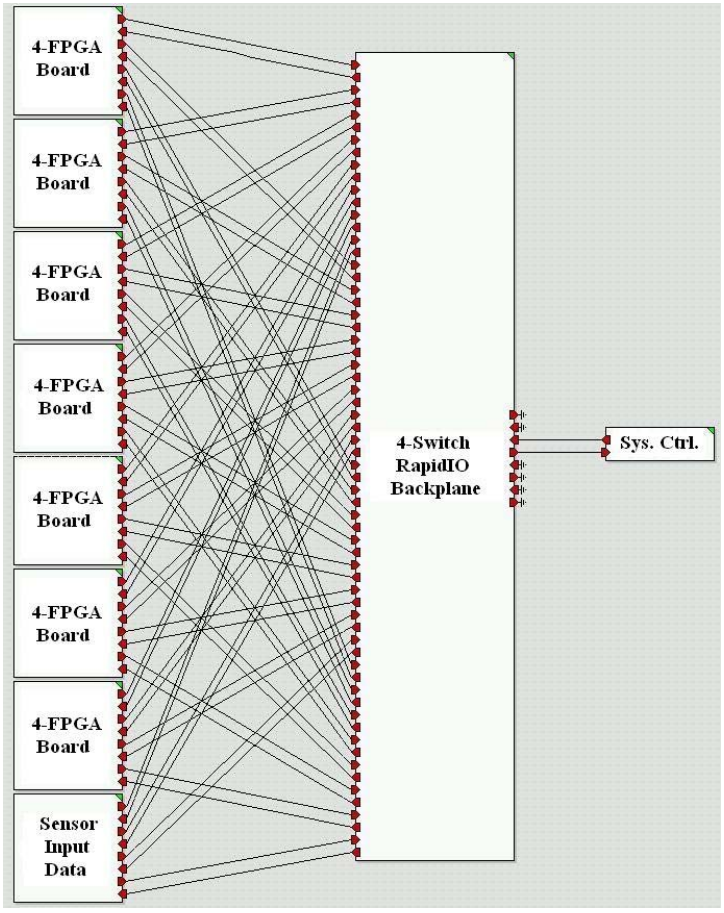


Figure 6-2. Baseline GMTI case study architecture.

Simulation Case Studies

This section presents the results of three simulative case studies on space-based payload processing. Our first case study revolves around four key GMTI kernels in our GMTI application described in Chapter 4. We then present research related to design of architectures to support timely delivery of latency-sensitive data in RapidIO systems running applications with high throughput requirements such as GMTI. Finally, we present an analysis of architectural approaches and tradeoffs for global-memory-based FFT processing for SAR on RapidIO-based systems.

Case Study 1: GMTI Processing

This research focuses on four key processing kernels that compose the backbone of the GMTI algorithm studied in Chapter 4—Pulse Compression, Doppler Processing, Beamforming, and Constant False-Alarm Rate detection (CFAR). These kernels were selected because they occupy more than 99.999% of GMTI's total single-processor execution time based on the processors assumed in Chapter 4, and we have working FPGA-based implementations of each of these processing kernels to serve as validation for our simulation models and provide us with realistic inputs to our application model scripts. Pulse Compression and Doppler Processing primarily consist of FFTs and complex vector multiplies, Beamforming uses a matrix multiply, and CFAR performs magnituding and a sliding window average and threshold comparison function.

Since this work is based on FPGAs, we can easily measure computation time for a given data size using our testbed and input this data as a scripted model parameter. We have found that once data is loaded onto the FPGA, processing times are completely deterministic. The deterministic processing times are a result of each processing engine having direct access to dedicated banks of SRAM which it may read or write in a single

cycle with no contention possible. However, getting data into the FPGA's SRAM from local SDRAM or over RapidIO is not deterministic because of other requestors (such as the RapidIO endpoint) that may be attempting to access the SDRAM, and this is exactly the type of behavior we seek to model.

For this case study, a complete new data cube arrives from the spacecraft's sensors every 256 ms and is evenly distributed to the system's processors over RapidIO after initial pre-processing is performed (outside the scope of our case study). We use the straightforward data parallel GMTI partitioning that is also the focus of Chapter 4 due to its ability to produce CPI results with the lowest latencies. Each input data cube contains 256 pulses, 6 beams, and a number of range cells that is varied depending on our experiment parameters. For our application, each data element consists of two 16-bit integers, one real and one imaginary, and original data cube sizes range from 200-400 MB. However, after Pulse Compression (first stage), the size of the data cube shrinks by a factor of 0.6 as 40% of the range cells are discarded, and after Beamforming (third stage), the data cube shrinks by a factor of 0.5 as 3 beams are formed from 6 input. At the conclusion of CFAR detection, 1 KB of detection results is passed from each processing element to the system controller. For our baseline configuration, each FPGA-based processing element in the system has a single Pulse Compression core, a single Doppler Processing core, a single Beamforming core, and a single CFAR core. Since the algorithm processes in stages, only one type of processing core is active at any one time.

Our first experiment is designed to examine the performance of the GMTI algorithm when the throughput of the FPGA and RapidIO interface to the DDR SDRAM is varied. All data arriving over RapidIO is first written to SDRAM, then read by the

FPGA for processing, and then written back to SDRAM after processing, so this interface is a very important focal point of this work. Our current testbed DDR SDRAM only supports a maximum of 8 Gbps, which is the minimum needed to provide full throughput to 250 MHz, DDR, 8-bit parallel RapidIO, so we use simulation to explore the effects of increasing the DDR SDRAM and interface throughput. We ran simulations with varied SDRAM throughputs and data cube sizes (by varying number of range cells), using all 28 FPGA processing elements possible with our baseline GMTI system, and results are shown in Figure 6-3. We report the average latency for full delivery and computation of each CPI, measured from the time the sensor input begins to transfer to the processors to the time the system controller receives the last bit of CFAR detection results. We generally run 8 CPIs for each individual simulation; however, we discard the first and last CPIs of each run from the average so that results are not skewed by startup or wrap-up time in cases where there is overlap (double buffering) from one data cube to the next.

Figure 6-3 shows that the most significant performance increase occurs in the jump between 8 and 16 Gbps in all cases, although the jump was less significant in the 32k-ranges case for reasons we will explain. Between 8 and 16 Gbps, the memory subsystem goes from under-provisioned to adequately provisioned for GMTI's network and processing patterns. These results show that while 8 Gbps may be sufficient memory bandwidth to sustain full-duplex RapidIO communication associated with GMTI's corner turns, it leaves virtually no margin for any SDRAM access by the FPGA.

In the case of the 32k-ranges simulation, the data cube is small enough that the delivery and processing of the data in its entirety takes less than 256 ms. However, in the case of the other two sizes with 8 Gbps memory bandwidth, the delivery and processing

takes longer than 256 ms, making some amount of network-level double buffering necessary. During network-level double buffering, processors continue to work on CPI *N* while they receive the data for CPI *N+1*. The system can continue to meet real-time deadlines in this case as long as memory is large enough to accommodate both data cubes and as long as delivery and processing of the data can each individually complete in under 256 ms. Network-level double buffering is a key factor in utilization of the SDRAM interface, because it enables the situation where the FPGA may be accessing chunks of data from the SDRAM from CPI *N* while RapidIO is attempting to write CPI *N+1* to the SDRAM. Without double buffering, there is no overlap of communication and computation in our GMTI algorithm. The lack of overlap decreases the penalty imposed by the 8 Gbps interface since this interface is still adequate for RapidIO and the FPGA as long as they are not accessing the memory at the same time.

Another factor of note shown in Figure 6-3 is the trend that the gap between performance of the 64k- and 48k-ranges cubes is larger than the gap between the 32k- and 48k-ranges cubes. With all other factors being equal, the performance of the GMTI application should scale completely linearly with the data cube size. However, the extensive use of double buffering for the 64k-ranges case causes additional contention for the memory subsystem and creates a performance penalty in all cases.

The reasons for the performance penalty associated with double buffering can be further understood by examining the utilization of the SDRAM memory interface over the course of the simulation. Figures 6-4 shows SDRAM utilization over time for 8 CPIs for the 32k-ranges case (no double buffering needed) and 64k ranges case (extensive double buffering needed) with the 16 Gbps SDRAM interface. Areas of low SDRAM

activity are areas where only data distribution only is taking place. The “medium” peaks in the figure between 40-60% are due to processing or corner turn activity, while the 100% utilization peaks are due to local data redistribution, which is completely memory bound. Note that most of the first and last CPIs of the 64k-ranges case are setup and wrap-up during which no double buffering occurs, so focus on the middle six CPIs of the 64k-ranges chart to see the effects of double buffering. During periods of overlap between data distribution and processing activity (i.e., the medium peaks), utilization of the SDRAM interface of the 64k-ranges case is about 4% higher than the utilization of the SDRAM interface of the 32k-ranges case. This incoming data especially slows memory performance during local data redistribution (100% peaks), and slows network performance during corner turns due to network contention.

