CARMA: MANAGEMENT INFRASTRUCTURE AND MIDDLEWARE FOR MULTI-PARADIGM COMPUTING

By

IAN A. TROXEL

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

For my wife, Angela, who inspired this work's "speedy" completion.

ACKNOWLEDGMENTS

I thank the members of the HCS lab for their comradeship, council and assistance. I also thank the teachers and professors who have inspired my need to investigate the unknown. And last but certainly not least, I also thank my wife for her love, patience and understanding in the years it took to complete this degree.

TABLE OF CONTENTS

page

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

CARMA: MANAGEMENT INFRASTRUCTURE AND MIDDLEWARE
FOR MULTI-PARADIGM COMPUTING

By

Ian A. Troxel

December 2006

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

A recent trend toward multi-paradigm systems that couple General-Purpose Processors (GPPs) and other special-purpose devices such as Field-Programmable Gate Arrays (FPGAs) has shown impressive speedups. In fact, some researchers see such systems as vital to extending an anticipated slowing of Moore's law. However, extracting the full potential of large-scale, multi-paradigm, High-Performance Computing (HPC) systems has proven difficult due in part to a lack of traditional job management services upon which HPC users have come to rely. In this dissertation, the Comprehensive Approach to Reconfigurable Management Architecture (CARMA) is introduced and investigated. CARMA attempts to address the issue of large-scale, multi-paradigm system utilization by 1) providing a scalable, modular framework standard upon which other scholarly work can build; 2) conducting feasibility and tradeoff analysis of key components to determine strategies for reducing overhead and increasing performance and functionality; and 3) deploying initial prototypes with baseline services to receive feedback from the research community. In order to serve the needs of a wide range of HPC users, versions of CARMA tailored to two unique processing paradigms have been deployed and studied on an embedded space system and Beowulf cluster.

Several scholarly contributions have been made by developing and analyzing the first comprehensive framework to date for job and resource management and services. CARMA is a first step toward developing an open standard with a modular design that will spur inventiveness and foster industry, government and scholarly collaboration much like Condor, Globus and the Open Systems Interconnect model have done for the cluster, grid and networking environments. Developing and evaluating components to fill out the CARMA infrastructure provided key insight into how future multi-paradigm runtime management services should be structured. In addition, deploying initial prototypes with baseline services will open multi-paradigm systems to a wider community of users, shorten the development learning curve, increase the utilization and usability of existing and future systems, reduce the cost of integrating future technology, and enable a wide community of users to share in future design and development efforts. CARMA will increase the usability of existing and future multi-paradigm systems while helping to shape future research.

CHAPTER 1
INTRODUCTION

Infusing special-purpose devices into traditional computational clusters has been proposed as a means to overcome performance and scalability limitations designers will face as systems approach the physical limits of computing [1]. Among many limiting issues, deploying large-scale systems that can provide adequate power density and thermal dissipation at the device [2] and system level [3] has become increasingly difficult with today's powerful but power-hungry microprocessors. Adopting multi-paradigm and heterogeneous system approaches that leverage technologies such as Graphics Processing Units (GPUs), Reconfigurable Computing (RC) devices typically based on Field-Programmable Gate Arrays (FPGAs), and the Cell processor have demonstrated tremendous performance gains by offloading key computation kernels to these specialized coprocessors. By incorporating hardware well suited to the application, designers can more efficiently optimize overall system silicon area utilization and therefore minimize power requirements. In addition to providing improvements in both raw performance and system cost-effectiveness due to the competitive niche markets in which these specialized components were developed, multi-paradigm systems are a potential solution to avoiding a predicted slowing of Moore's Law [4] for systems composed of traditional processors.

While device technology has advanced significantly, there has been a notable lag in design, development and management tools for multi-paradigm systems as compared to the tools designers have come to expect in traditional HPC systems. Although RC and other special-purpose devices have improved the performance of many stand-alone applications, two major challenges in the area of system infrastructure are currently limiting the widespread adoption of these technologies. One challenge is to define representative programming models and create resultant development tools that provide an efficient means to develop parallel applications on

multi-paradigm systems. Though this topic is important in and of itself, numerous research projects are currently addressing this challenge for RC [5]. A related research area that has seen relatively little attention from the research community is the need for a versatile runtime infrastructure and management service that understands the heterogeneity of the underlying system architecture and uses dynamically-gathered runtime information about application performance on these devices to make intelligent management decisions. To achieve the full performance potential of multi-paradigm HPC systems, special-purpose devices must be afforded as much deference in the system as traditional processors currently enjoy, and management systems and tools must be restructured with this goal in mind.

To address this critical need, this dissertation proposes the Comprehensive Approach to Reconfigurable Management Architecture (CARMA). CARMA aims to make executing and debugging jobs in a multi-paradigm HPC environment as easy as it currently is to do so in traditional systems and thereby increase special-purpose device and multi-paradigm system usability. In achieving this goal, a versatile framework and middleware were investigated, designed, developed, deployed, and evaluated. CARMA provides key features that management systems currently lack for multi-paradigm clusters including the following:

- dynamic device discovery and management
- coherent multitasking/multi-user environment
- device-aware job scheduling and management
- design for fault tolerance and scalability
- parallel performance monitoring and debug of the special-purpose devices

The definition of what constitutes an HPC system varies widely. For example, physicists studying high-energy physics of subatomic particles via simulation (e.g., at a national lab) would likely define an HPC system that meets their processing needs on the order of hundreds or thousands of modern processors. In contrast, meteorologists (e.g., at NASA) studying weather

data produced via satellite interferometers on a satellite mission to Jupiter would consider an HPC system of a dozen decade-old processors onboard the spacecraft luxurious. In fact, due to demonstrated performance improvements, the multi-paradigm concept is making its way into systems of all sizes and complexities from satellites to supercomputers, and the need for a run-time infrastructure is a common problem. However, as each processing environment has its own unique adaptations and limitations, so too will CARMA adapt and be tailored to meet each specific environment's needs.

For this dissertation, the CARMA framework is first developed and deployed in an embedded, space-computing environment. Space computing is typically marked by nodes composed of low- to moderate-performance processors, scant per-node storage capacity, and interconnected by low-performance custom networks. Embedded systems are typically designed for custom applications and often require special considerations such as environmental-effects mitigation (e.g., extreme radiation and temperature) and autonomous modes of operation. CARMA was adapted to meet the needs of a particular NASA satellite mission that seeks to replace custom components and instead deploys more inexpensive components that require improved software management. Individual CARMA services for the embedded-computing version increased or diminished in importance as dictated by the environment's requirements. For example, embedded space systems require a robust fault manager to avoid, correct and recover from faults, while this service may play a minor role in some other environments. Tradeoff studies were undertaken to determine the optimal deployment of CARMA services, and the middleware's viability in terms of scalability, response time and overhead was analyzed using benchmark applications. Also, the embedded version of CARMA was extended in another

phase of this dissertation to further improve the overall fault tolerance of the system by including autonomic computing techniques.

For the final phase of this research, the CARMA infrastructure is adapted for the HPC cluster-computing environment. This environment is typically marked by nodes composed of moderate- to high-performance processors, limited per-node storage capacity, and typically interconnected with a message-passing, switched network. Clusters are best suited for general-purpose and scientific applications with moderate computation and communication requirements. A tradeoff analysis is undertaken to determine the optimal deployment of CARMA services within this environment and the CARMA cluster-computing design's viability is demonstrated.

In this dissertation, a background on multi-paradigm systems with an emphasis on reconfigurable computing, traditional job management services and related research is described in Chapter 2. Chapter 3 presents the adaptation of CARMA to embedded space computing, and Chapter 4 investigates the addition of autonomic monitoring and a meta-scheduler to the embedded version. Chapter 5 presents the extension and adaptation of CARMA to large-scale, dual-paradigm clusters, and Chapter 6 summarizes this dissertation with conclusions and directions for future research.

CHAPTER 2
BACKGROUND AND RELATED RESEARCH

In this chapter, the basic concepts of multi-paradigm systems and job management services are presented to provide a background for concrete understanding of CARMA's role. Also, the section concludes with a summary of the relevant research to date related to providing job management services for RC systems.

2.1 Multi-Paradigm Systems

The development of heterogeneous and specialized components based on traditional processor technology augmented to suit a particular class of application is not a new concept. For example, digital-signal processors are perhaps one of the first specialized microprocessors to be developed, and they have been deployed for decades. However, two recent and notable examples of devices that offer a substantial performance potential over traditional processors are the Cell heterogeneous processor and specialized GPUs. The Cell is a heterogeneous device that combines two types of processors, one full-scale traditional processor to handle operating system, I/O transfers and other control tasks, and several "synergistic processing elements" that provide SIMD stream processing for intensive data processing [6]. Beyond the heterogeneous nature of the architecture, the fine-grained clock control features in the device provide an additional level of power-aware processing capability. While the Cell has been initially deployed in the embedded gaming market, recent projects have suggested their deployment in large-scale systems has the potential to improve the performance of many HPC applications [7]. As for specialized processors, the GPU industry has seen a rapid acceleration in rendering performance, onboard processing features and the richness of tools available to developers. Graphics cards available "off-the-shelf" in today's market provide an attractive and cost-effective solution for HPC visualization and imaging applications [8]. In addition, GPUs have

successfully accelerated general-purpose applications such as kernels including the Fast-Fourier Transform, convolution [9] and sparse matrix solvers [10]. Indeed, such systems as the 30-card Stony Brook Cluster [11] have demonstrated attractive speedups over similarly classed contemporary systems and provide a first step toward large-scale adoption of such technology into multi-paradigm HPC systems.

In addition to the processor-centric options presented, a relatively new technology has been gaining attention that is a departure from the fixed-instruction-set Von Neumann architecture that is the hallmark of today's modern general-purpose processors (and therefore is also an artifact inherited by special-purpose processors). Hardware reconfigurable devices employing Reconfigurable Computing (RC) design strategies that tailor their underlying architecture to optimally process a given application have demonstrated impressive performance improvements over traditional processor systems [12]. FPGAs have traditionally formed the underlying architecture for RC systems and recent improvements in FPGA technology have driven a surge in their use as coprocessors (though different device types that provide a more coarse-grained reconfigurable fabric are also showing promise). FPGA coprocessors have demonstrated impressive speedups for numerous applications including radar image processing [13], DNA sequence searches [14], and facial recognition [15] among many others. In addition, speedups of 10× over GPUs have been demonstrated in head-to-head comparisons for imaging applications such as particle graphics simulation [16] and color correction and convolution operations [17]. Due to their ability to optimize execution and therefore reduce total execution time, RC systems are often more power-efficient as compared to traditional and even special-purpose processors for a given application though their FPGA components may nominally require more power [18].

Building upon these successes, several platforms capable of scaling to large system sizes have been successfully augmented with FPGA devices. Several Beowulf clusters have been deployed at various universities, supercomputing centers and government facilities, the 48-node cluster at the Air Force Research Laboratory in Rome, NY, being one example [19], and these systems have shown what can be achieved using off-the-shelf technology. Also, several vendors including Cray, Silicon Graphics and SRC Computers have developed systems based on commodity interconnects that show the great potential of integrating RC devices into future HPC systems.

While device technology has advanced significantly, there has been a notable lag in design, development and management tools for multi-paradigm systems as compared to the tools designers have come to expect in traditional HPC systems. Although RC and other special-purpose devices have improved the performance of many stand-alone applications, two major challenges in the area of system infrastructure are currently limiting the widespread adoption of these technologies. One challenge is to define representative programming models and create resultant development tools that provide an efficient means to develop parallel applications on multi-paradigm systems. Though this topic is important in and of itself, numerous research projects are currently underway and more information can be found elsewhere describing how this challenge is being addressed [5]. A related research area that has seen relatively little attention from the research community is the need for a versatile runtime infrastructure and management service that understands the heterogeneity of the underlying system architecture and uses dynamically-gathered runtime information about application performance on these devices to make intelligent management decisions. To achieve the full performance potential of multi-paradigm HPC systems, special-purpose devices must be afforded as much deference in the

system as traditional processors currently enjoy, and management systems and tools must be restructured with this goal in mind.

2.2 Job Management Services

Users of traditional HPC systems have grown to rely upon a number of Job Management Services (JMS) that improve their productivity. Several tools and parallel-programming models have been developed to seamlessly execute jobs on large-scale, traditional-processor systems. By coordinating system resources, JMS tools increase system utilization and availability and are therefore a key component in any large-scale system. JMS concepts have been refined over a few decades and much research has produced many mature tools and several papers provide good overviews [21]. The remainder of this section highlights the most important details of traditional JMS and Section 2.3 describes research related to managing RC resources.

Job scheduling tools take in jobs from users and determine the nodes upon which the job will execute. Some of the popular tools include the Portable Batch System (PBS) and the Application Level Scheduling (AppLeS). The PBS performs centralized job scheduling using priority-based, preemptive FIFO queues that can be created for various schedule and resource requirements [22]. Tools like PBS are representative of the centralized management philosophy shared by almost all JMS that limits their scalability. As an alternative, the AppLeS framework from the San Diego Supercomputer Center [23] takes a distributed approach to adaptively schedule applications on computational grids. Even though agents are fully distributed, AppLeS has a potential scalability limitation in terms of the number of applications that can execute on a given system. CARMA seeks to provide a scalable framework for cluster computing using a fully-distributed approach that is not application- but rather hardware-specific.

Schedulers rely upon performance monitors to measure and disseminate system health and performance information upon which scheduling decisions are made, and providing this

information in a fast, efficient and distributed manner will be critical for CARMA and other JMS. The Network Weather Service (NWS) from UC-Santa Barbara is a distributed system that periodically monitors and dynamically forecasts the performance level of various network and computational resources using numerical models [24]. However, the extendibility of NWS to include information on special-purpose devices is limited. The Gossip-Enabled Monitoring Service (GEMS), from the University of Florida, is a highly responsive and scalable health and performance resource monitoring service for heterogeneous, distributed systems [25]. The GEMS tool is based on epidemic or gossip protocols which have proven to be an effective means by which failures can be detected in large, distributed systems in an asynchronous manner without the limitations associated with reliable multicasting for group communications. The tool is scalable to thousands of nodes while maintaining fast response times, fault tolerance and low resource utilization requirements. Also, GEMS has an easily extensible interface that has been used to incorporate special-purpose devices and application-specific data as previously described. CARMA could be made to support other monitoring services but the scalability and extensibility of GEMS made it an attractive option for large-scale clusters. Also, Self Reliant is a commercial tool that provides fault-tolerant communication and system monitoring of cluster resources. Though the overhead of this tool does not make it viable for large-scale systems, its fault tolerance features make it attractive for embedded space systems.

Comprehensive tools merge multiple tools to ensure the timely, correct and complete execution of jobs in a fully-featured JMS. Some examples of these monolithic tools include the Condor tools suite, the Load Sharing Facility (LSF), and the Distributed Queuing System (DQS). Condor and the collection of tools developed around its philosophy is an open-source collection of management tools from the University of Wisconsin at Madison and is built on the principle

of distributing batch jobs around a loosely-coupled computer cluster [26]. The Load Sharing Facility (LSF) from the Platform Computing Corporation handles batch, interactive and parallel execution and job scheduling in a heterogeneous UNIX environment. LSF recovers from the loss of the master job handler by having all of the slave job handlers elect a new master and checkpointing is used in the case of job or node failures [27]. DQS from Florida State University was the first non-commercial cluster job manager and the concept was borrowed by the makers of the Network Queuing System. Jobs are prioritized and scheduled based on hard (i.e., required) and soft (i.e., would be helpful but not necessary) resource requirements and distributed to resource queues in either a round-robin or minimal-weighted average load fashion. As for fault tolerance, DQS provides a shadow master scheduler feature, and if a machine crashes, the master scheduler will no longer submit jobs to it [28]. The job queuing philosophy found within DQS (and many other JMS) has been adapted within the CARMA framework.

While comprehensive JMS offer a fully-featured, top-down approach to system management, their features are often monolithic and custom, limiting their extensibility. Also, the majority of these tools are based on a centralized management philosophy which is simple to design but suffers from central point of failures and performance bottlenecks. Several research efforts are underway to build extensive management frameworks within which users can build specific functionality tailored for their needs at multiple levels. These frameworks typically provide an intermediary between many of the other management tools previously described. As such, these tools are also large-scale efforts including the collaboration of hundreds of researchers, where Cactus [29] and Globus [30] are two such frameworks. Such grid-based tools are too broad in scope to form the basis of a multi-paradigm cluster management tool, but future

work may be undertaken to enable CARMA to provide site resource information and scheduling to these tools.

2.3 JMS for Reconfigurable Computing Systems

A thorough background of the RC concept beyond that presented in the introduction is not covered here because numerous publications such as recent works by others such as Compton and Hauck [31] provide excellent introductions to the topic. As previously described, the lack of a versatile runtime infrastructure and management service that understands the heterogeneity of the underlying system architecture and uses dynamically-gathered runtime information about application performance on RC devices to make intelligent management decisions is limiting the widespread adoption of RC devices. Most RC system and FPGA-board vendors develop runtime management systems for their boards, but these tools only provide cursory functionality and are designed with a single-user and often single-node mentality. There have been several notable attempts to provide a feature-rich JMS for dual-paradigm systems, and these include extensions to existing cluster management tools and designing new management frameworks, middleware services, and schedulers. A discussion and brief description of some of the notable projects follow.

2.3.1 Building Upon COTS JMS

In a recent project, researchers at George Washington and George Mason University (GW/GMU) performed a survey and analysis of select JMS (including LSF, CODINE, PBS and Condor), and their findings suggested LSF was the best traditional JMS to extend for use in dual-paradigm systems [32]. GW/GMU researchers developed a simple extension to LSF's External Load Information Manager (ELIM) to provide an additional "hook" for RC resources to be considered in scheduling decisions as shown in Figure 2-11. Effectively, a semaphore is added for each FPGA resource to tell the LSF scheduler which RC resources are available at any given

period of time. If there are no resources available for FPGA execution, the submitted job returns an error code and must be resubmitted at a later time [33]. While an important first step, unfortunately, this semaphore approach to including FPGAs in scheduling decisions relegates their role in the system to that of black-box, peripheral components. As such, only a very limited subset of the services provided by traditional JMS can be made available to the FPGA side of the dual-paradigm machine. For example, LSF cannot provide checkpointing and other fault-tolerance mechanisms, performance and health monitoring or advanced scheduling for the FPGA resource in a seamless fashion because LSF has less information about RC resources than it does of system memory. While the maturity of LSF and other COTS JMS provide an attractive option for incremental inclusion of FPGAs into traditional systems, for FPGAs to play as important a role in the system as processors, management tools must have intimate knowledge of their features and functionality. In addition, LSF and many other traditional JMS cannot easily provide hardware virtualization and other advanced features suggested by the FPGA-paradigm frameworks.

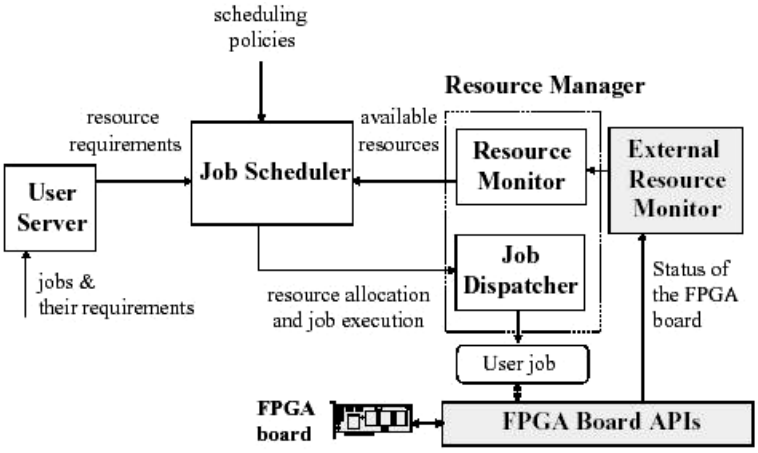


Figure 2-1. LSF Extension for Dual-paradigm Systems (c/o GW/GMU)

Using a similar approach to that taken by GW/GMU, the University of Tennessee has extended its GridSolve grid scheduler and job manager to provide job deployment and result

collection across a grid of distributed RC-enhanced clusters [34]. However, the GridSolve extension provides even fewer features than the LSF extension. GridSolve is a centralized management tool that provides limited fault-tolerance and scalability. Also, the extension provides similar simplistic semaphore-based scheduling and does not provide a task migration feature or any of the traditional JMS or additional features suggested by the FPGA-centric frameworks.

2.3.2 Dual-paradigm Management Frameworks

Two types of management frameworks have been designed by Imperial College, London. One of the frameworks, the Imaging and Graphics Operator Libraries (IGOL), is a layered architecture standard providing clear delineations and interfaces between hardware and software components for both design and execution [35]. IGOL provides a standard, systematic approach to selecting, customizing and distributing hardware configurations as application building blocks. This hierarchical, three-tiered hardware framework supports modular core development and distribution, with generic and RC-specific versions as shown in Figures 2-2 and 2-3, respectively, and allows researchers to focus on their particular niche within an FPGA library design framework. Modules within the IGOL framework conform to the Microsoft Component Object Model (COM) framework allowing access to the many tools available for COM objects. IGOL is primarily intended for code development and testing and does not specify any traditional, run-time JMS but has been integrated with Celoxica's Handel-C development kit.

The other management framework from Imperial College provides a generic framework for the run-time management of FPGA bitstreams [36]. A diagram of the framework's major components is shown in Figure 2-4. Applications request a configuration to be loaded onto the FPGA by the *Loader* from a *Configuration Store*. The Loader asks the *Monitor* to determine the FPGA's configuration state (i.e., which configuration is loaded if any and whether it is in use or

not) and either loads the configuration or sends a busy response back to the application. This run-time manager provides an important agent-based framework that abstracts control of the FPGA away from the user. However, the framework does not include other JMS aspects required for dual-paradigm machines (ex. processor considerations) and would not necessarily scale well to large-scale systems. The framework is as yet undeveloped and untested.