Another trend displayed in Figure 6-3 is the leveling off of the performance improvements as the SDRAM interface reaches 24 and 32 Gbps. This is largely due to Amdahl’s Law, as the RapidIO network itself is unaffected by improvements in the DDR SDRAM interface once it has reached the point where it can supply full-duplex communication to each endpoint. Since the RapidIO network is capped at 8 Gbps of full-duplex throughput, improvements to the memory subsystem do not significantly affect data distribution or corner turns. Figure 6-5 shows the execution time breakdown of the execution of the 32k ranges, 16 Gbps memory interface CPI time. This case was chosen because it does not require double buffering and leads to easier visualization and understanding of execution time components. Almost 60% of the execution time of each CPI is consumed by receiving data, leading to the diminishing returns with higher memory subsystem performance seen in Figure 6-3.

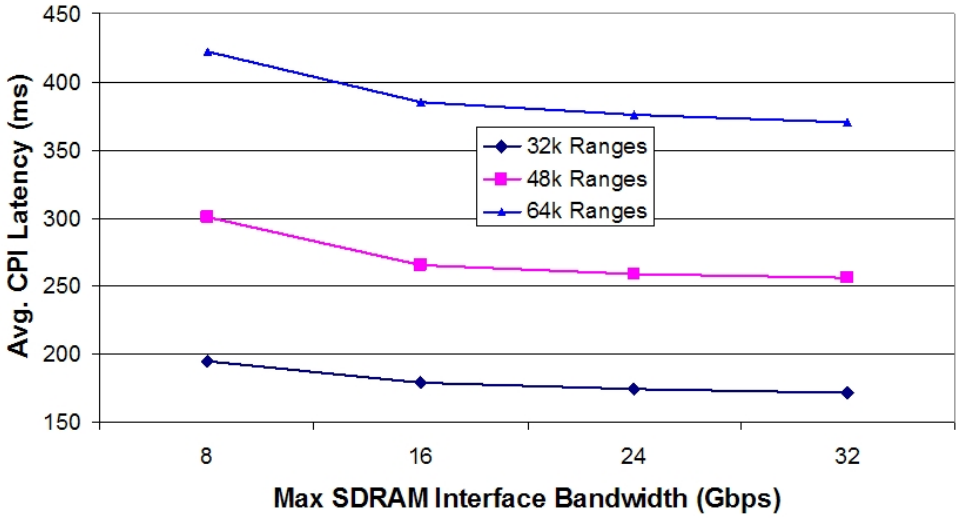
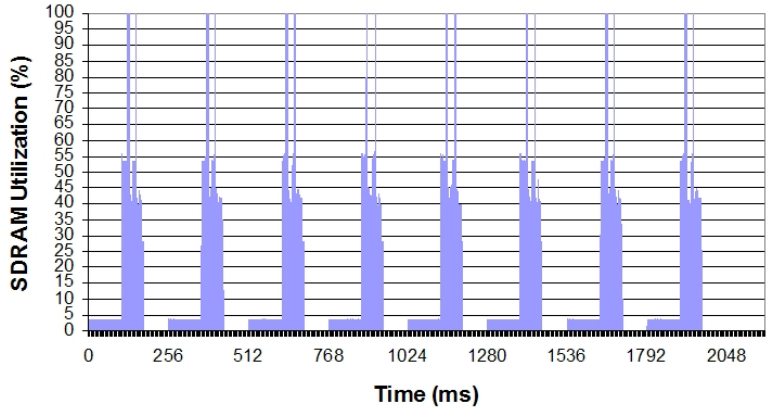
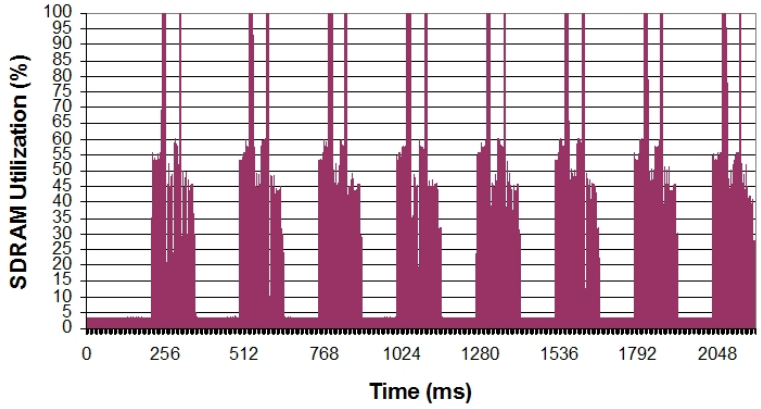


Figure 6-3. GMTI performance: 256 pulses, 6 beams, 1 engine per task per FPGA.



A



B

Figure 6-4. SDRAM utilization versus time: 256 pulses, 6 beams, 1 engine per task per FPGA. A) 32k-ranges case. B) 64k-ranges case.

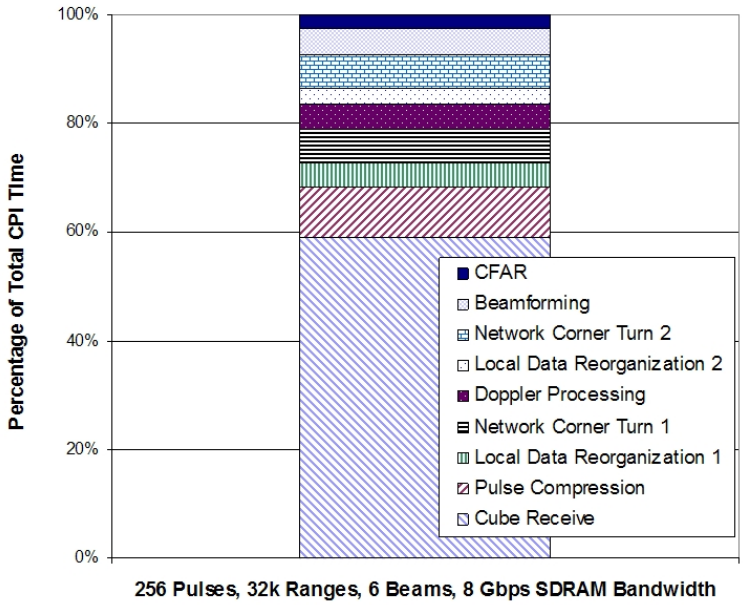


Figure 6-5. GMTI CPI execution time components.

Our second experiment varies system sizes to gain insight into the performance of the GMTI algorithm over a smaller number of processors. Current radiation-hardened FPGAs with sizes capable of supporting the GMTI application have not yet reached the level of fault tolerance of radiation-hardened ASIC technology currently available. Therefore, a true FPGA-based flight solution would likely need to use some form of Triple-Modular Redundancy (TMR) [66] or other advanced software-based means of detection and recovery from transient faults [67]. A TMR-based solution will require three times the number of FPGA resources for a given application, which can severely limit the practical scalability of the system as redundant, active resources must be included. It is important to note that our implementation of the GMTI algorithm does not take advantage of the reconfigurable capability of the FPGAs, so there is nothing stopping the application from being mapped to a rad-hard ASIC at the cost of future flexibility if a change in the application or algorithm is desired. Other factors that may limit the number of processors in a real-world space system might include power, size, or

weight concerns, or the desire to save a certain number of processors as cold spares in the event of permanent hardware faults. Due to the wide variety of parameters that may limit practical system size, we show performance and speedup from 8 to 28 nodes for the 32k-ranges data cube in Figure 6-6. The data cube size was selected because the relatively small size allows a smaller number of processors to meet the real-time deadline by using double buffering.

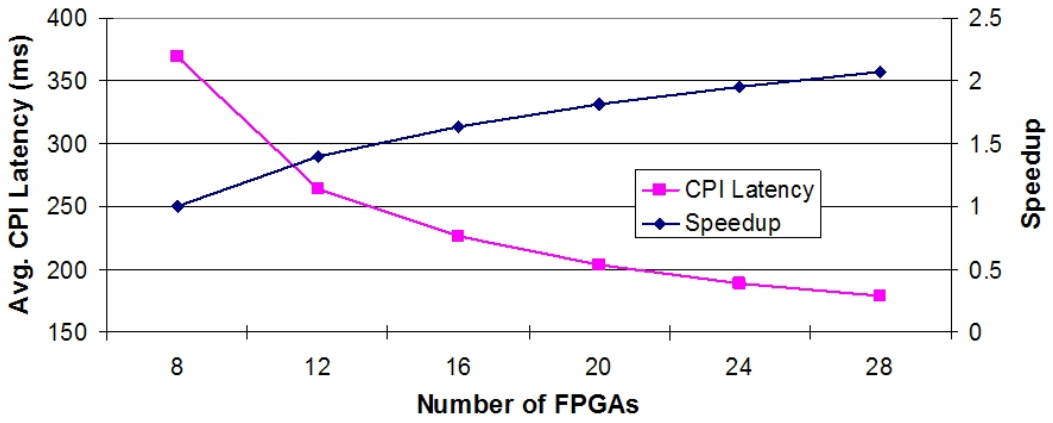


Figure 6-6. GMTI scalability: 256 pulses, 32k ranges, 6 beams, 1 engine per task per FPGA.

As system sizes in Figure 6-6 increase towards our baseline maximum of 28 processors, CPI latency significantly decreases, but speedup is only about 2.07 when moving from 8 to 28 processors. While corner turns and processing stand to reap large benefits from a greater number of processors, the initial data distribution is still limited by the fact that we have four RapidIO ports coming from the sensor inputs. Since this data distribution takes a large portion of the total execution time, once again Amdahl's Law limits the application speedup unless more data inputs are added to the system.

Our final GMTI experiment varies the number of processing engines dedicated to each GMTI task on each FPGA. While our testbed is bound to one processing engine per

task per FPGA by the limitations of our Xilinx Virtex-II Pro 2VP40 FPGAs (19,392 slices, 192 BlockRAMs, and 192 multipliers) [68], we can use simulation to study the advantages that larger FPGAs such as higher-end Virtex-II Pro or Virtex-4 devices [69] can provide. These larger FPGAs can have more than four times the resources of our 2VP40, so we seek to examine system performance with one to four processing engines per task per FPGA. Figure 6-7 shows the performance of the 64k-ranges data cube processing with the SDRAM interface speed and number of processing engines varied.

The results in Figure 6-7 show performance becoming increasingly memory-bound as the number of processing engines increases. As engines are added, GMTI processing becomes even more memory-intensive than shown in our previous results. One major reason behind this trend is a factor we have not yet discussed—processor-level double buffering. While network-level double buffering involves the processing of one cube while receiving the next one, processor-level double buffering involves breaking the data cube up into chunks for local processing. A chunk is read from SDRAM to on-chip SRAM for processing, and the FPGA immediately sets up a DMA so that it has another chunk of data ready once it is finished with the current chunk. In this manner, the latency of local memory access can be largely hidden by overlapping with computation, but memory bandwidth required during processing phases roughly doubles. Our processing algorithms all can process approximately one 32-bit element of data each clock cycle (minus startup latency penalties), with a clock rate of 125 MHz for the processing engines. This rate leads to a “processing bandwidth” of 4 Gbps. In order to perform double buffering at the processor level, as is done in all of our GMTI experiments, each

engine must have access to at least twice the 4 Gbps processing bandwidth (i.e., 8 Gbps) because it must read data from SDRAM as well as write results back to SDRAM.

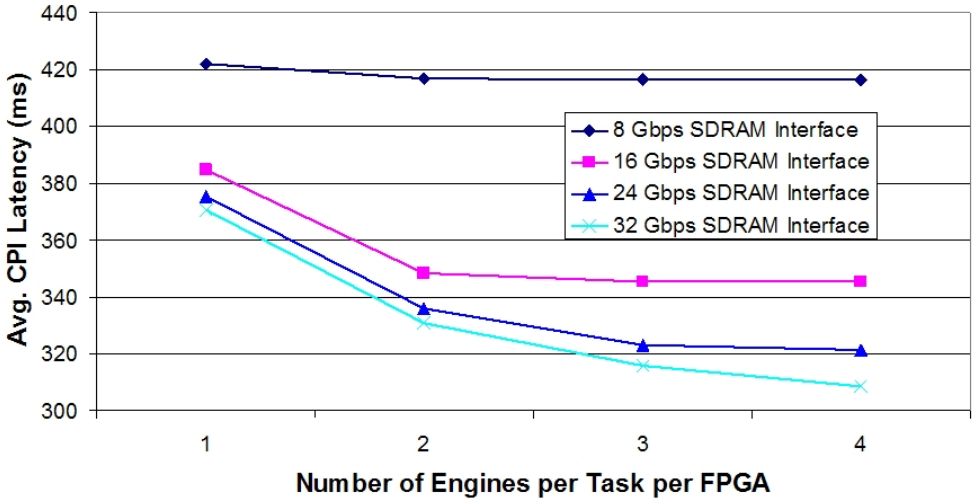


Figure 6-7. Varied number of GMTI processing engines: 256 pulses, 64k ranges, 6 beams.

As additional engines are added, the bandwidth required compounds, and the incoming double-buffered RapidIO traffic continues to require memory bandwidth as well. Figure 6-7 shows a trend that each successive increment in bandwidth beyond 16 Gbps allows the effective addition of one additional engine per task before performance begins to level off (i.e., three engines provide an improvement at 24 Gbps and four engines provide an improvement at 32 Gbps). This trend is due to the fact that each additional engine requires approximately 8 Gbps of dedicated bandwidth. However, we again see diminishing returns in each performance jump due to the additional dependence on the RapidIO network throughput for data delivery and corner turns. Note that the 8 Gbps interface is under-provisioned in all cases and its performance suffers accordingly.

Case Study 2: Latency-Sensitive RapidIO Data Delivery

The focus of our second case study deals with the latency of individual critical RapidIO packets in a system with heavy additional network traffic. Our baseline system

and application for this case study are the same as our GMTI case study. However, rather than focusing on the parameters and performance of the GMTI application, we instead focus on the delivery time of regularly generated, latency-sensitive, “critical packets” that are passed from the system controller to the processing nodes. In a real-world system, these critical packets would contain auxiliary data such as health or status information, or global system timing data. These packets are generated much less frequently than the primary application dataflow, and hence do not occupy a significant amount of network throughput. For our case study, a single critical packet is generated at the system controller and sent to each node every 150 us, with a spacing of 5 us between critical packets sent to different destinations. The critical packets are assigned RapidIO priority level 1, while all other application traffic is assigned RapidIO priority level 0. The use of a higher priority allows the critical packets to pass other lower-priority packets in the fabric if both are buffered at the same switch. However, the RapidIO specification does not specifically provide a mechanism for a higher-priority packet to interrupt the transmission of a lower-priority packet that is already in progress. Because RapidIO is a packet-switched network, many types of contention scenarios can arise once a packet has left its source, making latency generally very difficult to predict in a system that is not tightly controlled.

The primary goal of this study is to determine and attempt to reduce the average latency of critical traffic as well as the upper bound of this latency. In addition, we would like to measure and reduce the variability of latency of critical traffic (i.e., “jitter”). In a perfect scenario for latency-sensitive traffic, it would be ideal for RapidIO to behave more like a circuit-switched network with very predictable latency of control data. In

many previous systems, a separate network for control and data would be provided, but this work seeks to determine the feasibility of using a single RapidIO network as both a high-throughput data network and a control network capable of providing predictable, low latency.

The techniques we seek to evaluate in simulation are described below:

- Dedicated paths: This option provides a dedicated backplane switch that is connected to the system controller as well as each "board-level" switch (i.e., the switch shown in Figure 3-1), in an attempt to partially mimic the functionality of a dedicated path in a circuit-switched network. In our system, a critical packet must always pass through two switches to travel from the system controller to an end node, and this configuration ensures that the packet will not encounter any contention until it reaches the board-level switch. The configuration requires the addition of one extra switch port on each board-level switch (nine ports total, one for the dedicated path) as well as one backplane switch (for a total of five backplane switches rather than four).

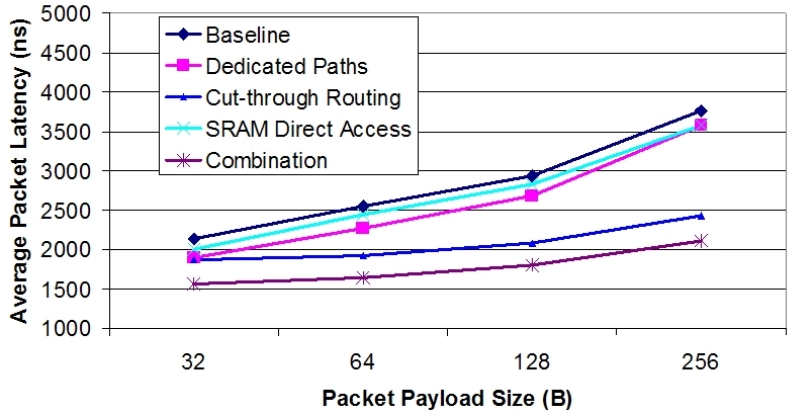
- Cut-through routing: The RapidIO specification supports both store-and-forward and cut-through routing. Our switch models also allow selectable routing types, although the baseline switch assumed for this work is store-and-forward and based on first-generation RapidIO switches for space.

- SRAM direct access: As previously described, all incoming RapidIO packets in our system must first be written to SDRAM before the processing element can access this data by bringing it into SRAM. This behavior is typical of most networking designs (e.g., an Ethernet NIC does not have direct access to a host processor's cache), however, it may not be the only option in a tightly-coupled embedded system. This alternative provides a direct path to on-chip SRAM for incoming RapidIO data and bypasses the DDR SDRAM interface that non-critical traffic must traverse twice (once to be written to memory by the RapidIO network controller, and once to be read out by the processor).

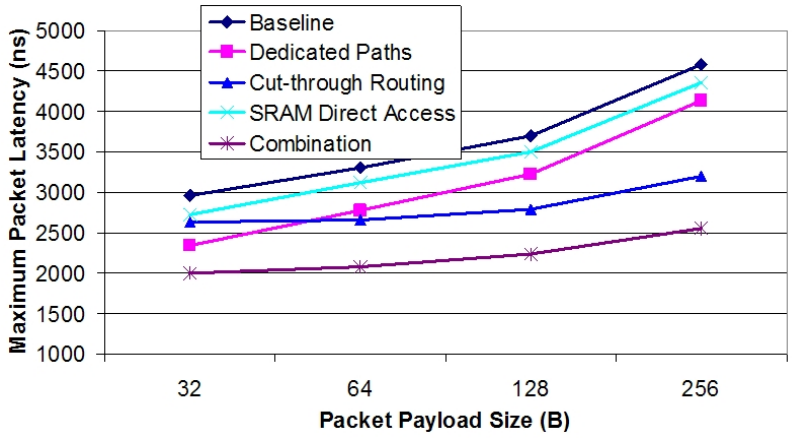
- Combination: All methods described above are used in parallel in the same system.