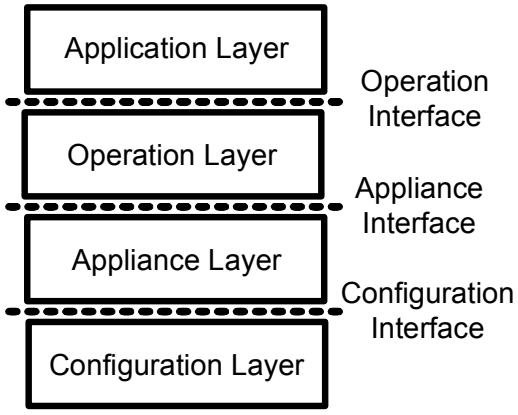


Figure 2-2. Generic IGOL Abstraction Layers (c/o Imperial College)

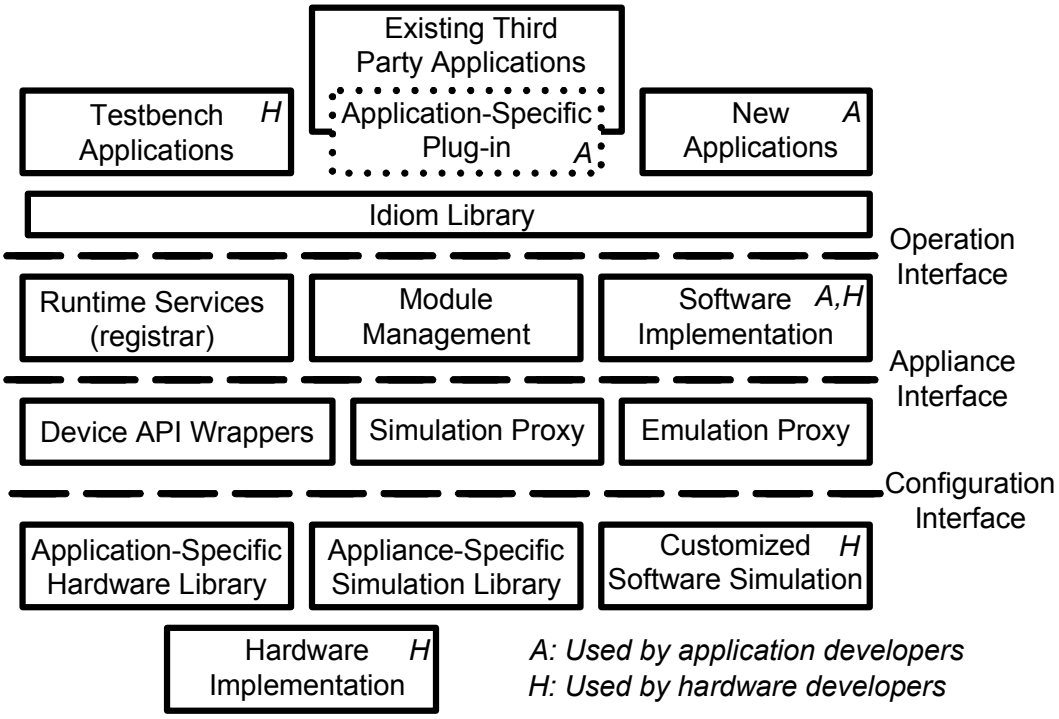


Figure 2-3. RC-specific IGOL Abstraction Layers (c/o Imperial College)

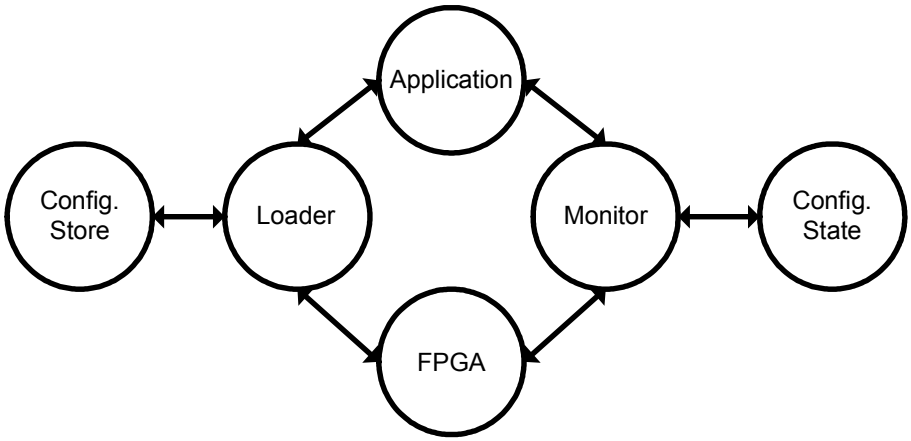


Figure 2-4. Run-time Management Framework (c/o Imperial College)

The University of Glasgow's Reconfigurable Architecture Group (RAGE) has developed a virtual hardware manager for FPGA systems. The RAGE system provides a run-time abstraction layer for users to reconfigure FPGAs without directly accessing device drivers, reserve chunks of FPGA resources for partially-reconfigurable systems and a circuit transformation capability to change the location of post-PAR designs [37]. The RAGE run-time framework is shown in Figure 2-5. Much like the Imperial College run-time manager, the RAGE manager provides an important agent-based framework that abstracts control of the FPGA away from the user. However, the framework also does not include other JMS aspects required for dual-paradigm machines (e.g., job management) and would not necessarily scale well to large-scale systems. The framework has been deployed in a limited capacity and work has progressed on component-design abstraction and transformation with no additional work on the run-time manager to date.

The Distributed Reconfiguration Metacomputer (DRMC) from the University College Cork, Ireland, provides an environment in which computations can be constructed in a high-level manner and executed on dual-paradigm clusters [38]. DRMC, as shown in Figure 2-6, is built on top of the peer-to-peer metacomputer called LinuxNOW [39]. Users create a condensed computation graph detailing how to execute their job including the task's processor executables

and FPGA bitstreams as well as data dependencies between the tasks. The condensed graph engine submits instructions based on this graph to the system scheduler which in turn submits execution threads to the processor and FPGA via a dedicated agent for each resource in the system. DRMC provides a fully-distributed, run-time management layer for dual-paradigm systems that provides many of the JMS features found in traditional clusters.

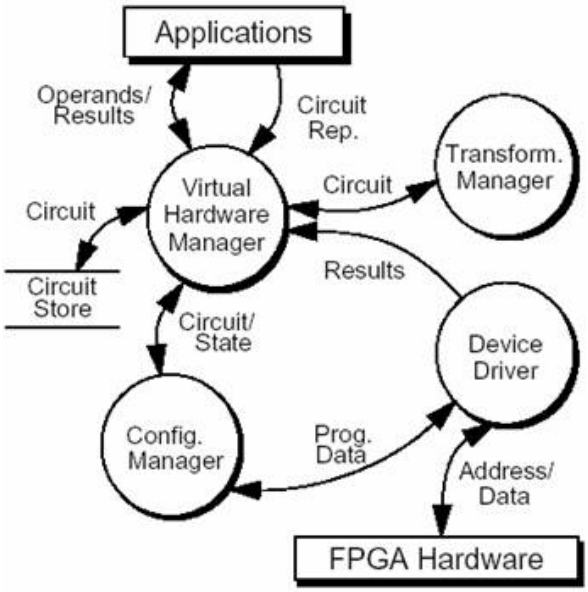


Figure 2-5. RAGE Run-time Management Framework (c/o University of Glasgow)

However, there are a few features that limit DRMC's ability. Traditional message-passing libraries such as MPI cannot be used within the framework for parallel processing due to the condensed graph computing model in which programs are specified. While this feature provides the advantage of abstracting inter-process communication from the user, run-time communication decisions can be poorly optimized for an application's communication patterns, severely limiting performance. Also, legacy code written in other programming models cannot be easily ported to this new communication syntax. Additionally, LinuxNOW was built for small-scale systems and does not scale up to hundreds or thousands of nodes. The designers acknowledge this limitation and are looking into a hierarchical version with limited global-

scheduling and execution facilities. One final limitation is a lack of advanced JMS features such as checkpointing and FPGA-resource monitoring.

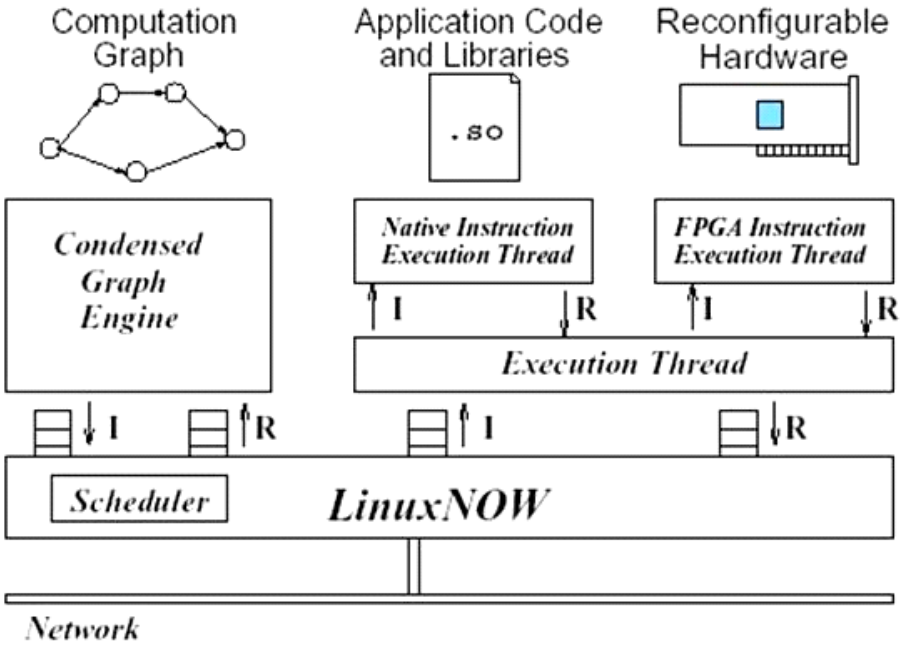


Figure 2-6. DRMC Run-time Management Framework (c/o University College Cork)

The Hybridthreads research initiative at the University of Kansas seeks to blur the line between hardware and software processing at the thread level. The service provides a collection of managers that determine how and when software and hardware threads have access to resources within a node, taking a holistic view of the processors and RC devices [40]. While a theoretically interesting concept, much research and development remains to show the merits of the concept, especially to determine the scalability of the design. Also, the service lacks some traditional JMS features by focusing primarily on the mutex lock requirements to ensure proper access serialization. The authors suggest another JMS tool may be added above their system and CARMA is a tool that fits into the category.

2.3.3 Middleware APIs

As part of the SLAAC project, researchers at Virginia Polytechnic University developed the Adaptive Computing System (ACS) and its Application Programming Interface (API). Each

FPGA board vendor typically provides an API in order for users to link their applications to the FPGA accelerator. However, a code rewrite would be necessary should a user wish to move their code to another vendor's platform. The ACS API project proposed to provide a standard API to allow for code portability across platforms and also to build a unified middleware for systems composed of heterogeneous FPGA boards. The developers clearly state that the ACS API is not a programming language, simply a middleware API which simply loads bitfiles generated by other design capture tools. The API also includes some limited MPI functions to provide an additional layer of inter-process communication abstraction. The ACS API provides hardware-virtualization features required for a multi-user/multi-job environment but does not do so in a scalable manner [41]. The ADAPTERS tool, a joint project from the Honeywell Technology Center, the University of Southern California and other universities sought to provide a domain-specific, integrated design and run-time management system built on top of the ACS API [42]. Unfortunately, these efforts along with many others were not further pursued after DARPA funding expired for RC systems research.

A related standardization project at Xilinx, one of the leading FPGA manufacturers, known as the Xilinx HardWare InterFace (XHWIF) sought to develop a common FPGA hardware interface [43]. While this much needed standard did gain some ground, the underlying programming tool that supported the XHWIF API is no longer supported by the manufacturer. The Universal Standard for Unified Reconfigurable Platforms (USURP) project at the University of Florida is attempting to fill in the holes left by these projects and develop a much needed standard API and hardware interface [5]. CARMA is designed to easily interface with other tools and standards and is interoperable with the USURP framework and more information on this connection is found in Chapter 5.

2.3.4 Custom Tools and Schedulers

Janus from Virginia Tech provides an architecture-independent run-time environment based on JHDL. Janus was designed to perform algorithm design exploration and therefore does not attempt to provide many traditional JMS such as scheduling, process migration, etc. However, the tool does provide a measure of job multitasking by specifically addressing run-time reconfiguration and time-multiplexed hardware virtualization. Janus provides the first step toward a run-time management tool for JHDL, the object-oriented nature of which lends itself well to hardware abstraction [44]. However, the overhead imposed by Java virtual machines may mitigate the performance acceleration gained by RC devices.

A scheduler for the Stream Computations Organized for Reconfigurable Execution (SCORE) project was developed in a joint effort by UC-Berkeley and the California Institute of Technology. Applications are specified in a dataflow graph format that contains computational operators and their dependencies which communicate via streams of data tokens. The graph explicitly describes producer-consumer relationships much like Directed Acyclic Graphs used by Condor and other traditional JMS. Individual operators can be made to execute in parallel to form token-based pipelines [45]. The SCORE scheduler follows a quasi-static scheduling scheme that produces a static schedule for a dataflow graph which can adapt to slowly varying run-time characteristics (such as changes in the dataflow graph composition as operations complete) only when these characteristics vary widely in the coarse of execution [46].

Researchers at the University of Complutense, Madrid, Spain and UC-Irvine have developed a compile-time scheduler that statically generates a subset of all optimal run-time schedules. A simple run-time scheduler then chooses one of these schedules at run-time based on a set of predefined criteria [47]. This method reduces run-time scheduling overhead but may lead to a relatively poor schedule as only a subset of all possible schedules are available. Also,

the lack of runtime information known a priori may lead to schedules that are completely unsuitable. It should be noted that numerous research projects have been devoted to static temporal and spatial partitioning for multi-FPGA systems [48, 49] but this work is considered out of the scope of this discussion. Many of these scheduling tools and techniques are not made for dual-paradigm machines but their advanced FGPA scheduling techniques are likely needed for future dual-paradigm schedulers.

2.3.5 Job Management Service Summary

Traditional JMS and operating systems allow users and developers to focus on their particular application by providing a coherent multi-user/multitasking environment with appropriate resource management and a high degree of code interoperability and portability through abstraction. Application engineers and scientists have come to rely upon JMS features such as automated job and task scheduling, system performance and health monitoring, fault recovery, and data management and delivery. Numerous JMS tools have been developed for traditional processor systems, however none of these tools understand or target RC devices in a true dual-paradigm fashion. Some initial efforts to provide a framework or collection of middleware services are either geared toward hardware design only, have a limited single-user view of the system, are vendor proprietary or have limited scalability beyond ten or so nodes. Many of the tools available today inherently lack scalability, fault tolerance and hardware agnosticism that will be necessary to execute computationally demanding applications on future large-scale, dual-paradigm systems.

There is a great need to provide an all-encompassing framework and collection of middleware services for multi-paradigm systems to meet the needs of traditional HPC users. A successful framework should borrow the best concepts from past work including: an agent-based, hierarchical, and modular framework; supporting hardware virtualization with run-time

support for platform independence; be composed of fully-distributed services for scalability and fault tolerance; and the JMS should have intimate knowledge of RC systems and applications. The novel Comprehensive Approach to Reconfigurable Management Architecture (CARMA) is proposed to address these specific needs. CARMA and this dissertation aim to make executing jobs in dual-paradigm HPC environment as easy and robust as executing jobs in traditional environments.

The CARMA framework will provide the means to achieve the full performance potential of multi-paradigm systems and improve their accessibility for mainstream users. Specific needs that will be addressed by CARMA's versatile runtime framework and system management service include: dynamic RC fabric discovery and management; coherent multitasking, multi-user environment; robust job scheduling and management; design for fault tolerance and scalability; heterogeneous system support; device-independent programming model; debug and system health monitoring; and performance monitoring of the RC fabric. In achieving this goal, CARMA will meet three main objectives including: 1) provide a modular framework for researchers to extend and adjust as necessary to meet their own specific needs to spur inventiveness much like Condor, Globus and the Open Systems Interconnect [50] model have done for cluster and grid computing and all forms of networking, respectively; 2) analyze the feasibility of the framework by developing key components and refining their organization and structure; and 3) deploy an initial prototype with baseline services to open multi-paradigm systems up to a wider community of researchers and receive feedback from their experiences. The next chapter shows how the CARMA concept has been successfully designed and investigated for embedded space computing.

CHAPTER 3
CARMA FOR EMBEDDED COMPUTING

With the ever-increasing demand for higher bandwidth and processing capacity of today's space exploration, space science, and defense missions, the ability to efficiently apply commercial-off-the-shelf technology for on-board computing is now a critical need. In response to this need, NASA's New Millennium Program office has commissioned the development of the Dependable Multiprocessor for use in payload and robotic missions. The Dependable Multiprocessor system provides power-efficient, high-performance, fault-tolerant cluster computing resources in a cost-effective and scalable manner. As a major step toward the flight system to be launched in 2009, Honeywell and the University of Florida have successfully investigated and developed a CARMA-based management system and associated middleware components to make the processing of science-mission data as easy in space as it is in ground-based clusters. This chapter provides a detailed description of the Dependable Multiprocessor's middleware technology and experimental results validating the concept and demonstrating the system's scalability even in the presence of faults.

3.1 Introduction

NASA and other space agencies have had a long and relatively productive history of space exploration as exemplified by recent rover missions to Mars. Traditionally, space exploration missions have essentially been remote-control platforms with all major decisions made by operators located in control centers on Earth. The onboard computers in these remote systems have contained minimal functionality, partially in order to satisfy design size and power constraints, but also to reduce complexity and therefore minimize the cost of developing components that can endure the harsh environment of space. Hence, these traditional space computers have been capable of doing little more than executing small sets of real-time

spacecraft control procedures, with little or no processing features remaining for instrument data processing. This approach has proven to be an effective means of meeting tight budget constraints because most missions to date have generated a manageable volume of data that can be compressed and post-processed by ground stations.

However, as outlined in NASA's latest strategic plan and other sources, the demand for onboard processing is predicted to increase substantially due to several factors [51]. As the capabilities of instruments on exploration platforms increase in terms of the number, type and quality of images produced in a given time period, additional processing capability will be required to cope with limited downlink bandwidth and line-of-sight challenges. Substantial bandwidth savings can be achieved by performing preprocessing and, if possible, knowledge extraction on raw data in situ. Beyond simple data collection, the ability for space probes to autonomously self-manage will be a critical feature to successfully execute planned space exploration missions. Autonomous spacecraft have the potential to substantially increase return on investment through opportunistic explorations conducted outside the Earth-bound operator control loop. In achieving this goal, the required processing capability becomes even more demanding when decisions must be made quickly for applications with real-time deadlines. However, providing the required level of onboard processing capability for such advanced features and simultaneously meeting tight budget requirements is a challenging problem that must be addressed.

In response, NASA has initiated several projects to develop technologies that address the onboard processing gap. One such program, NASA's New Millennium Program (NMP), provides a venue to test emergent technology for space. The Dependable Multiprocessor (DM) is one of the four experiments on the upcoming NMP Space Technology 8 (ST8) mission, to be

launched in 2009, and the experiment seeks to deploy Commercial-Off-The-Shelf (COTS) technology to boost onboard processing performance per watt [52]. The DM system combines COTS processors and networking components (e.g., Ethernet) with an adapted version of CARMA providing a novel and robust middleware system for customized application deployment and recovery. This system thereby maximizes system efficiency while maintaining the required level of reliability by adapting to the harsh environment of space. In addition, the CARMA-enabled DM system middleware provides a parallel processing environment comparable to that found in high-performance COTS clusters of which application scientists are familiar. By adopting a standard development strategy and runtime environment, the additional expense and time loss associated with porting of applications from the laboratory to the spacecraft payload can be significantly reduced.

The remainder of this chapter presents and examines the performance of portions of the CARMA-based components developed for the DM system's management middleware and is divided as follows. Section 3.2 presents past projects that have inspired the development of the DM system and other related research. Sections 3.3 and 3.4 provide an overview of the DM architecture and management software respectively. The prototype flight system and ground-based cluster on which the DM concept is being evaluated is described in Section 3.5, and Section 3.6 presents experimental performance analysis highlighting the scalability of the embedded versions of CARMA. Conclusions and future work for this phase of research are highlighted in Section 3.7.

3.2 Related Work

The design of the Dependable Multiprocessor builds upon a legacy of platforms and aerospace onboard computing research projects and the rich collection of tools and middleware concepts from the cluster computing paradigm. For example, the Advanced Onboard Signal

Processor (AOSP), developed by Raytheon Corporation for the USAF in the late 1970s and mid 1980s, provided significant breakthroughs in understanding the effects of natural and man-made radiation on computing systems and components in space [53]. The AOSP, though never deployed in space, was instrumental in developing hardware and software architectural design techniques for detection, isolation, and mitigation of radiation effects and provided the fundamental concepts behind much of the current work in fault-tolerant, high-performance distributed computing. The Advanced Architecture Onboard Processor (AAOP), a follow-on project to AOSP also developed at Raytheon Corporation, employed self-checking RISC processors and a bi-directional, chordal skip ring architecture in which any failed system element could be bypassed by using redundant skip links in the topology [54]. While the AAOP architecture provided an alternative that improved system fault tolerance versus contemporary custom-interconnect designs, the offered performance could not justify the additional power consumption of the system.

The Remote Exploration Experimentation (REE) project championed by NASA JPL sought to develop a scalable, fault-tolerant, high-performance supercomputer for space composed of COTS processors and networking components [55]. The REE system design deployed a middleware layer, the Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR), between a commercial operating system and applications that offered a customizable level of fault tolerance based on reliability and efficiency requirements. Within ARMOR, a centralized fault-tolerance manager oversees the correct execution of applications through remote agents that are deployed and removed with each application execution [56]. Through ARMOR, the system sought to employ Software-Implemented Fault Tolerance (SIFT) techniques to mitigate radiation-induced system faults without hardware replication. The

preliminary implementation of this system was an important first step and showed promise with testing performed via a fault-injection tool. Unfortunately, system deployment was not achieved and scalability analysis was not undertaken. In developing the DM middleware, insight was gleaned from the REE system and the DM design will address some of the perceived shortcomings in terms of the potential scalability limitations of the REE middleware.

In addition to using traditional COTS processors, there have been a few significant efforts to incorporate Field-Programmable Gate Arrays (FPGAs) as compute resources to boost performance on select application kernels. The Australian scientific satellite, FedSat, launched in December 2002, sought to improve COTS processing power by deploying powerful FPGA co-processors to accelerate remote-sensing applications [57]. FedSat's payload consisted of a microprocessor, a reprogrammable FPGA, a radiation-hardened, antifuse-based FPGA to perform reconfiguration and configuration scrubbing, and memory (see Figure 3-1). A follow-on project supported by NASA LARC, Reconfigurable Scalable Computing (RSC), proposes a compute cluster for space composed entirely of triple-redundant MicroBlaze softcore processors running uC-Linux and hosted on Xilinx Virtex-4 FPGAs [58]. The DM payload will also employ FPGAs to improve performance using a combination of SIFT, internal FPGA mechanisms, and periodic scrubbing to overcome faults.

Beyond space systems, much research has been conducted to improve the fault tolerance of ground-based computational clusters. Although ground-based systems do not share the same strict power and environmental constraints as space systems, improving fault tolerance and availability is nonetheless important as such systems frequently execute critical applications with long execution times. One of several notable research projects in this area is the Dynamic Agent Replication eXtension (DARX) framework that provides a simple mechanism for deploying

replicated applications in an agent-based distributed computing environment [59]. The modular and scalable approach provided by DARX would likely provide performance benefits for ground-based systems but may be too resource-intensive to appropriately function on resource-limited embedded platforms. By contrast, the management system used in the UCLA Fault-Tolerant Cluster Testbed (FTCT) project performs scheduling, job deployment and failure recovery based on a central management group composed of three manager replicas [60]. While the central approach taken by the FTCT design reduces system complexity, hot-sparing manager resources can strain limited system resources and can become a bottleneck if not carefully designed. The next section provides an overview of the DM system architecture with high-level descriptions of the major components.

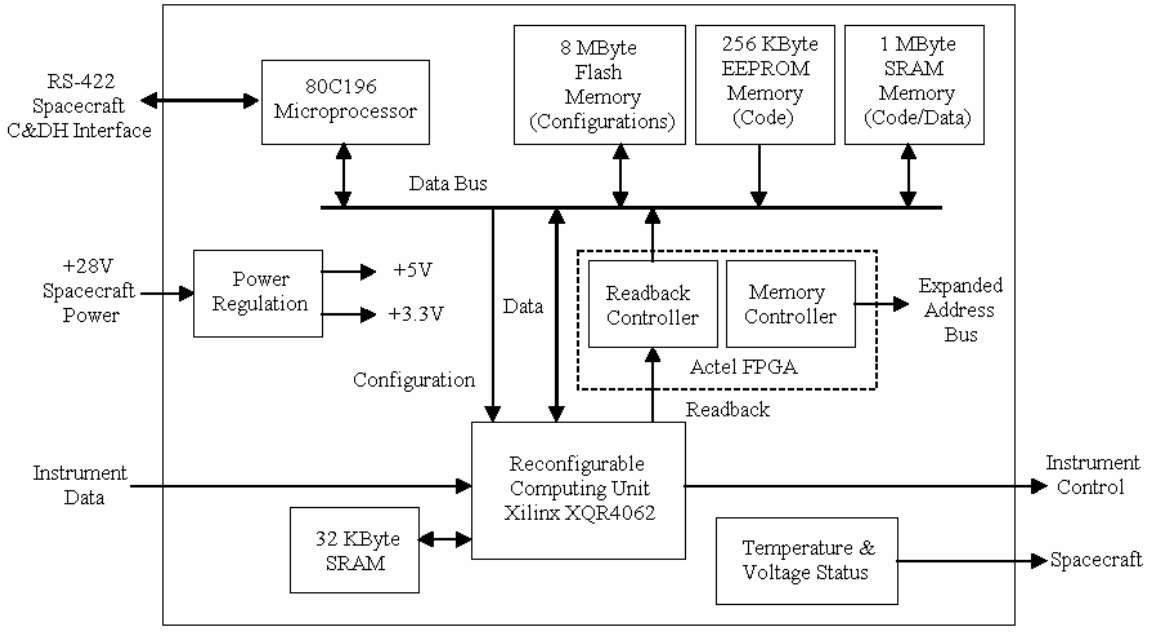


Figure 3-1. FedSat Reconfigurable Computing Payload (c/o John's Hopkins APL)

3.3 System Architecture

Building upon the strengths of past research efforts, the DM system provides a cost-effective, standard processing platform with a seamless transition from ground-based

computational clusters to space systems. By providing development and runtime environments familiar to earth and space science application developers, project development time, risk and cost can be substantially reduced. The DM hardware architecture (see Figure 3-2) follows an integrated-payload concept whereby components can be incrementally added to a standard system infrastructure inexpensively [61]. The DM platform is composed of a collection of COTS data processors (augmented with runtime-reconfigurable COTS FPGAs) interconnected by redundant COTS packet-switched networks such as Ethernet or RapidIO [62]. To guard against unrecoverable component failures, COTS components can be deployed with redundancy, and the choice of whether redundant components are cold or hot spares is mission-specific. The scalable nature of non-blocking switches provides distinct performance advantages over traditional bus-based architectures and also allows network-level redundancy to be added on a per-component basis. Additional peripherals or custom modules may be added to the network to extend the system's capability; however, these peripherals are outside of the scope of the base architecture.