This case study varies the size of the payload of the critical packets between 32 and 256 bytes, and explores the impact of the techniques described that attempt to reduce latency and jitter. We measure latency from the time the packet is generated at the system controller to the time it enters on-chip SRAM for access by the destination processor. For cases without direct access from RapidIO to SRAM, the processor always

immediately attempts to access the packet once it is available in DDR SDRAM. Because we have found that GMTI's corner turns create the greatest potential for high network traffic and variable latency, we create simulation runs that generate constant all-to-all corner turn traffic over 144 ms of simulated time. Over this time period, about 1000 critical packets are sent to each processing element. It is important to note that each non-critical packet generated as part of the corner turn has a maximum-sized 256-byte payload in all cases. Figure 6-8 shows both the average and maximum latency observed over the runtime of our simulations for each configuration.



A



B

Figure 6-8. Latency of critical packets. A) Average latency. B) Maximum latency.

As expected, the baseline configuration clearly had the highest maximum and average latency for all packet sizes. The difference between the average latency and the

maximum latency can be attributed to several factors. Most importantly, if a critical packet arrives at a switch but another non-critical packet has already arrived and is destined for the same switch output port, the critical packet will incur an additional delay of up to approximately one packet time (536 ns for a 256-byte non-critical packet payload with a 12-byte RapidIO header/trailer). This delay will be dependent on how far ahead of the critical packet the contending packet arrived at the switch (further ahead implies smaller delay). Note, however, that if multiple non-critical packets are buffered at the switch and destined for the same output port, the critical packet may pass all of the other packets that have not begun to transmit, bounding the delay to approximately one packet time per switch. On average, because traffic is being constantly transmitted by the corner turns, a critical packet will delay approximately 0.5 packet times per switch (most noticeable for packets with 256-byte payloads where overhead is less significant). For all cases, critical packets must travel through two switches to reach their destination. The next major difference between average and maximum latencies is due to the variability of the DDR SDRAM interface. While this interface is prioritized, if a critical packet arrives at an endpoint while another SDRAM transfer is in progress (such as an outgoing RapidIO packet being read from memory), the critical packet will have to wait for access to the bus. This contribution is far less significant than switch delay, however, because we limit memory access to 256 bytes at a time (one RapidIO packet payload), and the memory interface is running at 16 Gbps for these experiments (about 150 ns to copy one 256-byte packet payload). Other unavoidable fixed delays include a single store-and-forward delay for each packet between the RapidIO physical and logical layers, and latencies for the various processing, memory, and network modules in the system.

The dedicated paths optimization provides a moderate improvement to the average packet latency because it completely eliminates the potential for contention at the first switch a critical packet must traverse. We have already estimated the average delay at this switch to be about 0.5 packet times for large packets, and the improvement seen in Figure 6-8 (A) is consistent with this estimation. However, in the worst-case, where a critical packet arrives just after a non-critical packet destined for the same port, the dedicated paths optimization provides an even more substantial improvement because it eliminates this possibility from ever occurring at the first switch in a critical packet's path. One minor shortcoming of the dedicated paths optimization is that the improvement is relatively constant across all packet sizes and does not grow along with packet size. This trend occurs because the optimization does not improve critical-packet transfer time. Rather, dedicated paths reduces the time a critical packet may spend waiting for other packets, which are all of the maximum size (268 bytes including header/trailer) in this case study. Also, note that in the absence of contention dedicated paths provide no latency improvement.

Cut-through routing provided the most substantial improvements to both the average and maximum packet latencies in our case studies. Cut-through routing allows a packet to immediately flow from a switch's input to output buffer as long as there is no contention for the switch output port. This savings is applied to each and every packet, and avoids substantial packet copy delays which increase as packet sizes increases, leading to an increased savings with increased packet size. Improvements are relatively consistent for a given packet size in both the average- and maximum-latency cases. However, one other interesting point can be drawn from the cut-through results. It is

stated in literature that virtual cut-through routing (used in our switches) degrades to store-and-forward routing in the presence of contention [70]. While this is generally true, it is important to note that in our case, the switch performance essentially degrades to the performance of an unloaded store-and-forward switch because a critical packet may wait at most one packet time instead of two. We collected minimum packet latencies to verify this trend and found that the minimum latency for the baseline system was 3180 ns with 256-byte packet payloads, occurring with no contention at either switch. This result is nearly identical to the cut-through maximum latency of 3192 ns, occurring when there is heavy contention at both switches.

Direct access to SRAM from RapidIO provided small but consistent improvements to both the maximum and minimum packet latencies. This optimization does, however, provide at least some savings with each and every packet by avoiding a copy to SDRAM before the data is read into SRAM, and this savings increases with increasing packet size. This savings will further increase during contention scenarios, when a packet is forced to wait for access to the SDRAM interface. The savings could also grow if the local processing element was attempting to access SDRAM during the corner turns. While this activity does not occur in our GMTI application, other applications could certainly present even more stressful access patterns to the memory, further increasing the importance of scheduling computation and memory access such that packet latencies are not adversely affected and increasing the value of direct access to SRAM from RapidIO. One other point of note in the average case is that the improvements from dedicated paths and direct SRAM access appear to converge at the maximum packet payload size of 256 bytes. This convergence occurs because dedicated paths do not provide an increased

benefit as packet size grows, while direct SRAM access does provide greater benefits for larger packets.

Finally, the combination of all three methods proposed was extremely effective, and we found that the improvements from each method could be added to estimate the final result (to within about 100 ns in most cases—note that Figure 6-8 shows measured simulated results with all three optimizations active). The reason for the effectiveness of this technique is that each method improves non-overlapping aspects of the end-to-end system latency and none of the methods limit the performance of the others. Cut-through routing improves packet transfer time at each switch a packet traverses, while dedicated paths cut down on potential time that may be spent waiting for a packet to access a switch output port. SRAM direct access provides improvements within the end-node processor and network interface architecture. The most drastic benefits from the combination are noticed in the maximum-latency case, where the increased benefit of dedicated paths adds to the already-large cut-through benefit and as well as the small improvement of SRAM direct access.

While these three methods have shown significant improvements in the average and maximum latencies of critical traffic, we have not yet discussed the variability of individual packet latencies for each case. Figure 6-9 shows the standard deviation of packet latencies for each configuration using 256-byte packet payloads. Note that while cut-through routing provided the biggest improvements in average and maximum latency, it did not significantly help the standard deviation of packet latencies (i.e., jitter). SRAM direct access did slightly improve jitter by removing a variable delay that may be spent waiting for access to the SDRAM interface, but the largest improvements to jitter were

obtained by the dedicated paths system and the combination system. The dedicated paths system completely removes a point of contention and variability from the system by allowing critical packets to always pass freely through the first switch on the way to their destination, leading to the significant reduction in jitter.

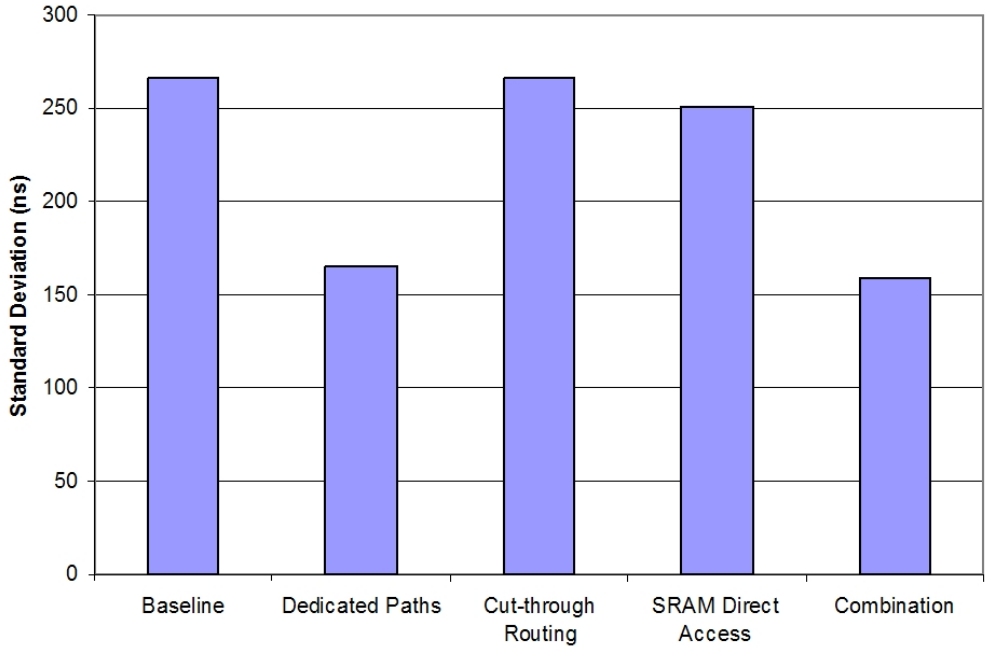


Figure 6-9. Standard deviation of packet latencies with 256-byte packet payloads.

Overall, this case study has shown that cut-through routing was our best method of improving packet latencies, but did not help with the variability of the latencies. Dedicated paths, however, did improve variability, while also providing a significant overall improvement to latencies, especially in the worst-case scenario. Direct access to SRAM also provided some reduction of critical-packet latency and jitter in our system and may lead to even larger improvements in cases where the processor is accessing SDRAM concurrently with heavy network traffic. The main advantage of all three of these tactics is that they can be overlapped and applied to the same system, with drastic improvements that are nearly as large as the sum of the individual improvements.

Case Study 3: Global-Memory-Based Fast Fourier Transform (FFT) Processing

Our third and final case study examines tradeoffs related to global-memory-based signal processing. Chunk-based processing out of a global memory is necessary for applications such as SAR that have memory requirements too large to be partitioned across the local memories of the processing engines in a space-based embedded system. While a full-fledged SAR implementation is outside the scope of this research, the FFT is a frequently-occurring processing kernel in any SAR system and in many other signal-processing applications, so we use global-memory-based FFTs as a case study for investigation of issues related to global-memory-based SAR processing in space systems. In addition, communication and memory access patterns remain the same no matter which type of chunk-based processing is being performed for SAR, so we can gain considerable insight in general SAR tradeoffs by focusing on the global-memory-based FFT for SAR processing. For this case study, we assume the images processed are 16k pulses by 16k ranges, with each data element being a 32-bit complex integer. Our simulated application performs one 16k-point FFT for each pulse, for a total of 16k FFTs. Therefore, each FFT operates on 64 KB of data. These FFTs are performed by the 16 total processors in the system in a distributed fashion (i.e., coarse-grained parallelism).

We have shown in the first phase of this research that smaller chunk sizes lead to the most efficient utilization of network resources and minimum contention, and that synchronization of communication and global-memory access is extremely important to achieve maximum algorithm performance. Therefore, we choose the absolute minimum chunk size possible as our starting point for experiments (64 KB in most cases due to the size of the data a 16k-point FFT operates on), and we continue to use the synchronized reads implementation as our baseline for global-memory-based processing. The baseline

method and our other proposed alternatives are explained below (abbreviations in parenthesis are used as reference in figures):

- Baseline: Each processor waits for a “read token” to arrive, at which point it is allowed to read a chunk of data from one of four global-memory ports. Once the processor has completed the RapidIO read accesses, the node passes the token to another processor. At any given time, there are four tokens circulating in the system, one for each global-memory port. Once the entire chunk of data has been read into SDRAM, the processor must read the chunk into on-chip SRAM, perform the FFT, write the chunk back to SDRAM, and then write the chunk to global memory over RapidIO.

- Processor-level double buffering (PDB): Processor-level double buffering is used to overlap processor SDRAM access with FFT processing. Processors follow the baseline method of synchronized access to global memory, but once data arrives in SDRAM, it is processed locally in a double-buffered manner. That is, once a processor has read its data from global memory, it reads a chunk into SRAM to being processing, and immediately begins reading another chunk while processing is occurring. We vary the chunk size that is read via RapidIO, while processor-level double buffering always occurs in 64 KB chunks (the size of data for the 16k-point FFT) for maximum efficiency. Because at least two local chunks are required for double buffering, the minimum chunk size that may be read from global memory is 128 KB if processor-level double buffering is performed.

- Network-level double buffering (NDB): Network-level double buffering is used to overlap RapidIO network traffic with FFT processing. The technique used is very similar to the one examined in the first phase of this work, and does not use synchronization to schedule access to global memory because we previously found that synchronization limits the effectiveness of network-level double buffering. The processors read a chunk of data from global memory, but immediately after receiving the chunk (before beginning processing) the processors begin a transaction to read another chunk from global memory.

- Processor-level and network-level double buffering (PNDB): This technique combines the previous two techniques to allow overlap between network data transmission and processing, as well as overlap between processing and memory access. No synchronization is used because of the network-level double buffering.

- SRAM direct (SRAM): This technique bypasses the use of the DDR SDRAM altogether, using the RapidIO direct to SRAM architecture feature described in the previous case study on RapidIO latency issues. Processing elements are able to read data from the global memory using RapidIO reads, and this data is transferred directly into the on-chip SRAM of the processing element rather than first being stored to DDR SDRAM. RapidIO writes back to global memory also occur straight from SRAM, and do not need to first be written back to DDR SDRAM. Network synchronization uses a read token and is identical to the baseline.

- SRAM direct with network-level double buffering (SRAM+NDB): This configuration is a combination of the SRAM direct technique and the network-level double-buffering technique. This method provides overlap between communication and computation, and removes any need for access to DDR SDRAM. Again, no synchronization is used in order to achieve maximum performance benefit from the network-level double buffering.

System parameters used for this case study are the same as those used for the previous two case studies, including a 16 Gbps DDR SDRAM interface. We varied network-level chunk sizes in most cases from 64 KB to 1 MB for each technique described, and results for the distributed, global-memory-based FFT are shown in Figure 6-10. Note that 64 KB is not a valid chunk size for the two techniques using processor-level double buffering because at least twice this amount of data must be read from global memory to accommodate double buffering of the 16k-point FFTs.

Figure 6-10 shows the results of our experiments. While these results may at first appear very erratic, there are several important trends visible in the figure. First, note the general trend is that the smallest possible chunk size performs the best in all cases except the processor-level double-buffering case, and also note that usually the smallest chunk size is the best by a significant margin. Smaller chunk sizes tend to reduce network contention and also allow processors to receive the global-memory read token quicker (in synchronized cases) and begin performing useful work.

For chunk sizes 128 KB and up in the baseline case, an interesting trend is exhibited that differs slightly from the results obtained in our first phase of research. While the smallest chunk size is clear and away the best, performance then begins to slightly improve again as chunk sizes reach 256 KB and up. This result occurs here because of the additional time that is now spent in the baseline due to explicit simulation of local memory access. This effectively increases the time that is spent processing in

each chunk, reducing network contention and effectively “excessively serializing” the algorithm, much like the results seen in the first phase of this research (Figure 4-9). Similar but slightly more exaggerated trends are exhibited by the processor-level double-buffering (PDB) case, because larger chunk sizes also allow more overlap between computation and memory access. In general, however, our results indicate that processor-level double buffering is not nearly as effective for global-memory-based processing as it is for our data parallel GMTI data cube distribution. The reason for this trend is that we are dealing with much smaller chunks for this case study, which means that the overhead of waiting for the data to arrive and then reading the first chunk of local processing data can no longer be offset by hundreds of iterations of double-buffered processing. Instead, we are performing only a maximum of 20 iterations (for the 1 MB case) of local processing, and as few as 2 iterations in the case with the smallest processor-level double-buffered chunk size of 128 KB. The baseline network-level double-buffered (NDB) case again performed well for small chunk sizes, with erratic results as chunk sizes increased due to contention for network resources. The configuration combining processor-level and network-level double buffering (P+NDB) in most cases did not achieve significantly better performance than the network-level-only double-buffered case, except for the 1 MB chunk size where the benefits of processor-level double buffering become apparent. The SRAM direct (SRAM) case provided substantial improvements over the baseline configuration, and unlike the other optimizations studied, it does not require any additional memory resources (either extra SRAM or SDRAM). In fact, it requires no off-chip SDRAM at all, leading to a potentially much simpler and less expensive system design with lower power

consumption. Finally, the SRAM direct case with network-level double-buffering (SRAM+NDB) case provided excellent performance, roughly equivalent performance to the standard network-level double-buffering case with small chunk sizes. This method also does not require any off-chip SDRAM access, sharing a key benefit of the standard SRAM direct case while providing near-optimum performance.

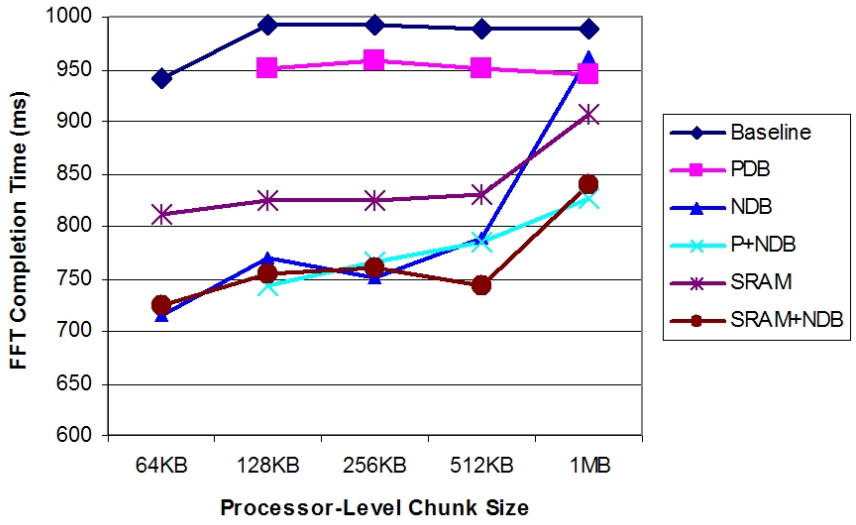


Figure 6-10. Distributed global-memory-based FFT completion time versus chunk size.