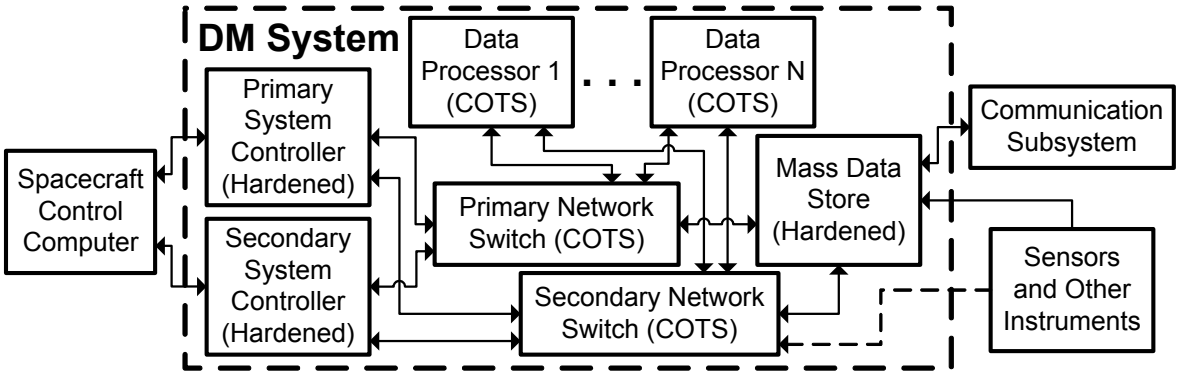


Figure 3-2. System Hardware Architecture of the Dependable Multiprocessor

Future versions of the DM system may be deployed with a full complement of COTS components but, in order to reduce project risk for the DM experiment, components that provide critical control functionality are radiation-hardened in the baseline system configuration. The

DM is controlled by one or more *System Controllers*, each a radiation-hardened single-board computer, which monitor and maintain the health of the system. Also, the system controller is responsible for interacting with the main controller for the entire spacecraft. Although system controllers are highly reliable components, they can be deployed in a redundant fashion for highly critical or long-term missions with cold or hot sparing. A radiation-hardened *Mass Data Store (MDS)* with onboard data handling and processing capabilities provides a common interface for sensors, downlink systems and other "peripherals" to attach to the DM system. Also, the MDS provides a globally accessible and secure location for storing checkpoints, I/O and other system data. The primary dataflow in the system is from instrument to MDS, through the cluster, back to the MDS, and finally to the ground via the spacecraft's *Communication Subsystem*. Because the MDS is a highly reliable component, it will likely have an adequate level of reliability for most missions and therefore need not be replicated. However, redundant spares or a fully distributed memory approach may be required for some missions. In fact, results from an investigation of the system performance suggest that a monolithic and centralized MDS may limit the scalability of certain applications and this work is presented in a previous publication [63]. The next section provides a detailed description of the CARMA-based middleware that ensures reliable processing in the DM system.

3.4 Middleware Architecture

The DM middleware has been designed with the resource-limited environment typical of embedded space systems in mind and yet is meant to scale up to hundreds of data processors per the goals for future generations of the technology. A top-level overview of the DM software architecture is illustrated in Figure 3-3. A key feature of this architecture is the integration of generic job management and software fault-tolerant techniques implemented in the middleware framework. The DM middleware is independent of and transparent to both the specific mission

application and the underlying platform. This transparency is achieved for mission applications through well-defined, high-level, Application Programming Interfaces (APIs) and policy definitions, and at the platform layer through abstract interfaces and library calls that isolate the middleware from the underlying platform. This method of isolation and encapsulation makes the middleware services portable to new platforms.

To achieve a standard runtime environment with which science application designers are accustomed, a commodity operating system such as a Linux variant forms the basis for the software platform on each system node including the control processor and mass data store (i.e., the Hardened Processor seen in Figure 3-3). Providing a COTS runtime system allows space scientists to develop their applications on inexpensive ground-based clusters and transfer their applications to the flight system with minimal effort. Such an easy path to flight deployment will reduce project costs and development time, ultimately leading to more science missions deployed over a given period of time. A description of the other DM middleware components (synonymously referred to as services or agents) follows.

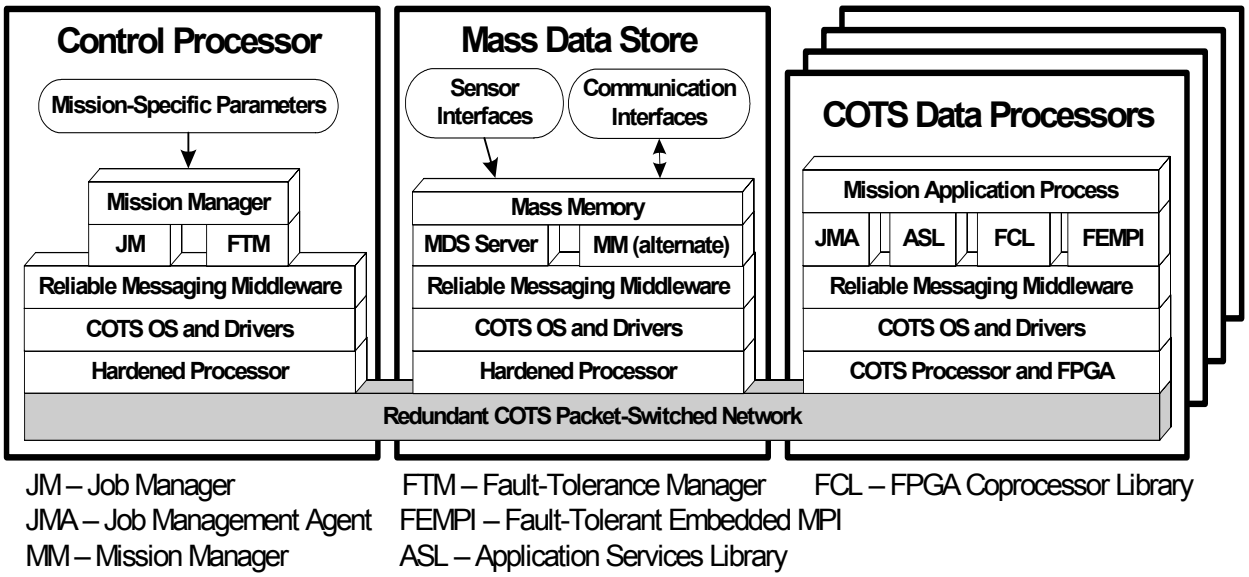


Figure 3-3. System Software Architecture of the Dependable Multiprocessor

3.4.1 Reliable Messaging Middleware

The Reliable Messaging Middleware provides a standard interface for communicating between all software components, including user application code, over the underlying packet-switched network. The messaging middleware provides guaranteed and in-order delivery of all messages on either the primary or secondary networks and does so in a scalable manner. Inter-processor communication includes numerous critical traffic types such as checkpoint information, error notifications, job and fault management control information, and application messages. Thus, maintaining reliable and timely delivery of such information is essential.

To reduce development time and improve platform interoperability, a commercial tool with cross-platform support from GoAhead, Inc., called SelfReliant (SR), serves as the reliable messaging middleware for the DM system. SR provides a host of cluster availability and management services but the DM prototype only uses the reliable distributed messaging, failure notification, and event logging services. The messaging service within SR is designed to address the need for intra- and inter-process communications between system elements for numerous application needs such as checkpointing, client/server communications, event notification, and time-critical communications. SR facilitates the DM messaging middleware by managing distributed virtual multicast groups with publish and subscribe mechanisms over primary and secondary networks. In this manner, applications need not manage explicit communication between specific nodes (i.e., much like socket communication) and instead simply publish messages to a virtual group that is managed by SR. The messaging service provides an effective and uniform way for distributed messaging components to efficiently communicate and coordinate their activities.

SR's failure notification service provides liveliness information about any agent, user application, and physical cluster resource via lightweight heartbeat mechanisms. System

resources and their relationships are abstractly represented as objects within a distributed database managed by SR and shared with the Fault-Tolerance Manager (FTM), which in turn uses liveliness information to assess the system's health. Agent and application heartbeats are managed locally within each node by the service and only state changes are reported externally by a lightweight watchdog process. In the system, the watchdog processes are managed in a hierarchical manner by a lead watchdog process executing on the system controller. State transition notifications from any node may be observed by agents executing on other nodes by subscribing to the appropriate multicast group within the reliable messaging service. Also, SR is responsible for discovering, incorporating, and monitoring the nodes within the cluster along with their associated network interfaces. The addition or failure of nodes and their network interfaces is communicated within the watchdog process hierarchy. Extensions to the baseline SR framework have been developed to interface the FTM, described in the next section, to the notification hierarchy. In particular, the extensions allow the FTM to detect when a service or application (including SR itself), has initialized correctly or failed. Also, one of the extensions provides a mechanism by which the FTM starts other middleware services in a fault-tolerant manner.

3.4.2 Fault-Tolerance Manager

Two centralized agents that execute on the hardened system controller manage the basic job deployment features of the DM cluster. The Fault-Tolerance Manager (FTM) is the central fault recovery agent for the DM system. The FTM collects liveliness and failure notifications from distributed software agents and the reliable messaging middleware in order to maintain a table representing an updated view of the system. The distributed nature of the agents ensures the central FTM does not become a monitoring bottleneck, especially since the FTM and other central DM software core components execute simultaneously on a relatively slow radiation-

hardened processor. Numerous mechanisms are in place to ensure the integrity of remote agents running on radiation-susceptible data processors. Update messages from the reliable middleware are serviced by dedicated threads of execution within the FTM on an interrupt basis to ensure such updates occur in a timely fashion. If an object's health state transitions to failed, diagnostic and recovery actions are triggered within the FTM per a set of user-defined recovery policies. Also, the FTM maintains a fault history of various metrics for use in the diagnosis and recovery process. This information is also used to make decisions about system configuration and by the Job Manager (JM) for application scheduling to maximize system availability.

Recovery policies defined within the DM framework are application-independent yet user configurable from a select set of options. To address failed system services, the FTM may be configured to take recovery actions including performing a diagnostic to identify the reason for the failure and then directly addressing the fault by restarting the service from a checkpoint or from fresh. Other recovery options include performing a software-driven or power-off reboot of the affected system node or shutting the node down and marking it as permanently failed until directed otherwise. For application recovery, users can define a number of recovery modes based on runtime conditions. This configurability is particularly important when executing parallel applications using the Message Passing Interface (MPI). The job manager frequently directs the recovery policies in the case of application failures and more information on FTM and JM interactions is provided in the next section.

In addition to implementing recovery policies, the FTM provides a unified view of the embedded cluster to the spacecraft control computer and the ground-based station user. It is the central software component through which the embedded system provides liveliness information to the spacecraft. The spacecraft control computer (see Figure 3-2) detects system failures via

missing heartbeats from the FTM. When a failure is discovered, a system-wide reboot is employed. In addition to monitoring system status, the FTM interface to the spacecraft control computer also presents a mechanism to remotely initiate and monitor diagnostic features provided by the DM middleware.

3.4.3 Job Manager and Agents

The primary functions of the DM Job Manager (JM) are job scheduling, resource allocation, dispatching processes, and directing application recovery based on user-defined policies. The JM employs an opportunistic load balancing scheduler, with gang scheduling for parallel jobs, which receives frequent system status updates from the FTM in order to maximize system availability. Gang scheduling refers to the requirement that all tasks in a parallel job be scheduled in an "all-or-nothing" fashion [64]. Jobs are described using a Directed Acyclic Graph (DAG) and are registered and tracked in the system by the JM via tables detailing the state of all jobs be they pending, currently executing, or suspected as failed and under recovery. These various job buffers are frequently checkpointed to the MDS to enable seamless recovery of the JM and all outstanding jobs. The JM heartbeats to the FTM via the reliable middleware to ensure system integrity and, if an unrecoverable failure on the control processor occurs, the backup controller is booted and the new JM loads the checkpointed tables and continues job scheduling from the last checkpoint. A more detailed explanation of the checkpoint mechanisms is provided in Section 3.4.7.

Much like the FTM, the centralized JM employs distributed software agents to gather application liveliness information. The JM also relies upon the agents to fork the execution of jobs and to manage all required runtime job information. The distributed nature of the Job Management Agents (JMAs) ensures that the central JM does not become a bottleneck, especially since the JM and other central DM software core components execute simultaneously

on a relatively slow radiation-hardened processor. Numerous mechanisms are in place to ensure the integrity of the JMAs executing on radiation-susceptible data processors. In the event of an application failure, the JM refers to a set of user-defined policies to direct the recovery process. In the event one or more processes fail in a parallel application (i.e., one spanning multiple data processors) then special recovery actions are taken. Several recovery options exist for parallel jobs, such as defined in related research from the University of Tennessee at Knoxville [65]. These options include a mode in which the application is removed from the system, a mode where the application continues with an operational processor assuming the extra workload, a mode in which the JM either migrates failed processes to healthy processors or instructs the FTM to recover the faulty components in order to reconstruct the system with the required number of nodes, and a mode where the remaining processes continue by evenly dividing the remaining workload amongst themselves. As mentioned, the ability of a job to recover in any of these modes is dictated by the underlying application. In addition, the Mission Manager (as seen in Figure 3-3) provides further direction to the JM to ensure recovery does not affect mission-wide performance. More information on the interaction between the mission manager and the JM is described in the next section.

3.4.4 Mission Manager

The Mission Manager (MM) forms the central decision-making authority in the DM system. The MM is deployed on a radiation-hardened processor for increased system dependability but may execute on the control processor or alternatively on the MDS if enough processing power exists on the device (denoted MM alternate in Figure 3-3). Deploying the MM on the MDS provides a slight performance advantage for several of its features but the transparent communication facilities provided by the reliable communication layer APIs allow for great flexibility in the design. The MM interacts with the FTM and JM to effectively

administer the system and ensure mission success based on three primary objectives: 1) deploy applications per mission policies, 2) collect health information on all aspects of the system and environment to make appropriate decisions, and 3) adjust mission policies autonomously per observations. The MM was extended to provide autonomic computing features and a detailed analysis of this important component is provided in the next chapter. The next several sections describe the various application interface libraries provided by the DM system.

3.4.5 FEMPI

The DM system employs an application-independent, fault-tolerant, message-passing middleware called FEMPI (Fault-tolerant Embedded Message Passing Interface). With FEMPI, we take a direct approach to providing fault tolerance and improving the availability of the HPC system in space. FEMPI is a lightweight, fault-tolerant design and implementation that provides process-level fault tolerance to the common Message Passing Interface (MPI) standard [66]. With MPI applications, failures can be broadly classified as process failures (individual processes of MPI application crashes) and network failures (communication failure between two MPI processes). FEMPI ensures reliable communication (reducing the chances of network failures) through the reliable messaging middleware. As far as process failures are concerned, any single failed process in a regular fault-intolerant MPI design will crash the entire application. By contrast, FEMPI prevents the entire application from crashing on individual process failures.

Fault tolerance and recovery is provided through three stages including detection of a fault, notification of the fault, and recovery from the fault. The FEMPI runtime services employ the features provided by the reliable messaging middleware in conjunction with the FTM and JM as shown in Figure 3-4. User applications heartbeat to the JMA via well-defined APIs and the JMA informs the FTM of any process failures by updating the system model. The FTM in turn informs the JM of an application failure and the JM directs FEMPI's runtime features (i.e., the

MPI Restore function) to perform a recovery based on the user-defined policy. On a failure, MPI Restore informs all the MPI processes of the failure. The status of senders and receivers (of messages) are checked in FEMPI before initiating communication to avoid attempts to establish communication with failed processes. If the communication partner (sender or receiver) fails after the status check and before communication, then a timeout-based recovery is used to recover out of the MPI function call. FEMPI can survive the crash of n-1 processes in an n-process job, and, if required, the system can re-spawn/restart them. Traditional MPI calls are transmitted between tasks over the reliable middleware's distributed communication facilities and a program written in conventional MPI can execute over FEMPI with little or no alteration. More information on the fault-tolerance and scalability characteristics of FEMPI can be found in a previous publication [67].

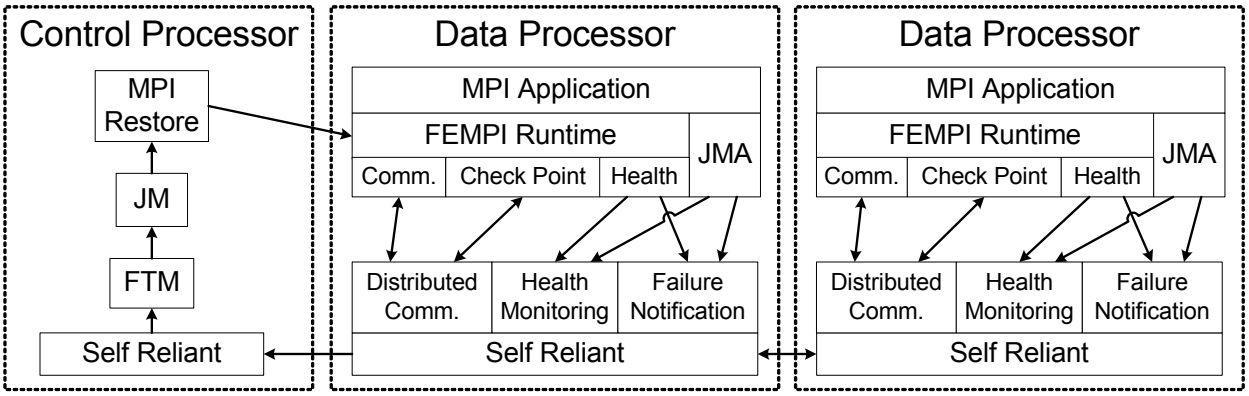


Figure 3-4. Interaction Between FEMPI and Related Software Components of the Dependable Multiprocessor

3.4.6 FPGA Co-Processor Library

Another set of library calls allow users to access Field-Programmable Gate Arrays (FPGAs) to speed up the computation of select application kernels. FPGA coprocessors are one key to achieving high-performance and efficiency in the DM cluster by providing a capability to exploit inherent algorithmic parallelism that often cannot be effectively uncovered with

traditional processors. This approach typically results in a 10 to 100 times improvement in application performance with significant reductions in power [68]. Additionally, FPGAs make the cluster a highly flexible platform, allowing on-demand configuration of hardware to support a variety of application-specific modules such as digital signal processing (DSP) cores, data compression, and vector processors. This overall flexibility allows application designers to adapt the dual-paradigm cluster hardware for a variety of mission-level requirements. The Universal Standard for Unified Reconfigurable Platforms (USURP) framework is being developed by researchers at the University of Florida as a unified solution for multi-platform FPGA development and this API will be brought to bear on the problem of code portability in the DM system [69]. USURP combines a compile-time interface between software and hardware and a run-time communication standard to support FPGA coprocessor functionality within a standard framework [5].

3.4.7 MDS Server and Checkpoint Library

The MDS Server process (shown in Figure 3-3) facilitates all data operations between user applications and the radiation-hardened mass memory. The reliable messaging service is used to reliably transfer data, using its many-to-one and one-to-one communication capabilities. Checkpoint and data requests are serviced on the Mass Data Store in parallel to allow for multiple simultaneous checkpoint or data accesses. The application-side API consists of a basic set of functions that allow data to be transferred to the MDS in a fully transparent fashion. These functions are similar to C-type interfaces and provide a method to write, read, rename, and remove stored checkpoints and other data files. The API also includes a function that assigns each application with a unique name that is used for storing checkpoints for that particular application. This name is generated based upon the name of the application and a unique job identifier and process identifier defined by the central JM when the job is scheduled. Upon

failover or restart of an application, the application may check the MDS for the presence of a specific checkpoint data, use the data if it is available, and complete the interrupted processing. Checkpoint content and frequency is user-directed to reduce system overhead. Other agents within the DM middleware use the same checkpointing facilities available to users to store critical information required for them to be restarted in the event of a failure. Also, the MM uses the user-API compare function when it performs voting on replicated jobs.

3.4.8 ABFT Library

The Algorithm-Based Fault Tolerance (ABFT) library is a collection of mathematical routines that can detect and in some cases correct data faults. Data faults are faults that allow an application to complete, but may produce an incorrect result. The seminal work in ABFT was completed in 1984 by Huang and Abraham [70]. The DM system includes several library functions for use by application developers as a fault-detection mechanism. ABFT operations provide fault tolerance of linear algebraic computations by adding check-sum values in extra rows and columns of the original matrices and then checking these values at the end of the computation. The mathematical relationships of these checksum values to the matrix data is preserved over linear operations. An error is detected by re-computing the checksums and comparing the new values to those in the rows and columns added to the original matrix. If an error is detected, an error code is returned to the calling application. The appeal of ABFT over simple replication is that the additional work that must be undertaken to check operations is of a lower order of magnitude than the operations themselves. For example, the check of an FFT is O(n), whereas the FFT itself is O(nlogn).

In the DM system, ABFT-enabled functions will be available for use by the application developer to perform automated, transparent, low-overhead error checking on linear algebraic computations. In the longer term, it is expected that other, non-algebraic algorithms will

similarly be ABFT-enabled and added to the library. If a data fault is detected via ABFT, a user-directed recovery strategy will be invoked based on the returned error code. A typical response would be to stop the application and restart from checkpointed values.

3.5 Experimental Setup

To investigate the performance of the CARMA-inspired DM middleware, experiments have been conducted on both a system designed to mirror when possible and emulate when necessary the features of the satellite system to be launched in 2009 and also on a typical ground-based cluster of significantly larger size. Beyond providing results for scalability analysis, executing the DM middleware and applications on both types of platforms demonstrates the system's flexibility and the ease with which space scientists may develop their applications on an equivalent platform. The next two sections describe the architecture of these systems.

3.5.1 Prototype Flight System

The prototype system developed for the current phase of the DM project, as shown in Figure 3-5, consists of a collection of COTS Single-Board Computers (SBCs) executing Linux (Monta Vista), a reset controller and power supply for performing power-off resets on a per-node basis, redundant Ethernet switches, and a development workstation acting as the satellite controller. Other operating systems may be used in future (e.g., one of several real-time variants), but Linux provides a rich set of development tools and facilitates from which to leverage. Six SBCs are used to mirror the specified number of data processor boards in the flight experiment (four) and also to emulate the functionality of radiation-hardened components (two) currently under development. Each SBC is comprised of a 650MHz PowerPC processor, memory, and dual 100Mbps Ethernet interfaces. The SBCs are interconnected with two COTS Ethernet switches and Ethernet is used for all system communication. Ethernet is the prevalent network for processing clusters due to its low-cost and relatively high performance and the

packet-switched technology provides distinct advantages for the DM system over bus-based approaches. A Linux workstation emulates the role of the Spacecraft Command and Control Processor, which is responsible for communication with and control of the DM system but is outside the scope of this research.

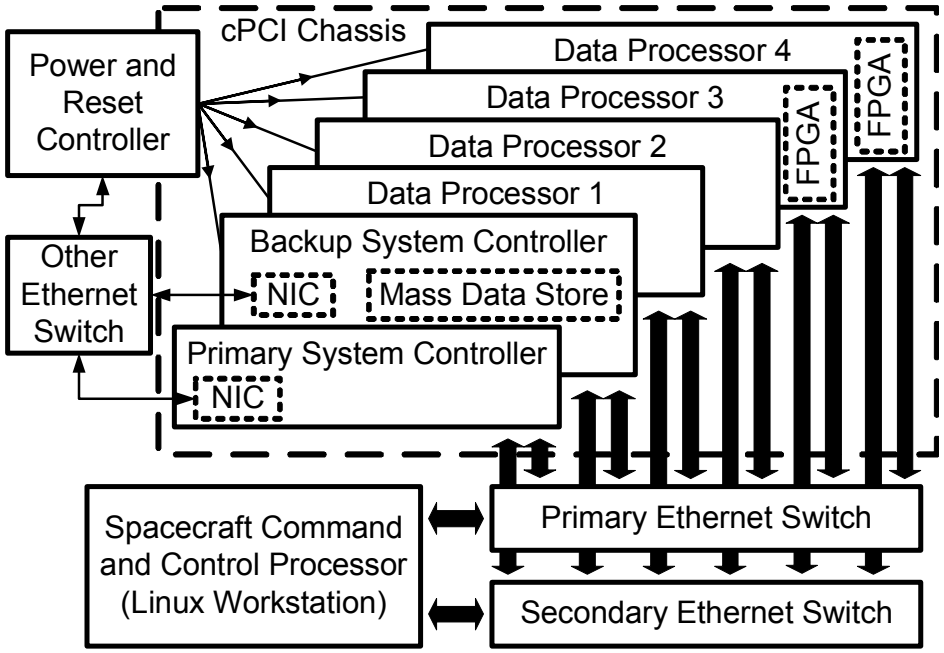


Figure 3-5. System Configuration of the Prototype Testbed

Beyond primary system controller functionality, a single SBC is used to emulate both the backup controller and the MDS. This dual-purpose setup is used due to the rarity of a system controller failure (projected to occur once per year or less frequent) and budget restrictions, but the failover procedure to the backup system controller has been well tested with a dedicated backup system controller (i.e., configuring the testbed with only three data processors). The SBCs are mounted in a Compact PCI chassis for the sole purpose of powering the boards (i.e., the bus is not used for communication of any type). The SBCs are hot-swappable and successful DM system fault tests have been conducted that include removing an SBC from chassis while the system is executing.

Various components (shown as boxes with dotted lines in Figure 3-5) have been included via PCI Mezzanine Card (PMC) slots on the SBCs in order to emulate features that the flight system will require. System controllers in the testbed are connected to a reset controller emulating the power-off reset control system via a third Ethernet interface card to a subsidiary Ethernet network (labeled *Other Ethernet Switch* in Figure 3-5). This connection will likely be implemented as discrete signals in the flight experiment. A number of data processors are equipped with an Alpha Data ADM-XRC-II FPGA card. The flight system will likely require a different type of interface due to the poor conduction cooling and shock resistance characteristics of PMC slots. The MDS memory in the system is currently implemented as a 40GB PMC hard drive, while the flight system will likely include a radiation-hardened, solid-state storage device currently under development.

3.5.2 Ground-based Cluster System

In order to test the scalability of the DM middleware to see how it will perform on future systems beyond the six-node testbed developed to emulate the exact system to be initially flown, the DM system and applications were executed on a cluster of traditional server machines each consisting of 2.4GHz Intel Xeon processors, memory and a Gigabit Ethernet network interface. The embedded version of CARMA and the DM middleware are designed to scale up to 64 data processors and investigations on clusters of these sizes and larger will be conducted once resources become available. As previously described, the fact that the underlying PowerPC processors in the testbed and the Xeons in the cluster have vastly different architectures and instruction sets (e.g., they each use different endian standards) is masked by the fact that the operating system and reliable messaging middleware provide abstract interfaces to the underlying hardware. These abstract interfaces provide a means to ensure portability and these experiments demonstrate that porting applications developed for the DM platform from a

ground-based cluster to the embedded space system is as easy as recompiling on the different platform. As a note, for any experiment on the cluster, a primary system controller and backup system controller configured as the MDS is assumed to be in the system and the total number of nodes reported denotes the number of data processors and does not include these other two nodes.

3.6 System Analysis

Several classes of experiments were undertaken to investigate the performance and scalability of the DM system. An analysis of the DM system's availability and ability to recover from faults was conducted and is presented in the next section. Sections 3.6.2 and 3.6.3 present and analyze the results of a job deployment and a scalability analysis of the DM software middleware respectively. Also, an analysis of the DM system's ability to efficiently schedule applications on the dual-paradigm nodes is presented in Section 3.6.4.

3.6.1 Analysis of the DM System's Fault Recovery Features

In order to provide a rough estimate of the expected availability provided by the DM system in the presence of faults, the time to recover from a failure was measured on the prototype flight system. As defined on the REE project, unavailability for the DM system is the time during which data is lost and cannot be recovered. Therefore, if the DM system suffers a failure yet can successfully recover from the failure and use reserve processing capability to compute all pending sensor and application data within real-time deadlines, the system is operating at 100% availability. Recovery times for components within the DM system have been measured and are presented in Table 3-1. All recovery times measure the time from when a fault is injected in to the application until the time at which an executing application fully recovers. This time includes the failure detection time, fault recovery decision time, the time to undertake the corrective actions, and the time to load the previous state from checkpoint.

Table 3-1. DM Component Recovery Times

Component	Recovery Time (sec)
Application	0.8385
JMA	0.7525
MDS Server	1.3414
SR Messaging Middleware	50.8579
Linux Operating System	52.1489

For the DM system's first mission, it has been estimated that three radiation-induced faults will manifest as software failures per 101-minute orbit. Assuming the worst-case scenario (i.e., each fault manifests as an operating system error) the nominal system uptime is 97.4188%, which again may actually be 100% availability as most applications do not require 100% system utilization. Unfortunately, the definition for system availability is mission-specific and cannot be readily generalized. Missions with other radiation upset rates and application mixes are under consideration. A preliminary analysis of the system using NFTAPE, a fault injection tool [71] that allows faults to be injected into a variety of system locations including CPU registers, memory and the stack, suggests that the likelihood of a fault occurring that manifests in such a manner as to disable a node (i.e., require a Linux OS recovery) is relatively small (7.05%) because most transient memory flips do not manifest in an error. In addition, the development team has also tested hardware faults by removing and then replacing SBCs from the hot-swappable cPCI chassis while applications are running on those system nodes as well as physically unplugging primary and secondary network cables. Also, directed fault injection using NFTAPE has been performed to test the DM system's job replication and ABFT features. Preliminary availability numbers (i.e., assuming a failure occurred when an application required the failed resource) have been determined to be around 99.9963% via these analyses. Any shortcomings discovered during these fault injection tests have been addressed and, after all

repairs, the DM system has successfully recovered from every fault injection test performed to date. The system is still undergoing additional fault-injection analysis.

In order to gauge the performance and scalability of the fault-recovery mechanisms beyond the six-node flight system, the ability of the DM middleware to recover from system faults was also analyzed on the ground-based cluster. The time required for the FTM to recover a failed JMA when the JMAs fail relatively infrequently (i.e., under normal conditions) proved to be constant at roughly 250ms for all cluster sizes investigated. However, an additional set of experiments was conducted on the cluster to test the limits of the service's scalability. In these tests, each JMA in the cluster was forced to fail one second after being deployed on a node and the results of these experiments are presented in Figure 3-6. Also, the FTM and other agent processes are forced to sleep periodically in order to minimize processor utilization. However, if these "sleep" durations are set to be too long, the response times of these agents become significantly delayed causing critical events to suffer performance penalties. A balance between response time and overhead imposed on the system must be struck and the mechanism by which to control this tradeoff are the controlled sleep times configured by the user. The optimal value for these sleep times may vary based on system performance and size so the optimal value for each agent must be determined to optimize system performance. The effects of varying the length of the period during which the FTM is idle, denoted sleep time, to four different values are also shown in Figure 3-6. The results demonstrate that the time for the FTM to recover JMAs under the most stressing conditions possible is still linearly bound as the number of nodes in the system increases. Also, reducing the sleep time to 50ms (and therefore decreasing the FTM's average response time) provides roughly a 13% performance improvement over setting the sleep time to 100ms in the best case (i.e., 20 nodes shown in Figure 3-6). Processor

utilization was found to be constant at roughly 2% for the FTM, 6% for SR and less than 1% for all other agents in each of these experiments, but each value roughly doubled when the same experiment was performed for the 25ms case (not shown). Since the 25ms FTM sleep time experiment provided approximately the same performance as that seen when setting the sleep time to 50ms, the optimal FTM sleep time for both the testbed and cluster is 50ms. Fault scalability experiments in which a node is required to be rebooted were not investigated on the ground-based cluster because the nodes require almost an order of magnitude more time to reboot than do the nodes in the flight system prototype.

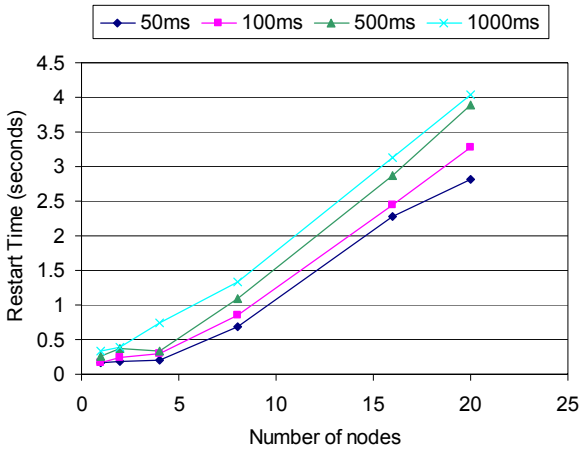


Figure 3-6. JMA Restart Time for Several FTM Sleep Times

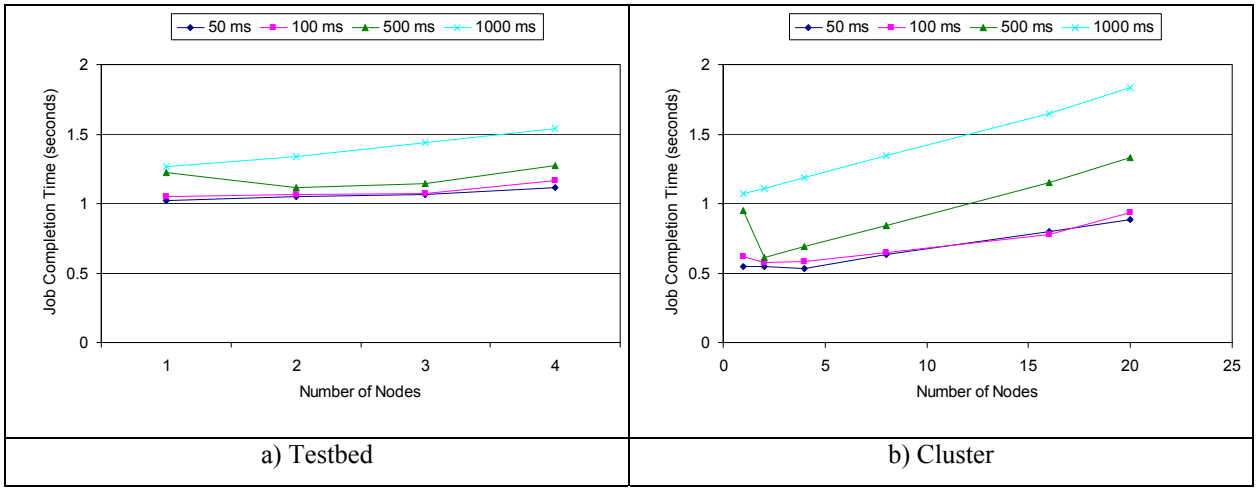
3.6.2 Analysis of the DM Middleware's Job Deployment Features

The DM system has been designed to scale to 64 or more processors and several experiments were conducted to ascertain the scalability of the DM middleware services. Of prime importance is determining the middleware's ability to deploy jobs in a scalable fashion while not imposing a great deal of overhead on the system. As previously described, agents in the DM system are periodically directed to be idle for a given period of time to minimize the burden of the processor on which they execute. To determine the optimal value for other agent sleep times besides the FTM and quantify job deployment overhead, a significant number of jobs

(more than 100 for each experiment) were deployed in the DM system. Since the focus of these experiments was to stress the system, each job did not perform any additional computation (i.e., no "real" work) but only executed the necessary application overhead of registering with the JMA and informing the JMA that the job is completed. Experiments were also conducted with applications that do perform "real" computation and the system's impact in those experiments was found to be less than the results found in these stressful tests because the management service was required to perform significantly fewer functions in a given period of time. Therefore, the operations of scheduling, deploying and removing a completed job from the DM system are independent of the job's execution time.

Figure 3-7 shows the results of the stressing experiments performed on both the testbed and the cluster. Many jobs were placed into the JM's job buffer, and the JM scheduled and deployed these jobs as quickly as system resources would allow. The job completion times shown are measured from the time the job was scheduled by the JM until the time at which the JM was notified by the corresponding JMA that the job completed successfully. The results were averaged over the execution of at least 100 jobs for each experiment and this average was then averaged over multiple independent experiments. To determine how sleep times affect the response time versus imposed system overhead, the JM sleep time was varied in this experiment between 50ms and 1000ms with the JMA sleep times fixed at 100ms. For the testbed system, adjusting the JM sleep time provided at most a 3.4% performance improvement (i.e., agent sleep time of 50ms versus 1000ms for the four-node case). However, since the processor utilization was measured to be constant at 2.5%, 2.3% and 5.3% for the JM, FTM and SR respectively for all experiments, a sleep time of 50ms provides the best performance for the testbed. A sleep time of 25ms was examined and was shown to provide the same performance as that of the 50ms

sleep time but with an additional 50% increase in processor utilization. SR was tuned to provide the optimal performance versus overhead for the system as a whole by adjusting buffer sizes, heartbeat intervals, etc. and then held constant in order to assess the newly developed system components. The overhead imposed on the system by other agents and services (besides SR) was found to be negligible and therefore the results of varying their sleep times are not presented.



a) Testbed b) Cluster

Figure 3-7. Job Completion Times for Various Agent Sleep Times

The results of the job completion time experiments on the ground-based cluster (Figure 3-7b) show the same general trend as is found in the testbed results. Agent sleep times of 50ms and 100ms provide a performance enhancement compared to instances when larger sleep time values are used, especially as the number of nodes scales. The outlying data point for the 500ms sleep time at one node occurs because the time to deploy and recover a job in that case is roughly equal to the JM sleep time. Frequently, the lone job active in the system completes just as the JM goes to sleep and therefore is not detected as complete until the JM becomes active again after a full sleep period. In other instances, one or more nodes are completing jobs at various points in time so the same effect is not observed. However, unlike in the testbed analysis, setting the JM sleep time to 50ms imposes an additional 20% processor utilization as compared to the

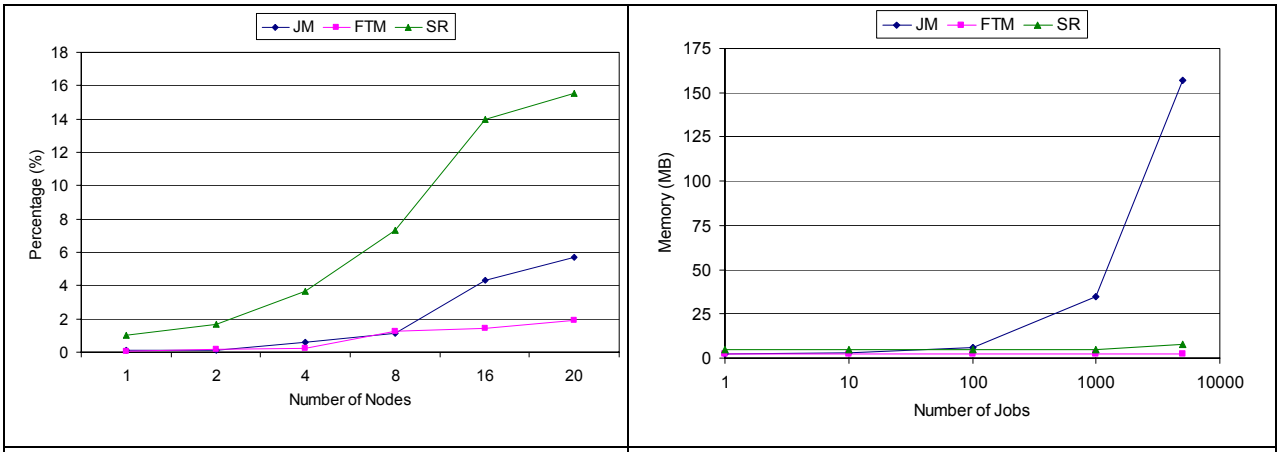
system when set to 100ms, though the overall processor utilization is relatively low for the DM middleware. The results suggest a JM sleep time of 100ms is optimal because the overhead observed when the JM sleep time is above 100ms imposes an overhead roughly equivalent to that observed when set to 100ms. The job completion times measured for the cluster are roughly half that of the values measured in the testbed experiments. This result suggests the cluster results show what can be expected when the testbed is scaled up in the coming years by roughly multiplying each performance value in Figure 3-7b by two. The results demonstrate the scalability of the CARMA-based DM middleware in that the overhead imposed on the system by the DM middleware is linearly bounded.

Figure 3-8a presents the measured processor utilization in the cluster when the agent sleep times are set to 100ms and Figure 3-8b shows system memory utilization. The SR processor utilization incorporates all agent communication and updates to the system information model, operations that could be considered as JM and FTM functions if it were possible to explicitly highlight those operations. Memory utilization was found to only be dependent on the size of the JM job buffer, which is the memory allocated to the JM to keep track of all jobs that are currently pending in the system. Note this buffer space does not limit the total number of jobs the system can execute, just the number that can be executing simultaneously. The memory utilization is scalable in that it is linearly bounded (note the logarithmic scale in Figure 3-8b), but a realistic system would likely require ten to twenty buffer slots and certainly not more than a few hundred. The memory utilization in this region of interest is roughly 4MB and the values presented in Figure 3-8b were the same on both platforms for all experiments.

3.6.3 Performance and Scalability Summary

The optimal system parameter values and middleware performance and scalability results observed in the preceding experiments are summarized in Table 3-2. Also, the projected

performance of the DM middleware on future flight systems incorporating more nodes than provided by our testbed facilities are included in Table 3-2 as well. The projections are based on our current testbed technology and therefore can be considered worst-case performance values because the flight system will incorporate improved versions of all components. Two system sizes were chosen for the projected values to match the planned maximum system size for the first full deployment of the product (i.e., 64 nodes) and to show the system size that saturates the limiting factor for the management service (i.e., control processor utilization).



a) Processor Utilization (Cluster)

b) Memory Utilization (Both Platforms)

Figure 3-8. System Overhead Imposed by the DM Middleware

As previously described, the optimal value for the FTM sleep time was observed to be 50ms for all experiments and this value will likely be optimal for larger flight systems as well because the majority of the FTM's operations are performed via independent, interrupt-based threads (i.e., updating internal tables due to heartbeat and failure notifications) that are largely unaffected by the sleep time. A JM sleep time of 50ms was found to be optimal for the testbed but 100ms was found as the optimal value on the 20-node cluster. This result was due to the commensurate increase in workload (i.e., number of jobs) imposed on the JM as the number of nodes was increased. The projected results suggest 100ms is also the optimal value for larger systems composed of the components found in the 4-node testbed due to an increase in control

processor utilization as node size increases. As previously described, the optimal JMA sleep was found to 100ms for both system types and is independent of node size because JMAs are fully distributed throughout the cluster and therefore inherently scalable. As a result, the optimal JMA sleep time for larger systems is also projected to be 100ms. The processor utilization values for the DM middleware are relatively low for the 4- and 20-node systems. Since the control processor is executing several critical and centralized agents while data processors only execute distributed monitoring agents (i.e., JMAs), the control processor is more heavily utilized for all cases while the data processor utilization is independent of system size. The projected results show the control processor to be moderately loaded for the 64-node system and heavily loaded for a system size of 100 nodes. While these results suggest 100 nodes is the potential upper bound on system size for the DM middleware, two facts must be considered. First, the flight system will feature processors and other components with improved performance characteristics and will therefore extend this number beyond 100 nodes. And second, a "cluster-of-clusters" deployment strategy can be incorporated whereby the cluster is divided into subgroups and managed by a single control processor with a separate control processor managing each subgroup. In this manner, the DM middleware could conceivably scale to hundreds of nodes, if one day the technology were available to provide power for such a cluster in space.

The FTM's ability to recover faulty JMAs on data processor nodes (denoted *JMA Recovery Time* in Table 3-2) was measured to be 250ms for all systems under typical conditions (i.e., a JMA fails once per minute or less frequently) and this value will hold for systems of any size. However, under the extreme case when each JMA in the system fails every second (denoted extreme in Table 3-2), the 20-node system requires an average of 2.7 seconds to repair each faulty JMA and this recovery time was found to increase linearly for the projected flight systems.

However, this extreme case is far more stressing than the actual deployed system would ever encounter because radiation-induced upset rates of approximately one per minute would likely be the most stressing situations expected (recall two or three faults per hour are projected for the orbit selected for the system's first mission). Also, no useful processing would occur in the system at this stressing failure rate so it would be preferable for the system to shut down if it ever encountered this improbable situation. The time required for the JM to deploy a job and clean up upon the job's completion (denoted *Job Deployment Overhead* in Table 3-2) was found to be roughly one second for both the testbed and 20-node cluster with the cluster being slightly faster due to the improved processor technology. The job deployment overhead is projected to have a relatively small increase for the 64 and 100 node systems because the JM performs relatively few operations per job and can service numerous job requests in parallel. The next section examines the middleware's ability to schedule jobs for the dual-paradigm system.

Table 3-2. System Parameter Summary

Parameter	4-Node Testbed	20-Node Cluster	64-Node Flight System (projected)	100-Node Flight System (projected)
Optimal FTM Sleep Time	0.05s	0.05s	0.05s	0.05s
Optimal JM Sleep Time	0.05s	0.1s	0.1s	0.1s
Optimal JMA Sleep Time	0.1s	0.1s	0.1s	0.1s
Control Processor Utilization	10.0%	24.1%	52.25%	95.3%
Data Processor Utilization	3.2%	3.2%	3.2%	3.2%
Control Processor Memory Utilization	6.1MB	6.6MB	6.9MB	7.1MB
Data Processor Memory Utilization	1.3MB	1.3MB	1.3MB	1.3MB
JMA Recovery Time (typical)	0.25s	0.25s	0.25s	0.25s
JMA Recovery Time (extreme)	0.25s	2.70s	9.44s	14.95s
Job Deployment Overhead	1.1s	0.95s	2.05s	2.95s

3.6.4 Scheduling Analysis

The previous sections contain results that illustrate the scalability of the proposed middleware and system design. In addition to providing job and system fault tolerance, the DM middleware uses a few job scheduling techniques commonly used in large-scale cluster systems. Some of the areas include batch scheduling (i.e., applications scheduled to nodes for the duration

of their execution time without preemption), gang scheduling (i.e., applications scheduled in a lock-step fashion where preemption is used to optimize resource utilization), and coscheduling [72] whereby application metrics are observed dynamically at runtime to continuously optimize job deployment. In order to provide a method to optimally utilize system resources with as little overhead as possible, the DM system employs a gang scheduler that uses OLB to quickly deploy parallel and serial applications. Batch scheduling was not chosen, since the inability to preempt applications would waste limited processor cycles, and coscheduling was not selected due to the overhead that would be imposed on the system. Due to the level of checkpointing required in the system to overcome faults due to radiation, parallel preemption is a relatively overhead-free operation (i.e., a simple kill of all parallel tasks). In addition, the heterogeneity of processing resources (e.g., availability of both CPUs and FPGAs) is taken into account when scheduling applications (dictated by a job's description file) to improve the quality of service that the applications observe.