In terms of general conclusions, the results in Figure 6-10 imply that large chunk sizes should not be considered for our case study application. While processor-level double buffering to hide memory access time initially would initially seem to be a promising approach, the relatively small chunk sizes used here in comparison to our GMTI algorithm render the processor-level double buffering largely ineffective (even with large chunk sizes) when compared to other methods using small chunk sizes.

For easy comparison of the methods examined, Figure 6-11 shows the maximum performance achieved for each method along with the chunk size at which the performance was obtained. In all cases except the processor-level double-buffered case,

this chunk size was the smallest one possible for the given approach. Our results indicate that for systems with very tight memory constraints, the direct SRAM based implementation (with no double buffering) provides an excellent performance advantage over the baseline with the added advantage of no off-chip memory required, resulting in power and cost savings for a real-world system. The tradeoff in this case is additional complexity required in the RapidIO endpoint design to permit access from RapidIO to SRAM. However, this complexity is not expected to be prohibitive. Also, in a production-level system, the endpoint design could be much less complex as a whole if the use of off-chip memory can be avoided. For near-maximum performance at the cost of doubled on-chip SRAM, our results indicate that the direct SRAM with network-level double buffering is a very promising option (the minor performance penalty from the standard network-level double-buffered case is less than 1% and is likely due to slightly increased network contention). Finally, for our existing RapidIO endpoint design the network-level double-buffered case provides ideal performance at the cost of requiring off-chip SDRAM.

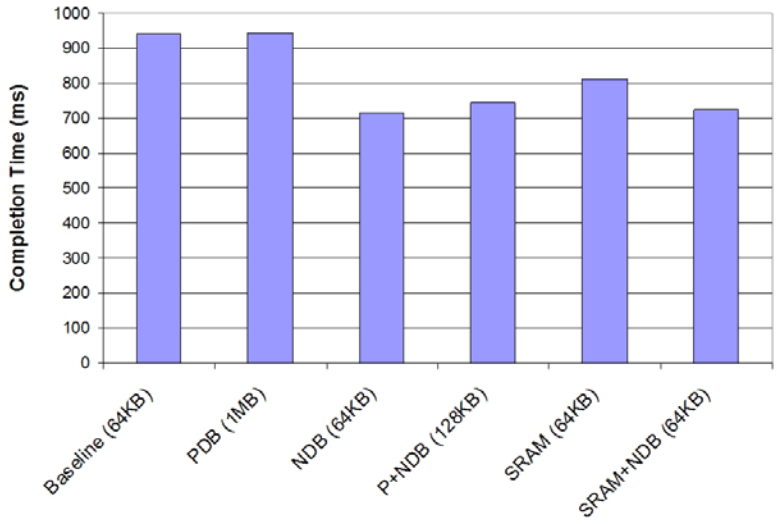


Figure 6-11. Distributed global-memory-based FFT minimum completion times.

Summary

In this chapter, we have employed a virtual-prototyping approach to perform three case studies related to GMTI and SAR processing in space systems with FPGA-based processing elements and a RapidIO interconnect. Our architecture used for these case studies is representative of a typical first-generation RapidIO system for space and has been designed with the guidance of our sponsors at Honeywell, Inc. An actual RapidIO hardware testbed serves as the basis for the case studies, and has been used in extensive calibration of the simulation models to ensure they accurately model a real-world system. However, one of the advantages of our simulation-based approach is that we can adjust system parameters to model hardware that we do not have available in the testbed, in terms of performance, scale, and features (e.g., faster memory interface, more or larger FPGAs, or direct access to SRAM from RapidIO).

Our first case study examined the performance of a series of GMTI processing kernels running on our simulated system. We found that network-level double buffering allowed processing of larger data cubes than would be possible without double buffering, but it did cause a noticeable relative performance hit for the larger cube sizes due to increased contention for the network and DDR SDRAM interface. While our hardware testbed only possesses a memory interface capable of at most 8 Gbps, our simulation experiments found that this interface is sufficient for cases that do not use double buffering, because the processing and communication phases of GMTI are inherently separated in terms of a single CPI. Our results showed that performance can be substantially improved by using larger FPGAs and additional engines per task on each FPGA, but only if memory bandwidth is increased to a sufficient point to sustain the extra memory traffic. In general, however, our experiments showed diminishing returns

as memory bandwidth and system sizes increased due largely to Amdahl's Law and the fact that data distribution time is dependent primarily on RapidIO network throughput and occupies a major percentage of total application execution time.

Our second case study examined the latency of critical RapidIO packets transmitted from the system controller to the processing nodes during execution of the GMTI application. This case study showed that employing cut-through routing for the RapidIO switches was the best method of improving average and maximum packet latency, but it did not improve the jitter of critical packets. We found that for our prioritized RapidIO traffic, the latency of cut-through packets even under worst-case contention was only about as high as the best-case latency for store-and-forward packets. The use of dedicated paths for critical traffic moderately improved average latency and significantly improved maximum latency, while also improving jitter by eliminating one point of packet contention in the system. Direct SRAM access via RapidIO provided minor improvements for latency of packets in our application but, for other applications with different traffic patterns or systems with lower memory bandwidth, the optimization could provide more substantial improvements. One very important result of this case study comes from the fact that the optimizations studied affect different portions of the end-to-end system latency, and hence can be combined with additive improvements to average latency, maximum latency, and jitter.

The third and final case study we performed for this research used a distributed FFT application to gain insight into tradeoffs in performing applications such as SAR that require a global memory in a space-based embedded system to due very high dataset sizes. Our results showed that small chunk sizes generally yielded the best performance,

and processor-level double buffering did not provide significant benefit due to the relatively small amount of local data (KB versus MB) available compared to the GMTI application. Network-level double buffering, however, did provide significant benefits for the FFT application by allowing overlap between data reception and processing. We also examined the performance of the application with direct access enabled from a processor's RapidIO interface to on-chip SRAM. This optimization provided significant performance benefits over the baseline while allowing the cost-effective and power-reducing design of processor-level architectures with only small, fast on-chip SRAM rather than also requiring off-chip SDRAM. The last configuration we examined used network-level double buffering in addition to direct RapidIO access to SRAM. This configuration exhibited performance nearly identical to the standard network-level double-buffered configuration, with the same power and cost benefits of the other SRAM-based design.

It is important to note that all of the fault-tolerant designs explored in the second phase of this work can be mapped to our case study systems with small modifications while achieving a relatively even level of performance. In the case of "low-end" current systems with a smaller number of logical processors and no serial RapidIO available, the baseline redundant network design will excel in terms of power and size/cost considerations due to its lack of multiplexers in the network design. However, the RapidIO Fault-Tolerant Clos (FTC) network also provides a slightly higher level of fault-tolerance with comparable power, size, and cost for current systems. For future systems with serial RapidIO capability and a larger number of processing elements, the serial RapidIO redundant first stage network with extra switch core is the clear-cut ideal

solution for our case study systems. In all cases, the performance results obtained in this phase are independent of the network-level fault tolerance solution ultimately chosen.

CHAPTER 7
CONCLUSIONS

In this study we presented background information on space-based payload processing and embedded networks. All relevant standards for data networks for modern space systems were examined, and the selection of the RapidIO network as the focus of this work was justified. Additional background was presented on the RapidIO embedded systems interconnect in order to provide in-depth information that applies to the research that is examined in this study. An overview of our RapidIO modeling library and framework was presented, and this framework has expanded throughout the course of this research to allow us to model additional components contained in a complete space system such as FPGAs and memory devices.

The throughput and latency limitations associated with beaming large amounts of raw data to the ground have led to a push over the last several decades to perform more processing onboard a spacecraft and limit ground transmission to result data only. Although radiation-hardened components lag several technology generations behind their purely COTS counterparts, recent advances in the capabilities of radiation-hardened processing and network components are making compute- and network-intensive, real-time applications feasible for space systems. While data networks for space are not a new concept, previous systems have been largely bus-based architectures with contention issues and limited throughput and scalability. RapidIO is an open standard, packet-switched interconnect that was designed from the ground-up with a focus on embedded systems, and hence is potentially an ideal network for space applications.

The first phase of this work provides background and related research on the GMTI and SAR real-time SBR algorithms, and explores the mapping of these algorithms onto RapidIO-based space systems from a network-centric perspective. While there is a wide variety of literature on GMTI and SAR, most existing literature focuses on processing-related aspects rather than the complex communication patterns that greatly stress any communications network. This phase examines GMTI and SAR first individually and then jointly in order to determine the tradeoffs in creating a system architecture capable of alternating between the two algorithms. The RapidIO network is an important focus of this work, and important insight is gained and lessons learned for implementing real-time applications over a packet-switched embedded network. In particular, we found that the performance of the GMTI algorithm was heavily throughput-constrained, and it was necessary to intelligently plan and synchronization communication patterns to reduce network contention as much as possible. We found that timely data delivery was extremely important for the real-time algorithm to function properly, but we also found that even systems with modest RapidIO interconnects were capable of supporting GMTI for most data cube sizes under study when the data source was provided with additional RapidIO ports to meet data delivery requirements. We also found that network-level double buffering was of significant benefit for GMTI and helped to allow for scalability in terms of algorithm parameters and system size. Our SAR experiments found that SAR was less processor- and network-intensive than GMTI, but its higher memory requirements forced the use of a global-based-approach. Our results showed that generally writing/reading small chunks to/from the global memory provided the best performance, especially when network access was effectively synchronized. A chunk-

based double-buffered approach also showed performance improvements for SAR, although not to the degree that double buffering of entire data cubes could double the real-time deadline of GMTI and allow processing of larger cube sizes. Our results found that RapidIO switch parameters, method of flow control, and use of logical-layer responses did not significantly affect the performance of GMTI or SAR. This research also examined considerations for systems capable of both GMTI and SAR, and found that a global-memory-based approach for GMTI could lead to more balanced memory requirements for each application in exchange for a significant penalty in GMTI performance. As an additional deliverable from this phase of work, the RapidIO model library has been made publicly available to allow others to employ and modify our models for their specific system design needs.