Two realistic mission scenarios have been devised to test the effectiveness of the DM scheduler. The first is a planetary mapping mission that includes three algorithms for high- and low-resolution image processing of a celestial object. An image filtering algorithm continuously performs an edge-detection on a stream of images to provide surface contour information, while a Synthetic Aperture Radar (SAR) kernel [73] periodically processes a data cube representing a wide-swath radar image of the object's surface. When an image buffer is filled above a pre-defined level, a data compression algorithm compresses the images for downlink. The second mission performs object tracking with the same three algorithms used in the mapping mission but with a slight variation that dramatically changes the mission's runtime characteristics. As in the planetary mapping mission, the image filtering algorithm continuously performs an edge

detection of input images. However, the SAR algorithm only executes a detailed radar scan if an object or event of interest is discovered, thus the SAR kernel is non-periodic. The characteristics of these algorithms are summarized in Table 3-3. The execution times, as measured on the testbed previously defined, for each algorithm executing on one or multiple SBCs with or without the use of an FPGA coprocessor are also provided.

Execution times are denoted in terms of the number and type of nodes on which the application executes. For example, *1 SW SBC* denotes the case when the application executes only on the CPU of one SBC, while *1 HW SBC* denotes execution on one SBC using its FPGA coprocessor. Similarly, *2 SW SBCs* defines the case when an application is deployed in a parallel fashion on two nodes without an FPGA. The remaining two designations denote cases where a parallel application executes on a mixture of processor–only and FPGA-accelerated nodes.

Table 3-3. Mission and Algorithm Characteristics

Mission Description	**Scheduling Mode**	**Initial Priority Level**	**Execution Time Measured on Testbed (seconds)**				
			1 SW SBC	1 HW SBC	2 SW SBCs	1SW and 1HW SBC	2SW and 1HW SBC
1) Planetary Mapping							
Image Filtering	Continuous	Medium	10	5	5	4	3
SAR	Periodic	High	600	30	300	25	20
Data Compression	Non-Periodic	Low	50	N/A	25	N/A	N/A
2) Object Tracking							
Image Filtering	Continuous	Medium	10	5	5	4	3
SAR	Non-Periodic	High	600	30	300	25	20
Data Compression	Non-Periodic	Low	50	N/A	25	N/A	N/A

Multi-SBC versions of the image filtering algorithm are decomposed in a coarse, data-parallel manner such that each process is independently filtering one image at a time from a stream of images. In this manner, the total number of images to filter per processing interval is evenly distributed across the coordinating SBCs, although the total number of images to process varies each time that the application executes due to the time delay between successive deployments. For clarity, execution times denoted in the table for image filtering are the

effective execution time required per image. SAR is decomposed across multiple SBCs in a parallel fashion with all SBCs coordinating to complete a single iteration of the fixed-size SAR data set. Data compression is decomposed across up to two SBCs in a distributed manner where the SBCs evenly divide the fixed-size data set then compress their portions independently. An FPGA-accelerated version of the compression algorithm was not implemented, as noted with "N/A" in the table.

Several experiments were conducted to analyze the ability of the DM scheduler to effectively meet mission deadlines and to assess each scheme's impact on performance and buffer utilization. Data Ingress defines the rate at which data (i.e., images) enter the processing system from external sensors. Images are fixed at 2.6MB each in the experiments. For Mission 1, SAR processes 50 images worth of data is therefore periodically scheduled every time that 50 images enter the system. However, the non-periodic version of SAR deployed in Mission 2 is only executed when an object of event of interest is detected, and this detection event is emulated as a Poisson distribution with a mean equal to that of the periodicity of the SAR process used in Mission 1 for a fair comparison between results. Two scheduling improvements are considered, including use of application priorities in scheduling decisions, denoted *Priority*, and use of preemption mechanisms to remove one or more currently executing job to provide additional resources to a higher-priority application. These two enhancements are compared to a Baseline approach with neither of these features.

The priority levels noted in Table 3-3 are based upon the criticality of the application to the overall mission. However, the DM scheduler applies a best-effort approach to quality of service whereby if an application reaches a critical threshold then its priority may be increased. For example, the image filtering application becomes high priority if the number of images to be

filtered is double its allocated buffer size (i.e., 100 images), and the compression algorithm becomes medium or high priority if the data to be compressed is double (i.e., 200 images) or triple (i.e., 300 images) its allocated buffer size, respectively. Preemption occurs when an application with a higher priority has a full input buffer and preempting a lower priority application will provide the necessary system resources. Applications with equal priority levels cannot preempt each other but, in the case when two applications of equal priority are simultaneously available for deployment, the application that completed most recently will not be chosen.

3.6.4.1 Mission 1: Planetary mapping

In the experiments, real-time deadlines are treated as a dependent variable, thus providing a measure of the ability of one scheme versus another. The actual real-time deadline range for the applications highlighted in these experiments vary from 0.5 to 100 seconds depending upon the criticality of mission and other system-specific parameters. The ability of the DM scheduler to meet the real-time deadline requirements of the planetary mapping mission is illustrated in Figure 3-9. The *Minimum Real-Time Deadline* is a measure of the maximum amount of time that the SAR algorithm required to execute, measured from the time its processing period began until the time when the output was written to the data compression buffer. The results show that the Baseline approach is very poor at meeting the real-time deadline requirement due to an inefficient use of processing resources. The Priority scheme provides a marked improvement over the Baseline system and the Preemption provides additional improvement. By contrast, the results in Figure 3-10 show that the Preemption scheme requires a larger buffer to be deployed in the system as data ingress increases, due to resource contention. The Priority scheme requires the system to deploy less buffer space when data ingress increases because applications are scheduled more efficiently without resource contention due to thrashing.

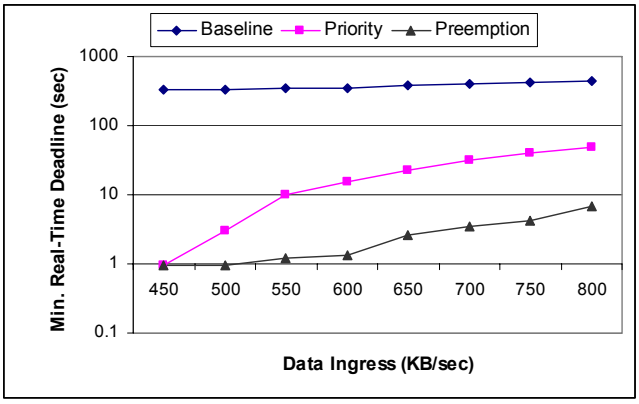


Figure 3-9. Minimum Real-Time Deadline vs. Data Ingress

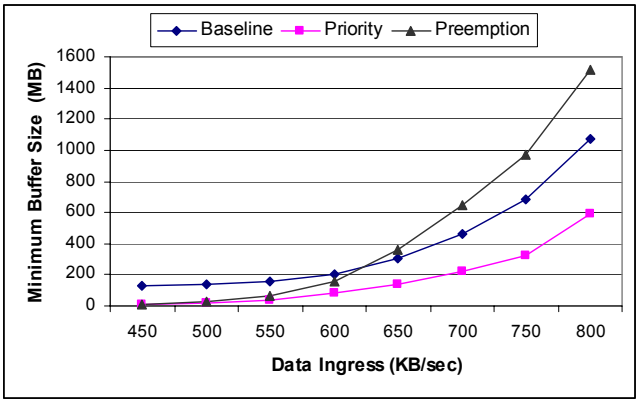


Figure 3-10. Buffer Utilization vs. Data Ingress

Overall mission performance results are provided in terms of throughput in Figure 3-11. The Preemption strategy provides the best performance up to 650KB/sec of ingress rate, but shows a continual decrease in performance due to thrashing. Essentially, the compression algorithm's buffer quickly fills up above this rate and becomes elevated to high priority in order to make room for more output data. The Priority scheme does not suffer the same thrashing effects because preemption is not permitted, so the performance of this scheme matches the Baseline above the critical point. Above the ingress rate of 650KB/sec, overhead due to the DM system middleware (i.e., monitoring, deploying, checkpointing, etc.) begins to affect the execution of Mission 1. Future improvements to the system will allow greater data ingress rates to be achieved without penalty.

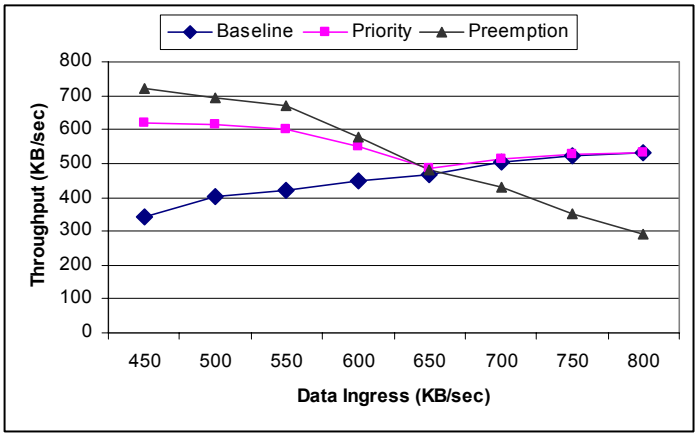


Figure 3-11. Data Throughput vs. Data Ingress

These results suggest that the Preemption strategy provides performance improvements for critical applications with a periodic deployment interval to a point at which the performance of the overall system is reduced to keep meeting the critical application's deadline. The Priority scheme provides an improvement over the Baseline scheme in all cases and mitigates the resource thrashing problem that degrades performance of the Preemption method.

3.6.4.2 Mission 2: Object tracking

In contrast to Mission 1, the ability of the DM scheduler to meet the real-time deadline requirements of the object tracking mission, one which contains a non-periodic application, is illustrated in Figure 3-12. The results show that the Baseline approach is also very poor at meeting the SAR real-time deadline requirement due to an inefficient use of processing resources. The Priority scheme still provides a marked improvement over the Baseline system but, in contrast to the results from Mission 1, the ability to meet the real-time deadline is somewhat reduced due to the unpredictable nature of when the SAR application will be executed. Without the ability of the Priority scheme to give adequate precedence, the SAR algorithm spends more time waiting in the job queue for available resources. This problem is averted by allowing SAR to displace already deployed applications in the Preempt scheme.

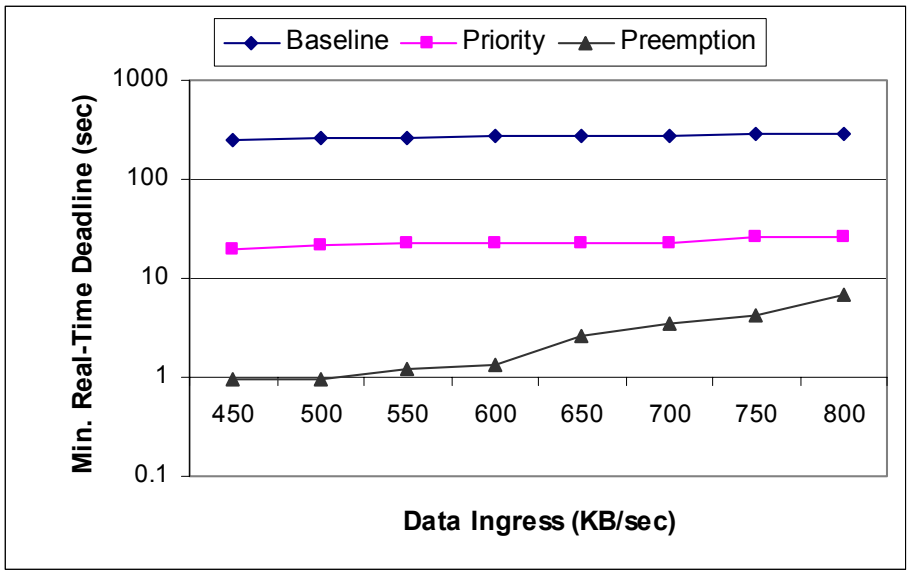


Figure 3-12. Minimum Real-Time Deadline vs. Data Ingress

The memory requirements for each scheme, presented in Figure 3-13, are dramatically reduced as compared to the results observed in Mission 1, because the SAR application is more randomly distributed, thereby reducing and amortizing the likelihood of contention. The memory requirements of the Priority scheme have increased relative to the Baseline scheme due to the increased queuing delay that SAR observes as previously described.

The results in Figure 3-14 highlight overall performance for Mission 2. The Preemption strategy again provides the best performance up to about 640KB/sec and still shows a continual decrease in performance due to thrashing when the compression algorithm's buffer fills. The Priority scheme does not schedule as efficiently as the data ingress rate increases and therefore no longer tracks the performance of the Baseline scheme. In fact, the Baseline scheme demonstrates a slight improvement over the results from Mission 1, since the simple first-in first-out approach outperforms the two contrived schemes. However, overall system throughput improvement is not advantageous since the minimum real-time deadline that the SAR algorithm can expect under the Baseline scheme is in hundreds of seconds.

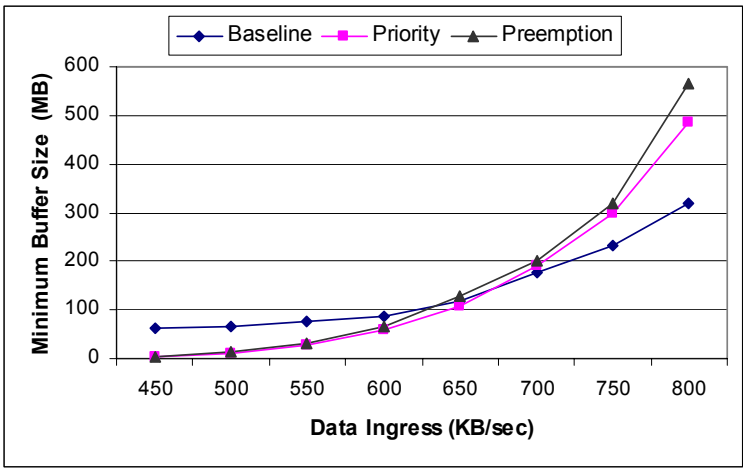


Figure 3-13. Buffer Utilization vs. Data Ingress

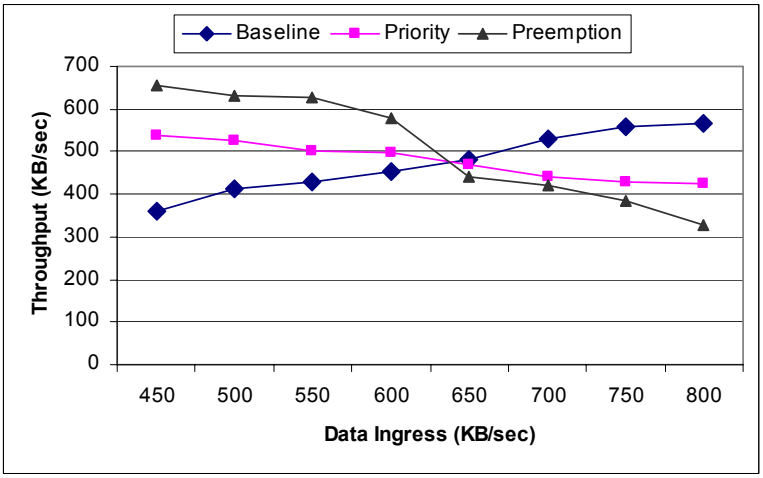


Figure 3-14. Data Throughput vs. Data Ingress

3.7 Conclusions

NASA's strategic plans for space exploration present significant challenges to space computer developers and traditional computing methods as architectures developed for space missions fall short of the requirements for next-generation spacecraft. The Dependable Multiprocessor (DM) technology addresses this need and provides the foundation for future space processors. The DM is an integrated parallel computing system that addresses all of the essential functions of a cluster computer for spacecraft payload processing. The system will provide the ability to rapidly infuse future commercial technology into the standard COTS

payload, a standard development and runtime environment familiar to scientific application developers, and a robust management service to overcome the radiation-susceptibility of COTS components, among other features. A detailed description of the DM middleware was presented and several experiments were conducted to analyze improvements in the prototype DM middleware and architecture.

An investigation of the system availability was undertaken and showed the system provides an acceptable level of availability for the first proposed space mission. Also, the system's fault tolerance capabilities were demonstrated to scale linearly with the number of nodes even when the number of node failures was far greater than a typical system would experience. It was determined that decreasing the FTM's sleep time provides a modest performance improvement in recovering from node failures. In addition, the development team has also tested hardware faults by removing and then replacing SBCs from the hot-swappable cPCI chassis while applications are running on those system nodes as well as physically unplugging primary and secondary network cables. Directed fault injection using NFTAPE has been performed to test the DM system's job replication and ABFT features. Nominal availability numbers (i.e., assuming a failure occurred when an application required the failed resource) of around 99.9963% have been determined to be a representative value via these analyses. Any shortcomings discovered during these fault injection tests have been addressed and, after all repairs, the DM system has successfully recovered from every fault injection test performed to date. A broader fault-tolerance and availability analysis is also underway.

Performance and scalability of CARMA's job deployment, monitoring and recovery system were analyzed on a prototype testbed that emulates the flight system and a ground-based cluster with many more nodes. For the testbed system, adjusting the agent sleep times provided

at most a 3.4% performance improvement (i.e., agent sleep time of 50ms versus 1000ms for the four-node case). Since the processor utilization was measured to be constant at 2.5%, 2.3% and 5.3% for the JM, FTM and SR respectively for all experiments, a sleep time of 50ms provides the best performance for the testbed. For the ground-based cluster, agent sleep times of 50ms and 100ms provide a performance enhancement compared to instances when larger sleep time values are used, especially as the number of nodes scales. However, unlike in the testbed analysis, setting the agent sleep times to 50ms imposes an additional 20% processor utilization as compared to the system when set to 100ms. A balance between response time and overhead imposed on the system must be struck and the results suggest an agent sleep time of 50ms and 100ms for the FTM and JM respectively is optimal for large-scale systems. The results demonstrate the scalability of the DM middleware to 100 nodes and beyond with the overhead imposed on the system by the DM middleware scaling linearly with system size.

Several experiments were undertaken to analyze the ability of the DM scheduler to meet real-time mission deadlines for applications in earth or space science. The results show that the Baseline approach, an OLB-based gang scheduler, provides good overall system performance for missions having applications with random execution patterns, but cannot meet typical real-time deadline requirements of either of the case-study missions. The Priority scheme is the best overall method for missions that incorporate applications with periodic execution patterns, but it suffers a performance reduction when scheduling applications with random execution patterns. The Preemption strategy provides performance improvements for critical applications with a periodic deployment interval to a point at which the performance of the overall system is reduced, since resource thrashing ensues as the scheduler attempts to meet the most critical

deadlines. Also, the Preemption strategy requires the system to deploy the most memory buffer resources.

The DM management system design and implementation has been examined and several system architecture tradeoffs have been analyzed to determine how best to deploy the system for the NMP flight design to be launched in 2009. The results presented in this paper demonstrate the validity of the CARMA-enhanced DM system and show the management service to be scalable and provides an adequate level of performance and fault tolerance with minimal overhead. A number of scholarly contributions for the first phase of this dissertation include the development of the first comprehensive framework to date for job and resource management and fault-tolerance services tailored for a dual-paradigm embedded aerospace system. This version of CARMA provides additional insight to refine the framework's design to make it more fault-tolerant, robust, flexible and portable while still functioning within the strict limitations of the embedded environment. Additional features developed to further improve the fault tolerance and scalability of the DM system were investigated and are presented in the next chapter.

CHAPTER 4
AUTONOMIC METASCHEDULER FOR EMBEDDED SYSTEMS

As NASA and other agencies continue to undertake ever challenging remote sensing missions, the ability of satellites and space probes to diagnose and autonomously recover from faults will be paramount. In addition, a more pronounced use of radiation-susceptible components in order to reduce cost makes the challenge of ensuring system dependability even more difficult. As discussed in the last chapter, the Dependable Multiprocessor is being developed by Honeywell Inc. and the University of Florida for an upcoming NASA New Millennium Program mission to meet these and other needs. Among other features, the platform deploys an autonomic software management system to increase system dependability and this chapter focuses on these features. In addition, one CARMA component in particular, namely the mission manager, has been investigated and developed to provide an autonomous means to adapt to environmental conditions and system failures. This section provides a detailed analysis of the management system philosophy with a focus on the adaptable mission manager. A variety of case studies are presented that highlight the dependability and performance improvement provided by the mission manager and autonomic health monitoring system deployed in realistic planetary orbits.

4.1 Introduction

Robust, autonomous and dependable onboard computational capabilities are of chief interest to mission planners and design engineers building space probes and exploration vehicles. Missing an important event in the course of a mission due to a failed component, control delays from ground station uplinks or an unacceptable level of data integrity can render a vast investment in time and resources useless. The delay from problem identification to correction can be so pronounced that the capability of future systems will be drastically reduced if humans

remain in the decision loop. As systems grow in capability and therefore complexity the need for autonomous decision making capabilities is more pronounced. Also, with critical decisions carried out in an autonomous manner, the onboard systems and capabilities that perform resource partitioning and adaptive fault tolerance decisions among other features must themselves be robust and dependable. Therefore, providing a hierarchical and distributed approach that can suffer systemic failures without incapacitating the entire system is also required.

As previously described, NASA's New Millennium Program (NMP) office has commissioned the development of an embedded cluster of Commercial Off-The-Shelf (COTS) processors for space missions [52]. The Dependable Multiprocessor (DM) technology provides numerous benefits, and the key benefit highlighted in this section is CARMA's ability to detect and autonomously overcome the Single-Event Upset (SEU) radiation susceptibility of COTS components while minimizing the performance impact of the employed Software-Implemented Fault Tolerance or SIFT fault mitigation technique. Also, by deploying an autonomic information gathering extension to the embedded version of CARMA, the DM ensures the dependability and scalability of its middleware. In addition to being robust, the DM middleware and SIFT techniques are highly generalizable and can therefore be tailored to meet the needs of almost any mission. Using a SIFT-based mitigation technique on a commodity platform that is not custom-developed for a single platform is a relatively novel way to address SEU mitigation of COTS components in space systems, and this research will help to analyze the efficacy of this approach.

The remainder of this chapter examines the mission management features of the DM system's middleware and is divided as follows. Section 4.2 provides a background of related research with Section 4.3 describing the new components added to the DM management system

not included in the previous chapter. Several experimental case studies highlighting the system's adaptable and autonomous fault-tolerance features are executed on the testbed system previously described and Section 4.4 defines the experimental setup with results presented in Section 4.5. Conclusions and future work are highlighted in Section 4.6.

4.2 Related Research

Autonomic computing borrows strategies from biological systems to enable computing systems to self-manage. By first examining the current state of their system and environment via probes, autonomic systems then independently take action to optimize the system per predefined policies. The primary objective of this paradigm is to empower complex systems with a rich set of information-gathering probes and decision-making autonomy to mitigate the relative inefficiency of the human-machine interface. A more in-depth background of autonomic computing is not included here as the topic has been covered by numerous sources such as Parashar and Hariri [74] and Sterritt, et al. [75]. Autonomic computing has developed to address the need to control large-scale geographically distributed computing systems such as telecommunication networks [76] and distributed database systems [77-79]. However, applying such techniques and lessons learned to similar challenges found in designing embedded space systems has the potential to provide numerous benefits.

Some recent projects have taken a first step towards removing human interaction from satellite control systems by providing an adaptable management service to autonomously transfer and route commands and data between spacecraft and distributed ground stations. For example, through the Integrated Test and Operations System (ITOS) developed at the NASA Goddard Space Flight Center, a communication infrastructure has been deployed to adaptively distribute mission data as needed between spacecraft and distributed ground-based facilities. The ITOS system links command and control facilities, science data processing resources and other

locations housing consumers of satellite data [80]. While the ITOS network provides a degree of autonomy, the system still requires user interaction and does not provide any onboard processing capability relegating the spacecraft to a passive role.