The second phase of this research explores RapidIO networks for payload processing from a fault tolerance perspective. Background and related research are provided on existing methods for providing fault tolerance in embedded systems as well as HPC environments. In addition, we provide RapidIO-specific fault tolerance background and also present related research on fault models that have been used in literature. While previous space systems have used the approach of employing an isolated, redundant network, this phase of work uses the switched nature of RapidIO to more efficiently create fault-tolerant systems. Ideas from literature are adapted and applied to RapidIO-based systems, and a novel approach combining simulation with analytical metrics is used to evaluate proposed system designs. Our results indicated that serial RapidIO-based architectures based on Clos networks with an extra-switch core provide an excellent balance between performance, fault tolerance, size, power, and cost.

The Fault-Tolerant Clos (FTC) architecture also provided a good alternative for parallel RapidIO-based architectures. In general, we found that employing an active or inactive extra backplane switch beyond the required bi-section bandwidth of a system was an effective method of providing fault tolerance. In cases with random traffic, the extra-switch core networks also provided a significant performance boost when the extra switch was used in an active mode and adaptive routing techniques were used. Of the adaptive routing techniques studied, round-robin routing provided near-optimal performance at very additional little cost in added logic in a production system.

The third phase of this research draws on results gathered in the previous two phases to create a set of real-world case studies with an actual RapidIO system architecture and FPGA-based GMTI and SAR processing kernels. Our results from this phase focus on the integration of the RapidIO network into a complete system including processing components, memory components, and an external data interface to the spacecraft. While our initial study of GMTI and SAR was network-centric, the work in this phase studies the performance of all important system components with an emphasis on how they integrate with the RapidIO network. The results from our first case study emphasized the importance of the DDR SDRAM interface shared by our processing elements and RapidIO network, as it was found to be the limiting factor in performance and scalability in several cases, especially when double buffering was used to allow increased dataset sizes. However, by avoiding double-buffered communication and ensuring minimal overlap between network and processor memory accesses, we found that real-time GMTI could still be performed with the 8 Gbps memory interface present in our hardware testbed. Our results also showed that performance can be significantly

improved by using larger FPGAs with additional engines per task, but only if memory bandwidth is increased to sustain the extra memory traffic created by the additional engines. In our second case study we studied three proposed methods of improving or bounding the latency and jitter of critical RapidIO packets in a system with heavy data traffic such as GMTI corner turns. We found that employing cut-through routing for the RapidIO switches was the best method of improving average and maximum latency, but did not improve the jitter of critical packets. However, the use of dedicated paths for critical packets did significantly improve jitter. Most importantly, we found that all of the optimizations studied can be employed in parallel with additive improvements, since none of the optimizations reduce the benefits of any of the others. Our third case study used a distributed FFT application to gain insight into tradeoffs in performing applications such as SAR that require a large global memory bank in a space-based embedded system. We found that processor-level double buffering was ineffective due to the small chunk sizes available, but network-level double buffering provided substantial performance benefits by overlapping communication and computation. We also found that providing direct access to on-chip SRAM from RapidIO was a promising method of potentially reducing system complexity, cost, and power consumption while also providing major performance improvements over our baseline SDRAM-based configuration. In addition, our results showed that combining network-level double buffering with direct SRAM access provided near-optimum performance with the same benefits as the standard direct SRAM-based design.

This work makes several scholarly contributions to the literature on networks for payload processing as well as GMTI and SAR. It is the first research study to perform a

thorough investigation of the issues related to building RapidIO-based systems for space. Through the development and use of our unique RapidIO modeling environment, we provide insight on several important points to consider when building RapidIO-based systems for real-time, network- and processor-intensive algorithms such as GMTI and SAR. Our research is also the first study to examine the issues related to creating fault-tolerant, real-time packet switched systems for space using methods other than simple network-level redundancy. This research uses a novel combination of analytical metrics and simulation results to draw conclusions into key tradeoffs in developing RapidIO-based architectures for space systems in terms of performance, fault-tolerance, size, power, and cost. This work is also the first research study to fully explore the considerations between interfacing a RapidIO network with high-performance processing elements through a shared memory subsystem. We explore these tradeoffs using a unique set of case studies built around realistic RapidIO-based systems and application kernels. Furthermore, our research is the first study to explore adaptive routing and latency-reduction techniques for RapidIO networks. While this research has focused on a specific set of system architectures and applications running over RapidIO-based networks, in many cases the results obtained throughout this study may also apply to similar systems or networks running other applications with similar communication patterns.

There are several avenues for future research that have been created as the result of this work. One potential area of research involves specific considerations for employing other emerging packet-switched networking technologies for space, such as the Advanced Switching Interconnect. In addition, a wider application range may be studied

using the model set constructed for this research in order to tackle new and emerging problems in the space sciences field. Future experimental research may also study practical implementation issues associated with the adaptive routing algorithms studied in Chapter 5, and the potential improvements to end-to-end system latencies for critical data studied in Chapter 6. The RapidIO direct-to-SRAM architectural improvement studied in Chapter 6 may also be examined experimentally over a range of applications with varied memory access, computation, and communication patterns. Finally, future simulative and experimental research may also explore the issues and tradeoffs associated with network and system configurations for space that employ reconfigurable hardware to allow the network type or topology to adapt to tailor itself to a given mission or application.

LIST OF REFERENCES

1. W. Peterson, “The VMEbus Handbook, Fourth Edition,” VITA, Scottsdale, Arizona, 1997.

2. PCI Industrial Computers Manufacturers Group, “CompactPCI Specification Short Form PICMG 2.0 R2.1,” Specification, September 1997.

3. Mercury Computer Systems, “RACEway Interlink Functional Specification TC-RWI-FS-1,” Specification, November 2000.

4. S. Parkes, C. McClements, “The SpaceWire Onboard Network for Spacecraft,” *Proc. SpaceOps 2004*, Montreal, Canada, May 17-21, 2004, pp. 1-10.

5. S. Fuller, “RapidIO – The Next Generation Communication Fabric for Embedded Application,” John Wiley & Sons, Inc., West Sussex, England, January 2005.

6. RapidIO Trade Association, updated September 20, 2006, “RapidIO: Home,” retrieved September 24, 2003, from http://www.rapidio.org.

7. R. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, et al., “The RAD750-a Radiation Hardened PowerPC Processor for High Performance Spaceborne Applications,” *Proc. IEEE Aerospace Conference 2001*, Big Sky, MT, March 10-17, 2001, pp. 2263-2272.

8. D. Woerner, “Revolutionary Systems and Technologies for Missions to the Outer Planets,” *Proc. Second 1AA Symposium on Realistic Near-Term Advanced Scientific Space Missions,* Aosta, Italy, June 26-July 1, 1998, pp. 187-192.

9. HyperTransport Technology Consortium, “HyperTransport I/O Link Specification,” Specification, April 2005.

10. InfiniBand Trade Association, “InfiniBand Architecture Specification Release 1.2,” Specification, October 2004.

11. Telecommunications Industry Association, “Electrical Characteristics of Low Voltage Differential Signaling (LVDS) Interface Circuits, Revision 1.2,” Specification, May 2000.

12. Advanced Switching Interconnect Special Interest Group, “ASI Core Architecture Specification, Revision 1.1,” Specification, November 2004.

13. J. Adams, C. Katsinis, W. Rosen, D. Hecht, V. Adams, H. Narravula, S. Sukhtankar, and R. Lachenmaier, "Simulation Experiments of a RapidIO-based Processing Architecture," *Proc. IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, Oct. 8-10, 2001, pp. 336-339.

14. M. McKenny, J. Dines, and D. Harle, "Transporting Multiple Classes of Traffic over a Generic Routing Device – An Investigation of the Performance of the RapidIO Interconnect Architecture," *Proc. 11th Annual IEEE International Conference on Networks*, Sydney, Australia, Sept. 28-Oct. 1, 2003, pp. 39-44.

15. RapidIO Trade Association, "RapidIO Interconnect Specification Documentation Overview," Specification, June 2002.

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

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

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

19. RapidIO Trade Association, "RapidIO Interconnect Specification, Part VIII: Error Management Extensions Specification," Specification, September 2002.

20. RapidIO Trade Association, "RapidIO Hardware Inter-operability Platform (HIP) Document," Specification, November 2002

21. RapidIO Trade Association, "RapidIO Interconnect Specification, Part IX: Flow Control Extensions Specification," Specification, June 2005.

22. RapidIO Trade Association, "RapidIO Interconnect Specification, Part X: Data Streaming Logical Layer, Phase I," Specification, June 2005.

23. RapidIO Trade Association, "RapidIO Interconnect Specification, Part XI: Multicast Extensions," Specification, June 2005.

24. G. Shippen, "RapidIO Technical Deep Dive I: Architecture and Protocol," Motorola Smart Networks Developer Forum 2003, Dallas, TX, March 20-23, 2003.

25. Redswitch, "RS-1001 16-Port, Serial/Parallel RapidIO Switch and PCI Bridge," Product brief, 2002.

26. Tundra Semiconductor Corporation, "Tsi500 Parallel RapidIO Multi-port Switch," Feature sheet, 2003.

27. Tundra Semiconductor Corporation, "Tsi500 Parallel RapidIO Multi-port Switch User Manual," User manual, 2003.

28. Xilinx, “LogiCORE RapidIO 8-bit Port Physical Layer Interface Design Guide,” Xilinx Intellectual Property Solutions, 2003.

29. Xilinx, “Xilinx Solutions for RapidIO,” Presentation, June 2002.

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

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

32. Leopard Logic, “Leopard Logic Unveils RapidIO Implementation Platform based on its Hyperblox Embedded FPGA Cores,” Press release, July 2002.

33. Praesum Communications, “RapidIO Physical Layer 8/16 LP-LVDS Core Product Brief,” Product brief, 2001.

34. G. Schocht, I. Troxel, K. Farhangian, P. Unger, D. Zinn, C. Mick, A. George, and H. Salzwedel, “System-Level Simulation Modeling with MLDesigner,” *Proc. 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Orlando, FL, October 12-15, 2003, pp. 207-212.

35. K. Shanmugen, V. Frost, and W. LaRue, “A Block-Oriented Network Simulator (BONeS),” *Simulation*, vol. 58, no. 2, pp. 83-94.

36. C. Clos, “A Study of Non-blocking Switching Networks,” *The Bell System Technical Journal*, vol. 32, no. 2, 1953, pp. 406-424.

37. T. Nohara, P. Weber, A. Premji, and C. Livingstone, “SAR-GMTI Processing with Canada’s Radarsat 2 Satellite,” *Proc. IEEE Adaptive Systems for Signal Processing, Communications, and Control Symposium*, Lake Louise, Canada, October 1-4, 2000, pp. 376-384.

38. T. Hacker, “Performance Analysis of a Space-based GMTI Radar System using Separated Spacecraft Interferometry,” Master’s thesis, Massachusetts Institute of Technology, May 2000.

39. M. Skalabrin and T. Einstein, “STAP Processing on Multiprocessor Systems: Distribution of 3-D Data Sets and Processor Allocation for Efficient Interprocessor Communication,” *Proc. Adaptive Sensor Array Processing Workshop*, Lexington, MA, March 13-15, 1996.

40. M. Lee and V. Prasanna, “High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing,” *Proc. 2nd International Workshop on Embedded Systems and Applications*, Geneva, Switzerland, April 5, 1997.

41. R. Brown and R. Linderman, “Algorithm Development for an Airborne Real-Time STAP Demonstration,” *Proc. IEEE National Radar Conference*, Syracuse, NY, May 13-15, 1997, pp. 331-336.

42. M. Linderman and R. Linderman, “Real-Time STAP Demonstration on an Embedded High Performance Computer,” *Proc. IEEE National Radar Conference*, Syracuse, NY, May 13-15, 1997, pp. 54-59.

43. A. Choudhary, W. Liao, P. Varshney, D. Weiner, R. Linderman, and M. Linderman, “Design, Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers,” Technical report, Northwestern University, 1998.

44. 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,” *Proc. IEEE 12th International Parallel Processing Symposium*, Orlando, FL, March 30-April 3, 1998, pp. 220-225.

45. 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. Third IEEE Workshop on High-Speed Local Networks*, Tampa, FL, November 16-18, 2004, pp. 710-717.

46. 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. Eighth Annual Workshop on High Performance Embedded Computing (HPEC)*, Lexington, MA, September 28-30, 2004.

47. D. Bueno, C. Conger, A. Leko, I. Troxel and A. George, "RapidIO-based Space System Architectures for Synthetic Aperture Radar and Ground Moving Target Indicator," *Proc. Ninth Annual Workshop on High Performance Embedded Computing (HPEC)*, Lexington, MA, Sep. 20-22, 2005.

48. J. Kepner, T. Currie, H. Kim, B. Matthew, A. McCabe, M. Moore, D. Rabinkin, A. Reuther, A. Rhoades, L. Tella, and N. Travinn, “Deployment of SAR and GMTI Signal Processing on a Boeing 707 Aircraft using pMatlab and a Bladed Linux Cluster,” *Proc. Eight Annual Workshop on High Performance Embedded Computing*, Lexington, MA, September 28-30, 2004.

49. G. Miller, D. Payne, T. Phung, H. Siegel, and R. Williams, “Parallel Processing of Spaceborne Imaging Radar Data,” *Proc. IEEE/ACM Supercomputing Conference*, San Diego, CA, December 4-8, 1995, pp. 35-41.

50. D. Belcher and C. Baker, “High-resolution Processing of Hybrid Strip-map/Spotlight Mode SAR,” *IEE Proc. Radar, Sonar and Navigation*, vol. 143, no. 6, December 1996, pp. 366-374.

51. K. Chen, "Multistage Interconnection Networks: Improved Routing Algorithms and Fault Tolerance," MSEE Thesis, New Jersey Institute of Technology, May 1991.

52. J. Carpinelli and C. Wang, "Performance of a New Decomposition Algorithm for Rearrangeable Fault-Tolerant Clos Interconnection Networks under Sub-Maximal and No-Fault Conditions," *Advances in Switching Networks: DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 42, July 7-9, 1997, pp. 103-117.

53. S. Chau, J. Smith, A. Tai, "A Design-Diversity Based Fault-Tolerant COTS Avionics Bus Network," *Proc. 2001 Pacific Rim International Symposium on Dependable Computing*, Seoul, South Korea, December 2001.

54. R. Hotchkiss, B. O'Neill, and S. Clark, "Fault Tolerance for an Embedded Wormhole Switched Network," *Proc. International Conference on Parallel Computing in Electrical Engineering*, Quebec, Canada, August 2000.

55. H. Olnowich, D. Kirk, "ALLNODE-RT: A Real Time, Fault Tolerant Network," *Proc. Second Workshop on Parallel and Distributed Real-Time Systems*, Cancun, Mexico, 1994.

56. P. Walker, "Fault-Tolerant FPGA-Based Switch Fabric for SpaceWire: Minimal Loss of Parts and Throughput per Chip Lost," *Proc. 4th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD),* Laurel, MD, September 11-13, 2001.

57. Extreme Networks, "Leveraging Redundancy to Build Fault-Tolerant Networks," Whitepaper, 2002.

58. J. Montanana, J. Flich, A. Robles, P. Lopez, and J. Duato, "A Transition-Based Fault-Tolerant Routing Methodology For InfiniBand Networks," *Proc. 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, April 2004.

59. J. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato. “Supporting Adaptive Routing in InfiniBand Networks,” *Proc. Eleventh Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, February 2003, pp. 165-172.

60. W. Stallings, “Data and Computer Communications, Seventh Edition,” Prentice Hall, Upper Saddle River, NJ, 2004.

61. V. Menasce, "RapidIO as the Foundation for Fault Tolerant Systems," Application note, March 2004.

62. A. Youssef and I. Scherson, "Randomized Routing on Benes-Clos Networks," *Proc. The New Frontiers: A Workshop on Future Directions of Massively Parallel Processing*, McLean, Virginia, October 1992.

63. Y. Yang and J. Wang, "A Fault-Tolerant Rearrangeable Permutation Network," *IEEE Transactions on Computers*, vol. 53, no. 4, April 2004, pp. 414-426.

64. N. Das and J. Dattagupta, "A Fault Location Technique and Alternate Routing in Benes Network," *Proc. Fourth Asian Test Symposium*, Bangalore, India, November 1995.

65. Micrel, “2.5V, 3.2Gbps, Differential 4:1 LVDS Multiplexer with Internal Input Termination,” Product datasheet, 2005.

66. G. Smith, “Enhanced Fault Tolerance for On Board FPGA Based Reconfigurable Computing,” *Proc. 9th Annual International MAPLD Conference*, Washington, D.C., September 26-28, 2006.

67. I. Troxel, E. Grobelny, G. Cieslewski, J. Curreri, M. Fischer, and A. George, "Reliable Management Services for COTS-based Space Systems and Applications," *Proc. International Conference on Embedded Systems & Applications (ESA)*, Las Vegas, NV, June 26-29, 2006.

68. Xilinx, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet,” Product specification, October 10, 2005.

69. Xilinx, “Virtex-4 Family Overview,” Preliminary product specification, February 10, 2006.

70. D. Culler and J. Singh with A. Gupta, “Parallel Computer Architecture: A Hardware/Software Approach,” Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

BIOGRAPHICAL SKETCH

David Bueno obtained his B.S. in computer engineering from the University of Florida in 2003 and his M.E. in electrical and computer engineering from the University of Florida in 2004. He has been a Research Assistant in the Department of Electrical and Computer Engineering at the University of Florida since 2002, and he has also served as a group leader in the High-performance Computing and Simulation (HCS) Research Laboratory since 2003. His primary research interests include both embedded and general-purpose high-performance computer systems and networks.