In order to reduce or eliminate the inherent delay that limits the performance of remote systems as well as mitigate other mission factors such as risk and cost, spacecraft must be empowered to make decisions in an autonomic fashion. Sterritt, et al. describes four distinctive properties that autonomic space systems must exhibit to prove successful [81]. The so called "self-* properties" of autonomic systems (i.e., self-configuring, self-monitoring, self-protecting, self-optimizing) provide a good metric to which systems should strive. Proposed projects have attempted to incorporate these properties and a few have been compared qualitatively by Rouff, et al. [82]. For example, NASA's Deep Space One (DS1) mission launched in October of 1998 included a beacon monitoring system allowing the spacecraft to self-monitor and apprise ground crews of its status autonomously thereby reducing the load on communication systems [83]. The DS1 mission was an important first step in aiding future deep space missions but only exhibited the most basic of the self-* properties.

The Autonomous Nano-Technology Swarm (ANTS) mission [84] seeks to employ autonomic techniques within a thousand or more member "swarm" of small spacecraft that will collectively explore the asteroid belt between Mars and Jupiter and is set to launch in the 2020 to 2030 timeframe. While many of the design details are unclear because the project is still in the planning stages, the system will likely consist of various probes with individualized specialties (e.g., X-ray sensors) grouped into teams in order to collectively explore asteroids and report back status to the rest of the swarm. The designers liken the system to an ant colony whereby each member makes decisions individually on how best to complete the objective (i.e., to explore the

asteroid belt for this mission). The mission planners see an autonomic system as a base requirement for success due to the remote distance and complex nature of the mission.

While future concept systems are needed to provide a vision to which research can strive, other missions have also incorporated autonomic management functionality to meet near-term objectives. It has been widely accepted that an agent-based philosophy is an appropriate means by which to reduce the design complexity of distributed real-time systems and yet not limit the system flexibility. For example, the Autonomic Cluster Management System (ACMS) concept deploys a system of agents that work collectively to execute applications [85]. The system includes a centralized and replicated configuration agent that deploys agents and applications throughout the system. Also, general agents exist on distributed nodes to execute applications and a central optimization agent analyses the system to ensure optimal system utilization. A performance and scalability analysis, albeit limited, of ACMS shows the concept to be a good step towards autonomous cluster system administration. In addition to performance and scalability improvements, autonomic computing has also been applied to ground and space systems to improve system dependability [86].

Insight was gleaned from these and other sources and the system is also designed to address some of the perceived shortcomings in previous research. First and foremost the DM middleware is designed to be autonomic and dependable, and the manner in which it addresses the four self-* properties among other features is further elaborated upon in Section 4.3. Beyond meeting those requirements, the DM middleware also addresses at least two additional key issues lacking in previous research. Experimental analysis of the system is conducted with adequate work loads and meaningful applications. Such experiments are paramount to effectively analyze the system. Also, experiments are being conducted on a prototype system that is equivalent to

the specified flight system as described in the previous chapter to provide a high degree of realism to the analysis.

In addition, the concept of providing autonomic features to satellite systems is not novel in and of itself but providing a framework by which mission and application designers can tailor the system to meet their specific needs has not been addressed. Too often concepts are too vague and general or alternatively too custom and select to be adequately applied to specific missions undertaken by a broad category of users. The DM middleware provides a general-purpose yet comprehensive mission management system that can also be easily customized to meet the needs of specific missions. The next section provides a detailed description of the DM middleware's autonomic features.

4.3 Mission Manager Description

The DM system infrastructure is composed of a number of software agents that coordinate to deploy and recover applications as previously described. A description of how applications are deployed in a dependable and autonomous fashion by the DM system follows. Detailed descriptions of relevant components and their interactions are provided with special emphasis on the Mission Manager (MM) and other autonomic computing extensions.

The Mission Manager forms the central decision-making authority in the DM system. The MM and several other primary components are deployed on the radiation-hardened control processor for increased system dependability. The three main tasks the MM performs to effectively administer the system and ensure mission success are 1) deploy applications per predetermined mission policies, 2) collect health information autonomically on all aspects of the system and environment to make appropriate runtime decisions, and 3) adjust mission policies autonomously per observations to optimize system performance and dependability. Detailed descriptions of each main MM task follow.

4.3.1 Application Scheduling and Deployment

To deploy applications based on mission policies, the MM relies upon a collection of input files developed by users to describe the policies to be enforced for the mission at hand. Two types of files, namely mission descriptions and job descriptions, are used to describe policies at the mission and application level, respectively. A mission is defined as the collection of all user applications that will execute in the system and the mission file explains to the MM how best to execute that mission. The mission description file includes policy information such as which applications to execute, how often to execute them, what triggers the execution of a particular application, what prioritization ordering to follow, what steps to take to improve application fault tolerance, etc. The MM loads the mission file on boot and mission files can be reloaded in-situ if mission applications change. The full set of parameters defined in the mission description file is listed Table 4-1.

A job is defined as a particular instance of an application deployed with a particular set of input data, replication scheme, etc. A job is composed of one or more tasks, each of which being represented by an independent executable most often executing on a separate processor. For jobs that include multiple tasks, MPI is used for inter-process communication between the tasks. The job description file describes all information required to execute a job including relevant files required to execute the job and policies such as which and how many resources are required to execute the job and how to recover from a failed task. The full set of possible parameters defined in the job description file is listed and described in Table 4-2.

Applications are scheduled and deployed according to priority, real-time deadline and type. Preference is given to applications with higher priority and an application's priority can be adjusted at runtime by the MM as dictated by a number of factors. For example, if an application is close to missing a real-time deadline then its priority is elevated so as to help ensure the

application will meet the deadline. In addition, critical applications with real-time deadlines may preempt other lower priority applications. In order to remove the possibility of starvation, the priorities of applications that have not executed in a set timeout period and are not triggered by an event are elevated. Also, the priorities of triggered applications that have received their trigger yet have not executed after a set timeout period are also elevated. Beyond priority, applications are deployed based on their type as previously described in Table 4-1.

Table 4-1. Mission Description File Format

File Entry	Description
Mission Name	Descriptive name to denote mission.
Number of Applications	Number of applications in this mission.
The following is included for each application	
Name	Descriptive name to denote application.
Job Description	File name of the job description that describes this application.
Real-time Deadline	Deadline by which the application must complete successfully. Used by the MM to adjust priorities and remove jobs that do not complete in time.
Base Priority	The initial priority level for the application. The MM may promote or demote applications as real-time deadlines require.
Application Type	Four types are currently defined and include *continuous*, *periodic*, *triggered* and *follower*. *Triggered* implies the app. requires a specific event trigger to begin (e.g., buffer full, input image ready) and *follower* implies the application will execute after another specific application completes.
Data Trigger	Defines event that will trigger the need to begin execution of an aperiodic application.
Periodicity	Defines periodicity of periodic applications.
Application To Follow	Defines the application whose completion marks the time to begin executing the specific follow application.
Follower Type	Applications may be defined to always execute no matter what the conditions or to only execute if enough time remains in the initial application's real-time deadline.
Base Replication Type	The initial type of replication suggested by the application developer is defined one of three options including *none*, *spatial* and *temporal*. The spatial policy defines the need to replicate an application physically across processors while the temporal policy requires the application to be replicated in time on the same processor. The MM may choose to adjust the replication type for an application based on environmental radiation levels and system load to best meet the mission's dependability and performance constraints.
Base Number Replicas	The initial number of replicas to deploy. The MM may choose to adjust the number of replicas to meet the mission's dependability and performance constraints.
Recovery Action	Recovery action to take in the event of an unrecoverable application failure. The defined policies include *rollback* in which case the application is restarted from its last checkpoint and *abort* in which case the current run of the application including all of its input data are removed from the system and the application is restarted based on its defined type (e.g., periodic)

Table 4-2. Job Description File Format

File Entry	Description
Job Name	Descriptive name to denote job.
Number of Tasks	Number of tasks in this job.
Recovery Policy	Recovery action to be taken by the JM in the event of an unrecoverable task failure. The defined policies include *abort* in which case the job is removed from the system, *rebuild* in which case the JM recovers the task from the last stable checkpoint, *ignore* is used for MPI applications in which case the JM instructs other tasks to continue execution with one less task, and *reduce* is used for MPI applications in which case the JM instructs other tasks of the job to redistribute their data based on having one less task.
Times to Recover	Number of times to attempt a recovery before an abort is declared. Recovery policies may be nested (e.g., reduce for N times, then recover for N times, then abort) and this parameter defines how many times to perform each scheme.
	The following is included for each task
Task Name	Descriptive name to denote task.
Execution Path	Linux file path to the task executable.
Task Executable	The binary file to be executed.
FPGA Requirements	Described whether this task *requires*, *prefers* or does not require an FPGA for execution. This information is used by the JM to make job-level scheduling decisions.
Bit File Name	File required to program an FPGA for this task's specific requirements.
Heartbeat Timeout	Time necessary to wait between heartbeats in order to declare the task failed.

In addition to scheduling applications, the MM also autonomously chooses which replication strategy is best suited for each execution of a particular application based on runtime environmental data and system status. The type of replication deployed (i.e., spatial or temporal) is chosen based on a number of factors including the type of application, the status of available resources and outstanding real-time deadlines of other applications. For example, if the system is currently lightly loaded, the MM may elect to deploy applications with spatial replication in order to devote more resources to completing all replicas of a particular application more quickly. Alternatively, applications would likely be deployed with temporal replication when the system is more heavily loaded in order to complete at least one execution of each application as quickly as possible. Of course, application priority also affects the decision of which applications are replicated spatially or temporally. It should be noted that the MM performs the vote comparison of output data from application replicas, making the service user transparent.

Environmental factors also affect replication decisions. The DM system is equipped with radiation sensors that inform the MM of the likelihood that a fault will occur at any point in time due to radiation effects. The MM adjusts the number and type of application replicas it chooses to deploy in the system based on this information. If the environment is more hostile than normal and faults are likely, the MM may deploy additional replicas to ensure a necessary level of application dependability. For example, if a particular application was routinely deployed without replication, the MM would likely deploy the application with at least two replicas to at least ensure the event of a likely failure could be detected. Critical applications would be deployed in a triple-replication mode in environments with a high level of radiation but the ability to autonomously adjust the degree of redundancy of an application can improve the overall system performance while still maintaining an appropriate level of dependability. The ability to selectively deploy replicas autonomously to improve performance is a key advantage provided by the MM and case studies illustrating the advantages of adaptable replication are highlighted in Section 4.5.

The chain of control by which applications are deployed in the system is illustrated in Figure 4-1. As previously described, the MM schedules applications based on a number of observed runtime factors and policies defined in the mission description file. Upon reaching a decision to deploy an application, the MM sends a request to execute a particular job to the Job Manager (JM). The JM schedules, deploys, manages, recovers and reports status of jobs currently in its job buffer. Job scheduling is performed in an Opportunistic-Load Balancing (OLB) scheme in which jobs in the buffer are assigned to any available resources. The JM schedules multi-task jobs using a gang scheduling method in which the entire job is not scheduled unless all tasks within the job can be simultaneously deployed.

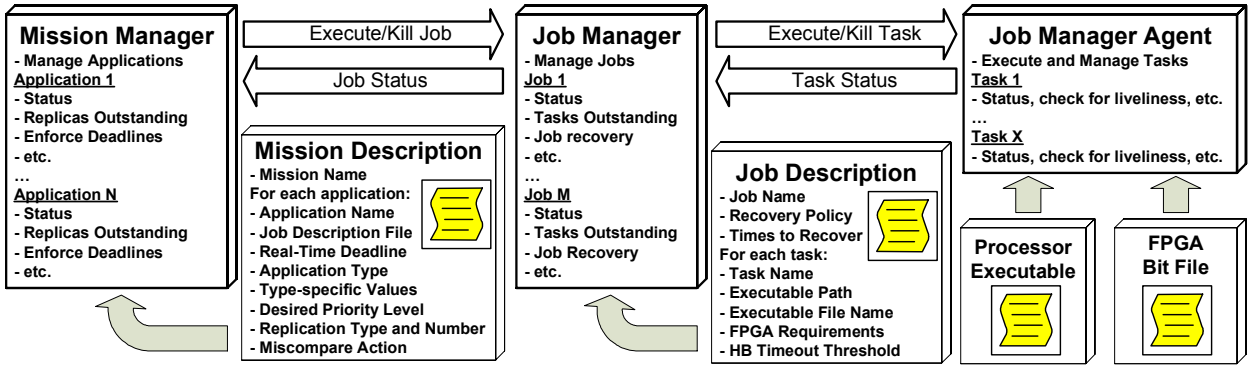


Figure 4-1. Job deployment and management control flow diagram

The MM enforces mission-level policies such as priority, preemption, real-time deadlines, etc. mitigating the need to have a complex scheduling scheme at the job level. Upon reaching a decision to deploy a job, the JM sends a request to execute a particular task to the Job Manager Agent (JMA). The JMA deploys, manages, and reports status of tasks currently in its task buffer. The JMA's chief role is to be the remote agent in charge of forking and executing user executables and monitoring their health locally via heartbeats and progress counters. One JMA is deployed on each data processor in the system, and the distributed nature of the JMAs is intended to improve system scalability by reducing the monitoring burden placed upon the JM.

4.3.2 Autonomic Information Collection System

In order for the MM to make informed decisions, numerous sensors and software agents throughout the system collect and disseminate pertinent information over the reliable communication middleware as shown in Figure 4-2. JMAs deployed on data processors provide health status of tasks that have been assigned to them. In addition, JMAs provide a node status to the Fault-Tolerance Manager (FTM), the central agent that monitors the health of system components. It is the FTM that ensures the healthy deployment of agents and detects, diagnoses and recovers from any systemic fault that may occur. Agents are deployed in the system based on the node type, where the FTM starts up the JM and MM on the radiation-hardened processor

and JMAs on data processors thus providing a measure of self-configurability. The FTM employs a combination of software and hardware probes to passively discern the state of the system but also actively launches diagnostic routines to gain further insight if an anomaly should occur. The FTM is responsible for detecting and self-healing any systemic faults by either restarting agents or power-cycling hardware resources.

Once the system health information is distilled, the FTM relays an update to the MM and JM to aid in scheduling decisions and to ensure those two agents deploy applications, jobs and tasks in a dependable manner. The MM and JM will autonomously "route around" any permanent system fault of data processors, thus providing a measure of self-healing ability at the job management level. It should be noted that the reliable communication middleware handles redundancy at the network level by means of typical error recovery features provided by packet-switched interconnects (e.g., CRC and TCP).

Sensors are deployed strategically throughout the DM system to detect radiation levels in the current environment through which the spacecraft is passing. Environment data is collected and reported to the MM for determining the likelihood of a fault to aid in deciding what degree of replication to deploy to ensure the correct level of dependability is maintained per mission policies. Also, a history of radiation levels is maintained for planetary orbiting missions to predict times during which the system will be exposed to high radiation levels at particular distances in the orbit such as the South Atlantic Anomaly or planetary poles on Earth. A priori knowledge of radiation levels improves the MM's ability to effectively deploy applications with the optimal level of replication, and case studies showing these advantages are highlighted in Section 4.5. This feature affords the system a high degree of self-optimization and self-protection that otherwise would not be possible.

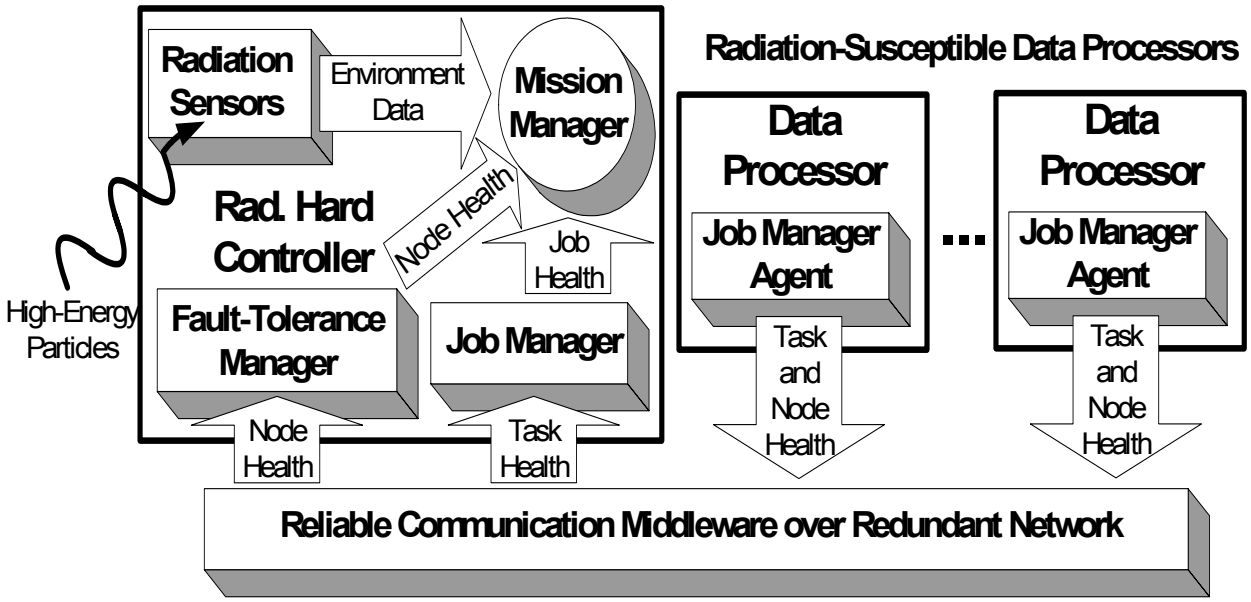


Figure 4-2. Environmental and system health and status information collection infrastructure diagram

4.4 Experimental Setup

The embedded testbed and ground cluster described in the previous chapter were used to evaluate the autonomic features of the DM middleware. A realistic planetary mapping mission scenario has been devised to test the effectiveness of the MM's ability to deploy application replications while optimizing for performance and dependability. The mission includes three algorithms for high- and low-resolution image processing of a celestial object. An image filtering algorithm continuously performs edge-detection on a stream of images to provide surface contour information, while a Synthetic Aperture Radar (SAR) kernel [69] periodically processes a data cube representing a wide-swath radar image of the object's surface. When an image buffer is filled above a pre-defined level, a data compression algorithm compresses the images for downlink. The characteristics of these algorithms are summarized in Table 4-3. The execution times (in seconds), as measured on the testbed previously defined, for each algorithm executing with double- or triple-modular redundancy without the presence of faults are provided.

Execution times were measured for the mission computing two different data sizes. One version denoted Large Data Set in Table 4-3 processes a data cube equivalent to 100 images (i.e., 250 MB of data) and the other version labeled Small Data Set computes a data cube half that size.

Table 4-3. Application Execution Times for Various Replication Schemes

Large Data Set (i.e., 100 Images per Execution)				
Application	2S	3S	2T	3T
Image Filter	5 sec	10 sec	3 sec	3 sec
SAR	300 sec	600 sec	200 sec	200 sec
Compression	25 sec	50 sec	15 sec	15 sec
Small Data Set (i.e., 50 Images per Execution)				
Application	2S	3S	2T	3T
Image Filter	4 sec	8 sec	3 sec	3 sec
SAR	202 sec	377 sec	100 sec	100 sec
Compression	17 sec	40 sec	10 sec	10 sec

Execution times are denoted in terms of the number and type of replication scheme by which the application was deployed. For example, *2S* denotes the case when the application executes in dual-redundant mode and is spatially replicated (i.e., the replicas execute simultaneously on different data processors). For a temporal replication deployment (denoted *T*), the application is deployed to any available data processor and then replicated only upon the completion of that instance. In this manner, no two replicas execute in the system simultaneously. It should also be noted that only single faults are being investigated at this time and therefore if a miscompare is detected for a triple-redundant deployment then the single invalid output is ignored and one of the two valid outputs is chosen. However, if a failure is detected for a dual-redundant deployment then the entire application must be rerun from scratch (a common method used in self-checking pairs).

For clarification, it should be noted that execution times for spatial replicas processing the small data set have been measured to be more than half the execution times of those when processing the large data set though the data size has halved. This result is due to contention at the mass data store from the increase in the communication to computation ratio when

processing the smaller data set. Temporal replications are unaffected by this contention because each replica executes consecutively and therefore does not contend for access to the mass data store.

4.5 Autonomic Replication Analysis

Several experiments were conducted to analyze the ability of the MM to deploy application replicas in an optimal fashion. As previously described, the MM collects information on environmental radiation levels and uses this information along with data on resource loading to decide what level of application fault tolerance to employ. As the testbed is neither equipped with radiation sensors (because they are currently under development) nor subjected to radiation tests, synthetic radiation data for several orbits was supplied to the MM in the course of these experiments.

To provide a set of baseline performance values against which to compare the experiments, the previously described mission was first executed by the MM while providing a stimulus mimicking the subjection of the system to varying levels of radiation. There is a fine art to predicting how to translate from a given level of radiation to a resultant number of manifest faults. To keep the experiment from becoming too complex, instead of defining a level of radiation we assume a number of system upsets per time period that represents manifest faults due to high-energy particle collisions. As there is a correlation between the level of radiation and number of manifest faults, this assumption is reasonable. For these baseline experiments, a predefined type of replication policy was imposed on the MM (i.e., the adaptable features of the MM were turned off). Also, all applications in the mission were replicated in the same fashion during each experiment while the MM could normally mix and match replication schemes on a per-application basis.

4.5.1 Application Scheduling and Deployment

The results for this particular mission with a large data set (see Figure 4-3) show that both forms of spatial replication outperform the temporal replications (though the difference is only slight for the triple-replica case). Also, the dual-spatial replicas outperform the triple-spatial replicas with increasing upset rates until roughly 5.2 upsets per hour and then degrade in relative performance. This result is due to the voting policy employed in the experiment whereby if a dual-replication algorithm encounters a fault the entire job must be rerun but if a triple-replicated application encounters a single data error then one of the two valid output files will be selected. Therefore, the triple-replicas will not require a rerun. However, the results show the triple-replicated scheme begins to suffer from double data errors at 6.5 upsets per hour, at which instance the entire job must be rerun. If the DM system is subject to too high a level of radiation from which the system cannot recover, such as that in which more than 6 or 6.5 upsets per hour are observed (e.g., a solar proton event), the MM will shutdown the DM system until the level of radiation is again within tolerance. This method of self-preservation is standard for most spacecraft.

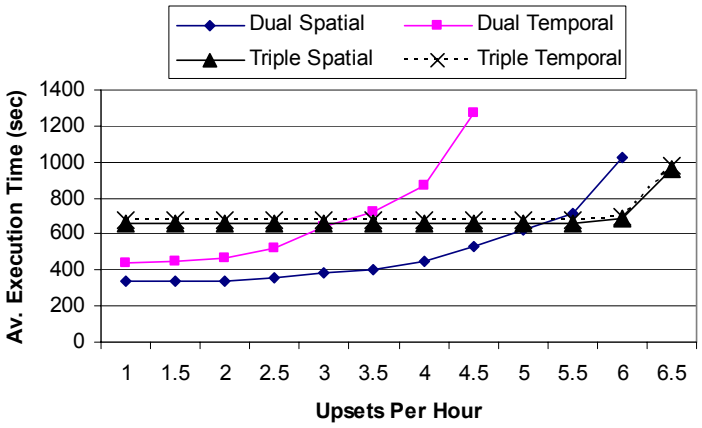


Figure 4-3. Average mission execution times with the large data set for various replication techniques

The results for this particular mission with a small data set (see Figure 4-4) show that both forms of temporal replication outperform the spatial replicas. As previously described, temporal replicas do not contend for the mass data store as do the spatial versions. Also, the dual-temporal replicas outperform the triple-temporal replicas with increasing upset rates until roughly 3.9 upsets per hour and then degrades in relative performance. This result is again due to the voting policy employed in the experiment. As with the large data set results, the triple-replicated scheme begins to suffer from double data errors at 6.5 upsets per hour, at which instance the entire job must be rerun.

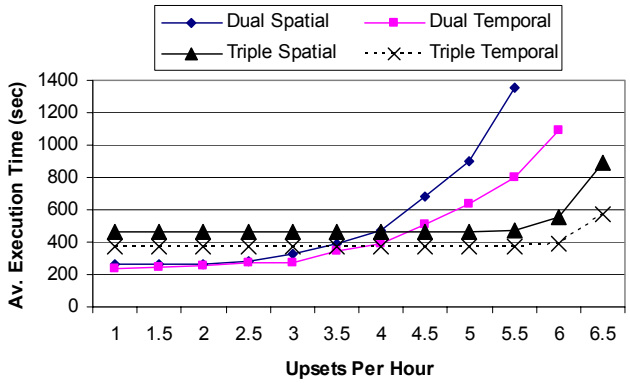


Figure 4-4. Average mission execution times with the small data set for various replication techniques

4.5.2. ST-8 Mission Orbit

In order to predict the performance of the DM system in-orbit, experiments were performed with synthetic upsets based on the predicted upset rates per hour for one of several proposed orbits the ST-8 experiment is slated to enter when deployed in 2009 (see Figure 4-5). The data presented is for a highly elliptical orbit defined as having a perigee of 1400km (over the poles), an apogee of 300km and a 98° inclination with a mean travel time of 101 minutes. This orbit was chosen to ensure the experiment is subject to a significant amount of radiation by traveling over the poles at a relatively high altitude. Figure 4-5 provides the predicted upset rate

for each ten-minute increment of the orbit (averaged over that distance). The peaks at 10 and 70 minutes denote the poles and the peak at 40 minutes denotes the South Atlantic Anomaly.

The previously described mission was executed by the MM while providing the radiation stimulus mimicking the orbit radiation data shown in Figure 4-5. Three different replication schemes were compared for large and small data sets (see Figures. 4-6 and 4-7) including the best two schemes from the baseline experiments (i.e., spatial replication for the large data sets and temporal for the small) and the MM's adaptive scheme. To be conservative, the MM is forced to select the poorer performing of the two replication schemes for the first time it traverses the orbit (denoted 1st Pass). As previously described, the MM has the ability to maintain a history to aid in predicting future patterns and therefore the results of including the historical information in the decision process are observed in the second orbit (denoted 2nd Pass). Figures 4-6 and 4-7 provide the average execution time values over discrete ten-minute intervals along the orbit for the large-data and small-data sets respectively. Table 4-4 summarizes the average total execution time for the mission deployed in each scheme over one entire orbit.

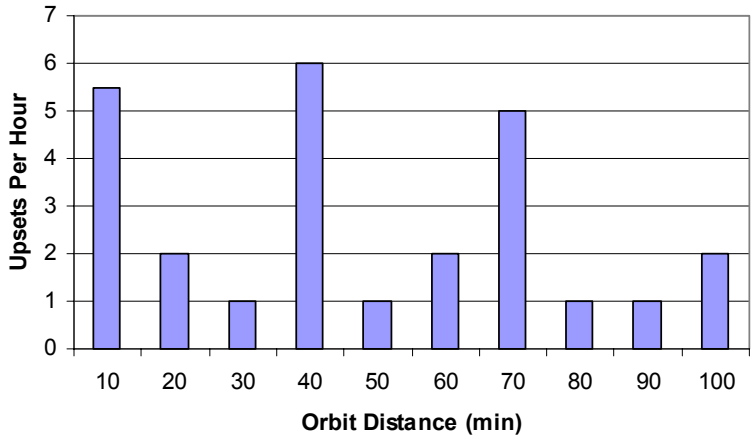


Figure 4-5. Predicted upsets per hour over a single ST-8 mission orbit

The results for the large data set experiments (see Figure 4-6) show the dual-spatial replication scheme outperforms the triple-spatial scheme and the first pass of the case when the MM is allowed to adapt to the perceived levels of radiation in the environment (denoted *Adaptive*) due to the relatively long execution times of the triple-replica scheme and the overall low radiation levels predicted in the orbit. With a relatively small likelihood of a fault occurring, the performance improvement gained by deploying a reduced level of fault tolerance outweighs the penalty suffered due to the few instances when a fault does occur. However, incorporating historical information into the decision-making process provides a benefit and the adaptive system outperforms the dual-spatial scheme in that instance.

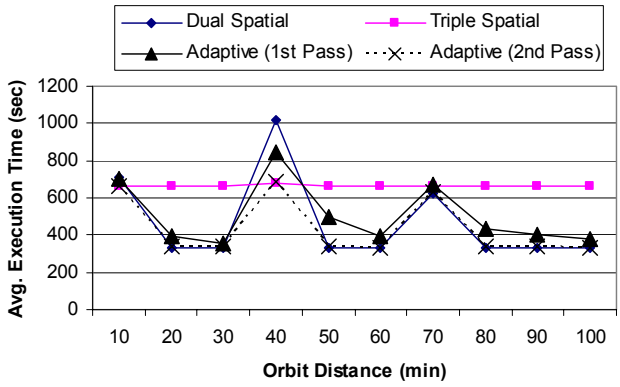


Figure 4-6. Average mission execution times for the large data set over a single ST-8 mission orbit

The results for the small data set experiments (see Figure 4-7) show the triple-temporal replication scheme outperforms the dual-temporal scheme and the adaptive scheme (for the first pass) due to the overhead penalty associated with rerunning failed executions in the dual-temporal scheme. For the small data set sizes, even the relatively low radiation levels predicted in the orbit affect the performance due to the high degree of variability in mission execution time. The adaptive scheme does not respond as quickly on the first pass for this experiment as compared to the large data set results because of increased variability in the performance of the

dual-temporal replicas. However, incorporating historical information into the decision-making process provides a more pronounced improvement (22.5%) compared to the results observed for large data sets (7.3%) and the adaptive system outperforms the triple-temporal scheme in that instance. These and all subsequent improvement results are averaged over ten experimental trials with less than 1% variability observed.

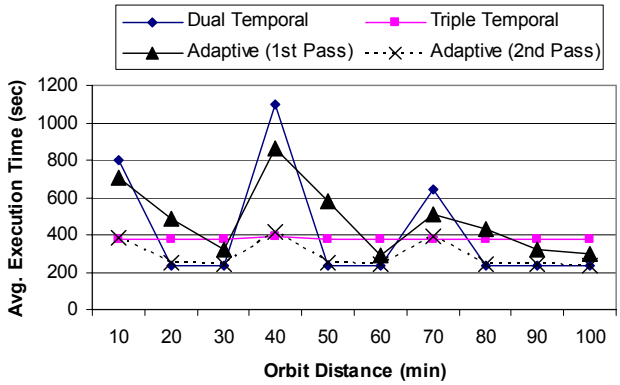


Figure 4-7. Average mission execution times for the small data set over a single ST-8 mission orbit

Table 4-4. Average Mission Execution Time Per Orbit for the ST-8 Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	458.3625
Triple Spatial	662.2278
Adaptive (1st Pass)	494.0694
Adaptive (2nd Pass)	424.7961
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	420.7652
Triple Temporal	376.0457
Adaptive (1st Pass)	481.2993
Adaptive (2nd Pass)	291.2609

4.5.3 International Space Station Orbit

To predict the performance of the DM system in other potential missions, a set of experiments was performed with synthetic upsets based on the predicted upset rates per hour from data collected on the International Space Station (ISS). The ISS orbit is defined as circular

with at an altitude of 400km and a 51.65° inclination with a mean travel time of 92 minutes. This orbit ensures the ISS does not traverse the Earth's poles in order to reduce the levels of radiation to which the crew are subjected. The upset rate data shown in Figure 4-8 was extrapolated from radiation measurements on the ISS [87] and are displayed as the predicted upset rate for each ten-minute increment of the orbit (averaged over that distance). The peaks at 10 and 70 minutes denote the times at which the ISS travels closest to the poles and the peak at 40 minutes denotes the South Atlantic Anomaly.

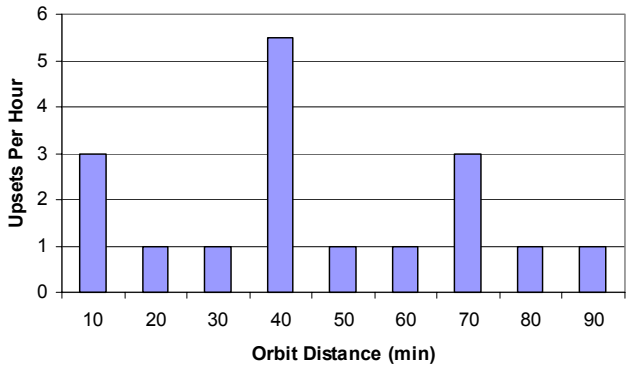


Figure 4-8. Predicted upsets per hour over a single ISS orbit

The previously described mission was executed by the MM while providing the radiation stimulus mimicking the orbit radiation data shown in Figure 4-8. As in the previous experiments, the best replication schemes were compared to the MM's adaptive scheme for large and small data sets. Figures 4-9 and 4-10 provide the average execution time values over discrete ten-minute intervals along the orbit for the large-data and small-data sets respectively. Table 4-5 provides a summary of the average total execution time for the mission deployed in each scheme over one entire orbit.

The results for the large data set experiments show the dual-spatial replication scheme outperforms the triple-spatial scheme and the adaptive scheme to a greater degree than on the

ST-8 mission due to the relatively long execution times of the triple-replicated scheme and the overall low radiation levels predicted in the orbit. With a relatively small likelihood of a fault occurring, the performance improvement gained by deploying a reduced level of fault tolerance outweighs the penalty suffered due to the few instances when a fault does occur. Also, incorporating historical information into the decision-making process for the large data set experiment did not provide a benefit for the ISS orbit.

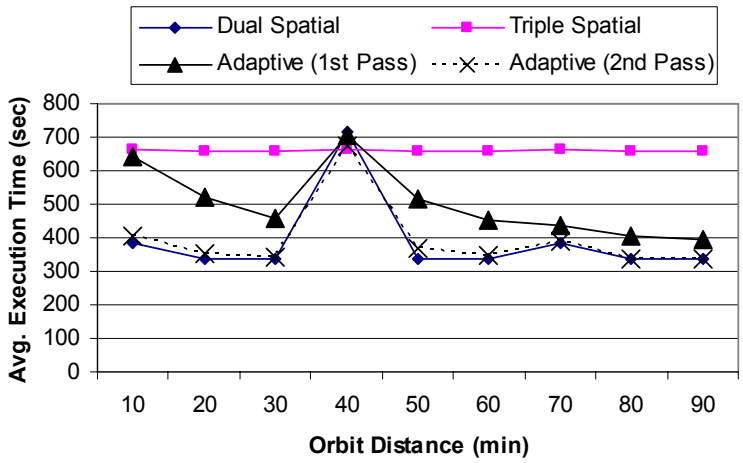


Figure 4-9. Average mission execution times for the large data set over a single ISS orbit

The results for the small data set experiments show the dual-temporal replication scheme outperforms the triple-temporal scheme and the adaptive scheme (for the first pass) due to the relatively long execution times of the triple-replicated scheme and the overall low radiation levels predicted in the orbit. For the small data set sizes, even the relatively low radiation levels predicted in the orbit affect the performance due to the high degree of variability in mission execution time for the dual-replicated scheme (i.e., at the SAA). Incorporating historical information into the decision-making process for the small data set experiments provides a performance improvement (12.6%) and the adaptive system outperforms the dual-temporal scheme in that instance.

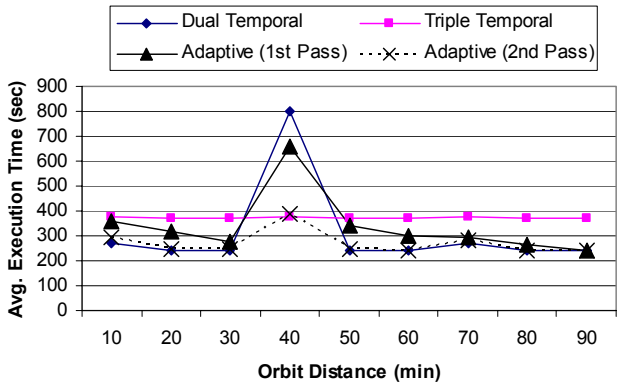


Figure 4-10. Average mission execution times for the small data set over a single ISS orbit

Table 4-5. Average Mission Execution Time Per Orbit for the ISS Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	349.3783
Triple Spatial	594.4989
Adaptive (1st Pass)	453.0311
Adaptive (2nd Pass)	354.8135
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	277.8657
Triple Temporal	336.9555
Adaptive (1st Pass)	306.1490
Adaptive (2nd Pass)	242.9705

4.5.4 Hubble Space Telescope Orbit

The previous results demonstrate that the performance improvement provided by the DM system's ability to adapt to observed levels of radiation are more pronounced for orbits with larger rates of SEUs. To predict the performance of the DM system in other orbits, a set of experiments were performed with synthetic upsets based on the predicted upset rates per hour from data collected on the Hubble Space Telescope (HST). The HST orbit is defined as circular at an altitude of 600km and a 28.5° inclination with a mean travel time of 100 minutes. The orbit of the HST does not pass as close to the Earth's poles as does the orbit of the ISS. However, because it is traveling at a higher altitude, the HST is subject to higher levels of radiation on average than is the ISS. The upset rate data shown in Figure 4-11 was extrapolated from

radiation measurements from the HST [88] and are displayed as the predicted upset rate for each ten-minute increment of the orbit (averaged over that distance). The peaks at 10 and 70 minutes denote the times at which the HST travels closest to the poles and the peak at 40 minutes denotes the South Atlantic Anomaly. As in the previous missions, the previously described mission was executed by the MM while providing the radiation stimulus for the orbit data shown in Figure 4-11.

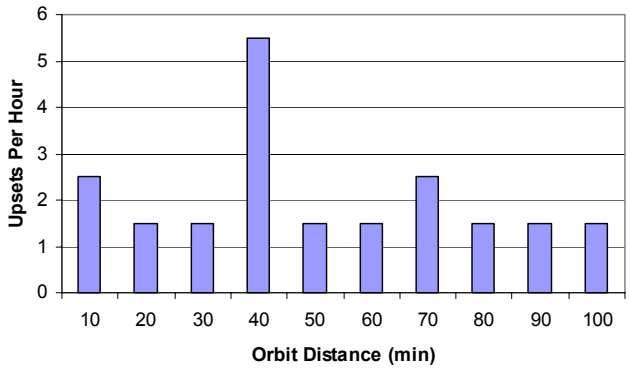


Figure 4-11. Predicted upsets per hour over a single HST orbit

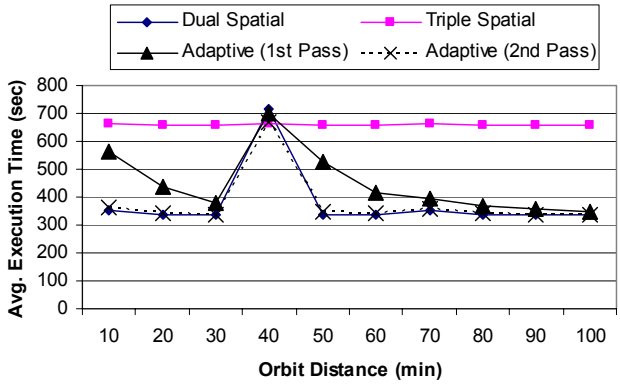


Figure 4-12. Average mission execution times for the large data set over a single HST orbit

The results for the large data set experiments (see Figure 4-12) show the dual-spatial replication scheme outperforms the triple-spatial scheme and the adaptive scheme (first pass) to a greater degree than on the ST-8 mission due to the relatively long execution times of the triple-

replicated scheme and the overall low radiation levels predicted in the orbit. Again, with a relatively small likelihood of a fault occurring, the performance improvement gained by deploying a reduced level of fault tolerance outweighs the penalty suffered due to the few instances when a fault does occur. Also, incorporating historical information into the decision-making process for the large data set experiment provides a small improvement over the dual-spatial scheme for the HST orbit.

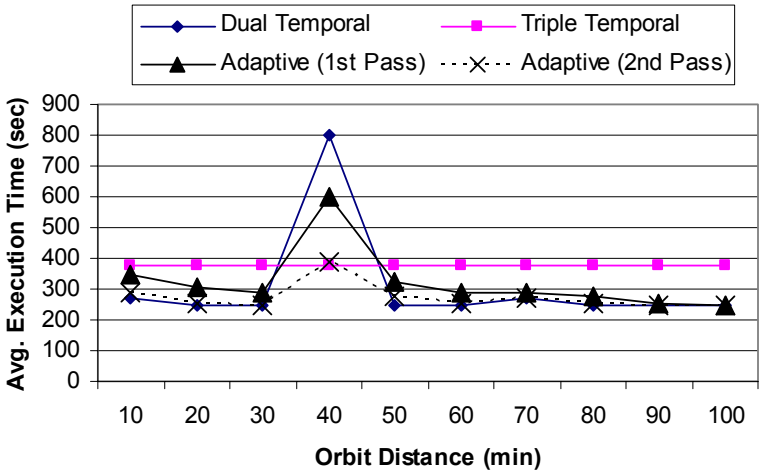


Figure 4-13. Average mission execution times for the small data set over a single HST orbit

The results for the small data set experiments (see Figure 4-13) show the dual-temporal replication scheme outperforms the triple-temporal scheme and the adaptive scheme (for the first pass) due to the relatively long execution times of the triple-replicated scheme and the overall low radiation levels predicted in the orbit. For the small data set sizes, even the relatively low radiation levels predicted in the orbit affect the performance due to the high degree of variability in mission execution time for the dual-replicated scheme (i.e., at the SAA). However, incorporating historical information into the decision-making process provides a more pronounced improvement (10.8%) compared to the results observed for large data sets (0.3%) and the adaptive system outperforms the dual-temporal scheme in that instance.

Table 4-6. Average Mission Execution Time Per Orbit for the HST Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	378.8977
Triple Spatial	660.4931
Adaptive (1st Pass)	448.9761
Adaptive (2nd Pass)	377.8815
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	305.7499
Triple Temporal	375.2958
Adaptive (1st Pass)	322.1881
Adaptive (2nd Pass)	272.8547

4.5.5 Global Positioning System Satellite Orbit

The ISS and HST traverse low-earth orbits which are typically not subject to high levels of radiation. To test the limits of the DM system's ability to guard against high levels of radiation, two sets of experiments were performed with synthetic upsets based on the predicted upset rates per hour from two high-earth orbits. This section highlights the orbit taken by satellites forming the Global Position System (GPS) and the next section provides a case study of the radiation experienced in a high-altitude Geostationary Orbit (denoted GEO). The orbit of GPS satellites is defined as circular at an altitude of 22600km and a 55° inclination with a mean travel time of 720 minutes. The orbit of GPS satellites tracks fairly close to that of the ISS (just many kilometers higher) and therefore passes close to the Earth's poles. These satellites do not pass within the bounds of the SAA (which extends from roughly 100km to 1500km above the Earth's surface) and therefore are not subject to its effects. However, due to the high altitude of their orbit, GPS satellites are subject to relatively high levels of radiation compared to other orbits. The upset rate data shown in Fig. 16 was extrapolated from radiation estimates for this orbit [89, 90] and displayed as the predicted upset rate for each sixty-minute increment of the orbit (averaged over that distance). The peaks at 60 and 420 minutes denote the times at which the satellite travels

closest to the poles. As in the previous missions, the previously described mission was executed by the MM while providing the radiation stimulus for the orbit data shown in Figure 4-14.

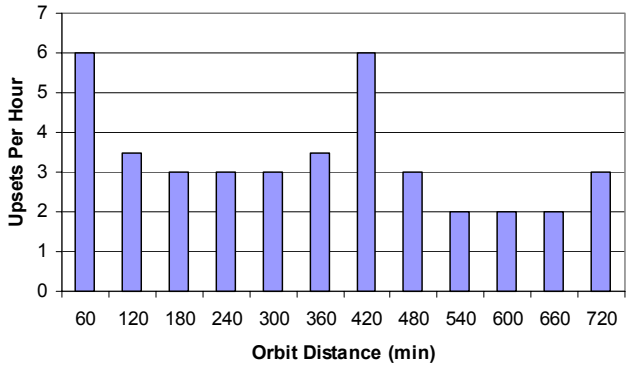


Figure 4-14. Predicted upsets per hour over a single GPS satellite orbit

The results for the large data set experiments (see Figure 4-15) show the dual-spatial replication scheme outperforms the triple-spatial scheme due again to the relatively long execution times of the triplicate version. However, unlike previous orbits, the adaptive scheme outperforms the dual- and triple-spatial schemes even on the first pass. Due to the relatively high levels of radiation in the orbit, the ability of the DM system to adapt provides a significant benefit. Also, incorporating historical information into the decision-making process for the large data set experiment provides an additional improvement over the dual-spatial scheme for the GPS orbit.

The results for the small data set experiments (see Figure 4-16) show the triple-temporal replication scheme outperforms the dual-temporal scheme and the adaptive scheme (for the first pass) due to the overhead penalty associated with rerunning failed executions in the dual-temporal scheme. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the dual-replicated performance due to the overhead associated with rerunning due to a failure. In fact, in this experiment, frames of data become backlogged by the dual-temporal

scheme after each instance the spacecraft traverses the poles during the orbit. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the performance of the first pass of the adaptive scheme because of the choice to take a conservative approach and pick the scheme with the worst performance as the initial starting condition for the MM. However, incorporating historical information into the decision-making process provides a performance improvement for both the small data set experiment (16.0%) and for large data sets (16.1%).

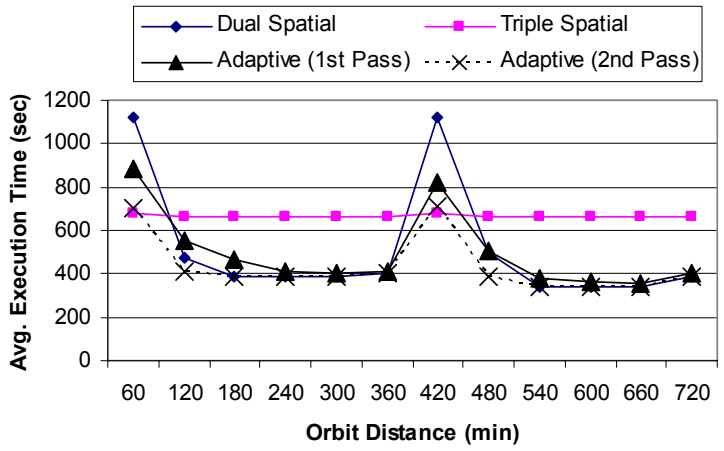


Figure 4-15. Average mission execution times for the large data set over a single GPS satellite orbit

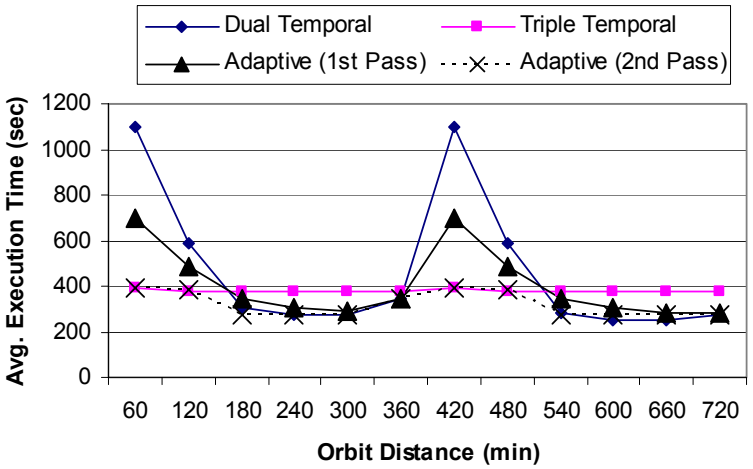


Figure 4-16. Average mission execution times for the small data set over a single GPS satellite orbit

Table 4-7. Average Mission Execution Time Per Orbit for the GPS Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	514.4008
Triple Spatial	665.0813
Adaptive (1st Pass)	495.7927
Adaptive (2nd Pass)	431.4807
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	468.8840
Triple Temporal	378.8465
Adaptive (1st Pass)	405.9629
Adaptive (2nd Pass)	318.1277

4.5.6 Geosynchronous Satellite Orbit

This section highlights the orbit taken by satellites in a high-altitude Geostationary Orbit (GEO). The orbit of GEO satellites is defined as circular at any altitude with an inclination of 0° (i.e., positioned over the Earth's equator) with a mean travel time of 1440 minutes (i.e., one Earth day). Satellites in GEO orbits do not pass close to the poles or through the SAA and are therefore not subject to high radiation levels associated with these zones. However, GEO satellites are typically placed at relatively high altitudes and are therefore subject to higher levels of radiation averaged over their orbit. The upset rate data shown in Figure 4-17 was extrapolated from radiation estimates for a GEO orbit with an altitude of 35790km [89, 90] and are displayed as the predicted upset rate for each 120-minute increment of the orbit (averaged over that distance). The peaks between 120 and 720 minutes denote the times at which the satellite is facing the sun and therefore not shielded by the Earth. As in the previous missions, the previously described mission was executed by the MM while providing the radiation stimulus for the orbit data shown in Figure 4-17.

The results for the large data set experiments (see Figure 4-18) show the dual-spatial replication scheme outperforms the triple-spatial scheme and the adaptive scheme (first pass) even with the relatively high radiation levels in the orbit. For the first half of the orbit, the

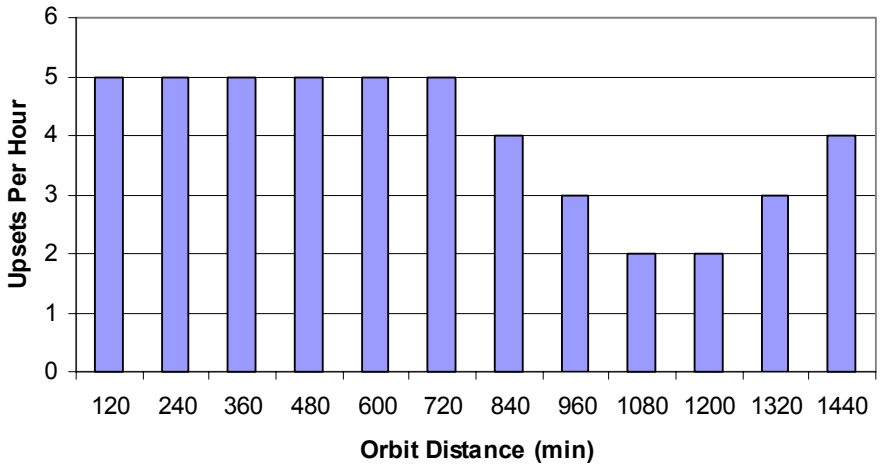


Figure 4-17. Predicted upsets per hour over a single GEO satellite orbit

overhead in terms of time lost due to recomputing a spatial replication is roughly equivalent to the overhead due to computing the mission in a triple-spatial deployment strategy. The dual-spatial replication scheme outperforms the triple-spatial scheme at all points in the orbit for large data sizes so there is no need to switch between the two deployment options. Therefore, the adaptive scheme does not provide an improvement when computing large data sizes on the GEO orbit.

The results for the small data set experiments (see Figure 4-19) show the triple-temporal replication scheme outperforms the dual-temporal scheme due to the overhead penalty associated with rerunning failed executions in the dual-temporal scheme. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the dual-replicated performance due to the overhead associated with rerunning due to a failure. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the performance of the first pass of the adaptive scheme because of the choice to take a conservative approach and pick the scheme with the worst performance as the initial starting condition for the MM. However, the adaptive scheme provides better performance than the triple-temporal replication scheme even for the first

pass. Incorporating historical information into the decision-making process for the small data set experiments provides an additional performance improvement (9.0%) over the triple-temporal scheme.

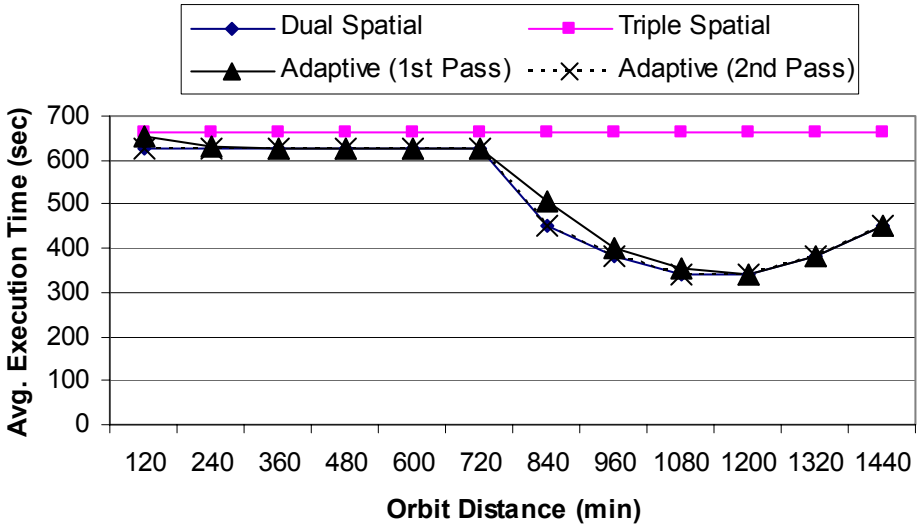


Figure 4-18. Average mission execution times for the large data set over a single GEO satellite orbit

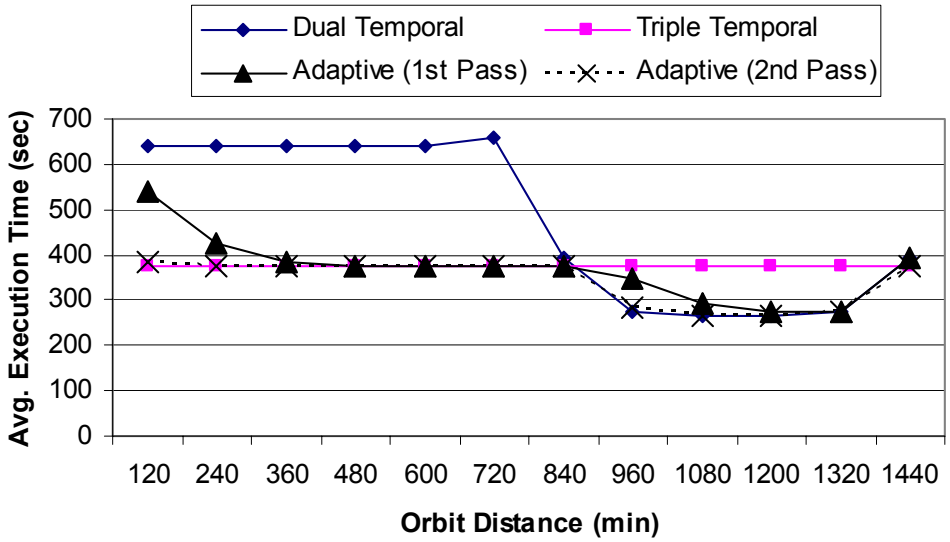


Figure 4-19. Average mission execution times for the small data set over a single GEO satellite orbit

Table 4-8. Average Mission Execution Time Per Orbit for the GEO Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	508.7437
Triple Spatial	661.7379
Adaptive (1st Pass)	518.4839
Adaptive (2nd Pass)	508.7442
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	477.0293
Triple Temporal	376.1819
Adaptive (1st Pass)	370.1834
Adaptive (2nd Pass)	342.4550

4.5.7 Martian Orbit

The previous results demonstrate the DM system's ability to provide performance benefits for most Earth orbits and additional experiments were conducted with synthetic upsets based on the predicted upset rates per hour from data collected on the MArtian Radiation envIronment Experiment (MARIE) to predict the performance of the DM system in an orbit around Mars. The MARIE experiment was launched in 2001 onboard the Mars Odyssey Orbiter and collected radiation data from March 2002 until September 2003. The MARIE orbit is defined as circular with a radius of 400km and a 93.1° inclination with a mean travel time of 120 minutes. The upset rate data shown in Figure 4-20 was extrapolated from MARIE radiation measurements [91] and displayed as the predicted upset rate for each ten-minute increment of the orbit (averaged over that distance). The values from 10 minutes to 60 minutes denote the locations at which the spacecraft is facing the sun and the other values denote instances when the spacecraft is shielded from the sun by Mars. As in the previous missions, the previously described mission was executed by the MM with the radiation stimulus for the orbit data shown in Figure 4-20.

The results for the large data set experiments (see Figure 4-21) show the dual-spatial replication scheme outperforms the triple-spatial scheme due again to the relatively long execution times of the triplicate version. However, the adaptive scheme outperforms the dual-

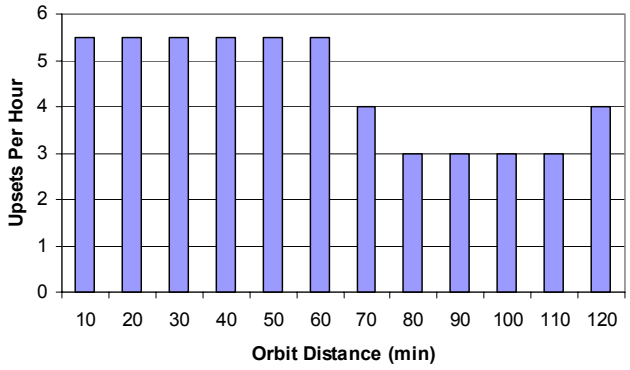


Figure 4-20. Predicted upsets per hour over a single MARIE orbit

and triple-spatial schemes even on the first pass. Due to the relatively high levels of radiation, the ability of the DM system to adapt provides a significant benefit. Also, incorporating historical information into the decision-making process for the large data set experiment provides an additional improvement over the dual-spatial scheme for the MARIE orbit.

The results for the small data set experiments (see Figure 4-22) show the triple-temporal replication scheme outperforms the dual-temporal scheme and the adaptive scheme (for the first pass) due to the overhead penalty associated with rerunning failed executions in the dual-temporal scheme. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the dual-replicated scheme's performance due to the overhead associated with rerunning due to a failure. In fact, in this experiment, frames of data become backlogged by the dual-temporal scheme in the first half of the orbit. For the small data set sizes, the relatively high radiation levels predicted in the orbit affect the performance of the first pass of the adaptive scheme because of the choice to take a conservative approach and pick the scheme with the worst performance as the initial starting condition for the MM. However, incorporating historical information into the decision-making process provides a performance improvement for both the small data set experiment (8.3%) and for large data sets (5.6%).

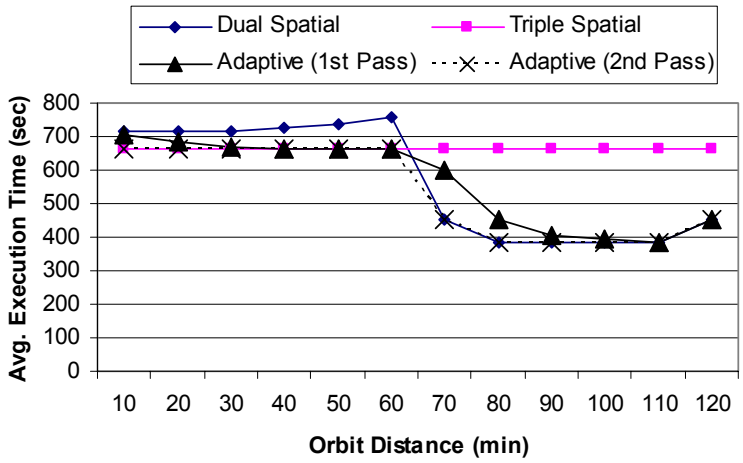


Figure 4-21. Average mission execution times for the large data set over a single MARIE orbit

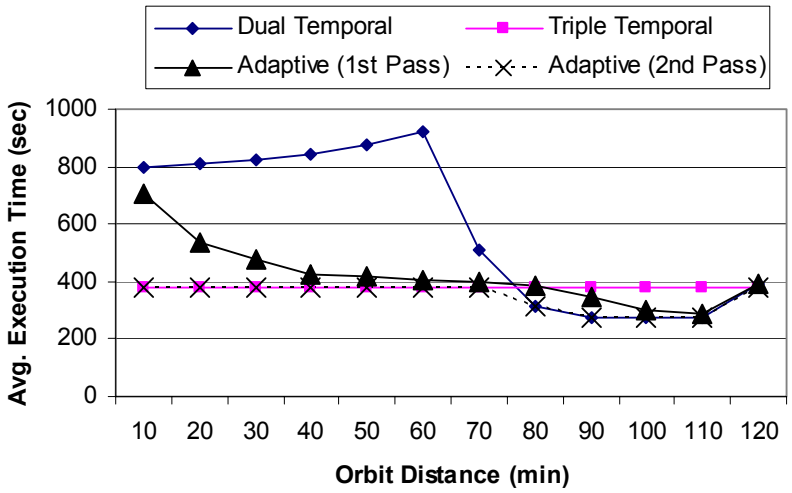


Figure 4-22. Average mission execution times for the small data set over a single MARIE orbit

Table 4-9. Average Mission Execution Time Per Orbit for the MARIE Mission

Large Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Spatial	566.9304
Triple Spatial	661.9557
Adaptive (1st Pass)	561.4724
Adaptive (2nd Pass)	534.8754
Small Data Set Experiments	
Scheme	Average Execution Time (sec)
Dual Temporal	592.2515
Triple Temporal	376.2472
Adaptive (1st Pass)	423.2146
Adaptive (2nd Pass)	345.1141

4.5.8 Summary of Results

Tables 4-10 and 4-11 summarize the performance improvements achieved by the DM system's adaptive deployment scheme for the large- and small-data experiments respectively. Results are sorted based on the level of performance improvement achieved for each orbit.

Table 4-10. Adaptive Deployment Performance Improvement for Large Data Sets

Orbit	Average Radiation Level (SEUs/hr)	Maximum Radiation Level (SEUs/hr)	Adaptive Scheme Performance Improvement
GPS	3.33	6	16.1%
ST-8	2.65	6	7.3%
MARIE	4.42	5.5	5.6%
HST	2.10	5.5	0.3%
GEO	4.00	5	0.0%
ISS	1.94	5.5	-1.5%

Performance improvements for the experiments with large data sets generally track with the maximum radiation level experienced by the spacecraft in the orbit because the cross-over point between the performance of the dual and triple replicated schemes is 5.2 SEUs per hour. Therefore, greater performance improvements are achieved by using the adaptive scheme when the maximum radiation level experienced in orbit rises highest above this cross-over point. Also, no performance improvement is provided by the adaptive scheme when the maximum radiation level does not rise above this cross-over point because adaptation is not required. However, the average radiation level also dictates the level of improvement achieved for orbits that have a given level of maximum radiation. The greater the average level of radiation experienced in the orbit the greater the improvement obtained by using the adaptive scheme because the improvement gained each time the system switches replication strategies is amplified the longer the system is exposed to that level of radiation. This trend is evident in the summarized data in Table 4-10 and is the contributing factor to the decrease in performance in the case of the ISS orbit even though the maximum level of radiation experienced in the orbit exceeds the cross-over point.

Table 4-11. Adaptive Deployment Performance Improvement for Small Data Sets

Orbit	Average Radiation Level (SEUs/hr)	Average Radiation Level Rate of Change (SEUs/hr^2)	Adaptive Scheme Performance Improvement
ST-8	2.65	2.7	22.5%
GPS	3.33	1.4	16.0%
ISS	1.94	1.7	12.6%
HST	2.10	1.2	10.8%
GEO	4.00	0.6	9.0%
MARIE	4.42	0.5	8.3%

Performance improvements for the small data set experiments follow a slightly different trend because of reduced execution times and the fact that the cross-over point between the performance of the dual- and triple-replicated schemes is much lower than for the large data set experiments (3.9 SEUs per hour). Increased performance improvements for small data sets are achieved by using the adaptive scheme when the system must quickly adapt to changing radiation levels in the environment. Therefore, the average rate of change in the radiation level experienced in the orbit (see Table 4-11) generally dictates the level of performance improvement achieved by using the adaptive scheme. However, the average radiation level also affects achieved performance because the improvement gained each time the system switches replication strategies is amplified the longer the system is exposed to that level of radiation. This phenomenon explains the lower than expected performance improvement for the ISS orbit.

4.6 CONCLUSIONS

The Mission Manager and other constituent components of the CARMA-enhanced Dependable Multiprocessor (DM) middleware provide a dependable and autonomic management system. The agent-based nature of the design makes the system resilient to faults and the distributed health and environmental information gathering system reduces the impact on system performance and ensures timely and guaranteed delivery. The MM provides a general-purpose yet comprehensive mission management system and a framework that can also be easily tailored to meet the needs of a diverse set of missions.

The DM middleware also addresses the "self-*" constraints that often set the standard for autonomic systems. Agents are deployed by the FTM in the system based on the node type, and the state of the system providing a measure of self-configurability. Also, the FTM employs a combination of software and hardware probes to passively learn the state of the system but also actively launches diagnostic routines to gain further insight if an anomaly should occur. The FTM is responsible for detecting and self-healing any systemic faults by either restarting agents or power-cycling hardware resources. The FTM also gathers and distills system health information to aid in scheduling decisions whereby the MM and JM may "route around" any permanent system fault of data processors thus providing an additional measure of self-healing ability at the job management level.

Sensors are deployed strategically throughout the DM system to detect radiation levels of the current environment. The MM may use this information to predict the likelihood of a fault when determining what degree of replication to deploy to ensure the correct level of dependability is maintained per mission policies. A priori knowledge of radiation levels improves the MM's ability to effectively deploy applications with the optimal level of replication, affording the system a high degree of self-optimization and self-protection that otherwise would not be possible. An analysis of the MM ability to improve system dependability and performance through adaptable fault tolerance was demonstrated in several case studies. The results show the adaptable scheme outperforms standard fixed schemes when including a history database in decision making. The adaptable deployment scheme provides the most performance improvement when the spacecraft is subjected to high levels of radiation and also must quickly adapt to the radiation environment. Also, the DM system allows computation to continue through the SAA where most electronic systems must shutdown. The next chapter

outlines the design, investigation and analysis of CARMA for large-scale, ground-based computational clusters.

CHAPTER 5
CARMA FOR LARGE-SCALE CLUSTER COMPUTING

The two previous chapters outlined a version of CARMA that focuses on improving system fault tolerance for embedded space systems. With some design modifications, the CARMA infrastructure can also ease the transition from traditional-processor to dual-paradigm systems to address the issue of large-scale RC system usability [92]. The best aspects of previous JMS tools and related RC projects described in Chapter 2 have been integrated to build a robust and scalable framework tailored for dual-paradigm systems. Also, the limiting aspects of previous tools have been avoided based on lessons learned by previous researchers. This chapter presents the design and analysis of a fully-distributed version of CARMA developed for large-scale clusters.

5.1 Design Overview

The Comprehensive Approach to Reconfigurable Management Architecture (CARMA) seeks to ease the integration of special-purpose devices into traditional high-performance computational systems. CARMA provides a standard interface to a diverse set of computational resources using this standard to normalize the runtime management of heterogeneous components. A description of the distributed service's design philosophy and main constituents follows.

5.1.1 Framework Design Philosophy

The modular and flexible CARMA framework was designed to support scalability and fault tolerance with minimal overhead while also promoting future upgrades and extensions. To support large-scale systems and ensure scalability, the runtime management service is composed of numerous independent, fully-distributed software agents that frequently communicate to schedule and execute tasks and configure devices among other management duties. Fault

tolerance has been inherently designed into the framework at numerous levels. The service's fully distributed infrastructure ensures no single point of failure. Also, each agent is implemented as a separate process rather than POSIX thread to allow runaway or crashed processes to be restarted without requiring a system-wide reboot during which valuable time and information would be lost. In addition, message queues (rather than shared memory, pipes, etc.) are used as the inter-process communication mechanism, and this feature allows processes to be stopped and restarted as necessary without losing in-flight management messages. The ability to quickly repair agents is very important for systems that must maintain high availability. Message queues also allow for numerous processes to share complex messages in a simple and scalable fashion. The framework's modular design increases extensibility and eases the integration of new resources and tools. With standard interfaces between components, end users are free to link, extend and replace components with the framework in a "plug-and-play" style. For example, an advanced job scheduler can replace the simplistic one developed to demonstrate the infrastructure's validity, and other examples such as this one are described in this section. Such extendibility promotes a collaborative approach for industry partners, standards committees and other research groups to contribute to the ongoing CARMA development effort.

The collection of software agents within CARMA that exists on each node in a heterogeneous cluster is shown in Figure 5-1. A node is defined as a set of processing components that are controlled by an independent and discernable operating system in the case of Beowulf clusters or an appropriate subset of resources in a large-scale system that has an operating system image that spans all processors. Each box in the figure denotes a runtime software agent, software library, development tool, source or executable file or hardware component. It should be noted that grey components are concepts, tools or hardware system

components (i.e., networks, processors and the RC fabric) that do not necessarily correspond to a runtime agent and are thus beyond the scope of CARMA infrastructure and this paper. Also, the *User Executable* process and CARMA agents shown in Figure 5-1 are separate processes (not kernel-level modifications) executing on the traditional processor in each system node but they are shown as separate entities in Figure 5-1 for clarity. A detailed description of each component within the CARMA management system follows.

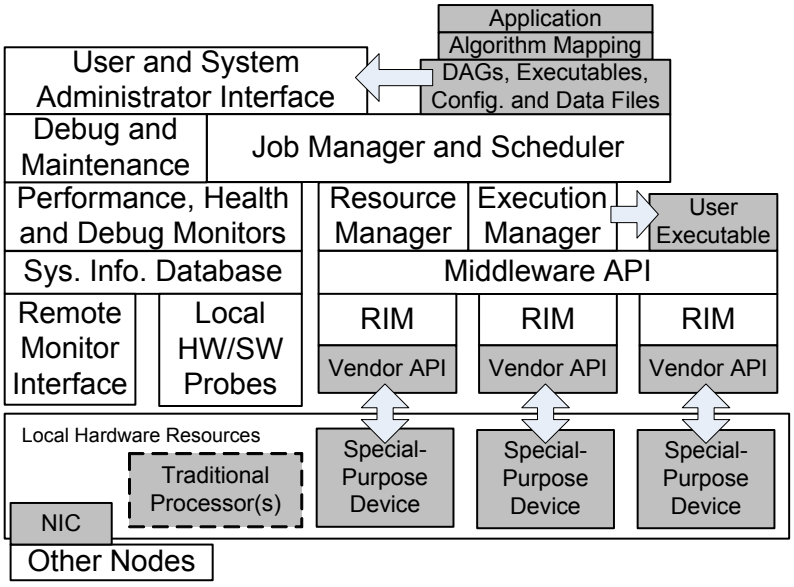


Figure 5-1. Distributed CARMA Framework

5.1.2 Representing Applications

As is required by most HPC job management tools, users must create and submit a collection of files used by the runtime service to execute their job. The *Applications* component in Figure 5-1 is a representation of the user's application that can be made to execute on the heterogeneous computer platform. Some typical forms include source code from high-level languages such as C/C++, MATLAB or FORTRAN, Hardware Description Languages (HDLs) such as VHDL or Verilog, or device-specific code represented as some form of assembly. The application code is passed to some collection of *Algorithm Mapping* tools that transform them

into one or more executable images for traditional processors and configuration files for special-purpose devices. Since compiler technology is well understood for traditional processors and numerous device-specific application mapping and code partitioning tools exist for special-purpose devices, the inclusion of the algorithm mapping preprocessing step is beyond the scope of the CARMA infrastructure at this time. However, an iterative algorithm mapping approach that optimizes the runtime performance of batch or recurring jobs may be included in the future. In not tailoring the management system to specific tools, the CARMA project seeks to support system heterogeneity by integrating as many devices into traditional HPC systems as possible. A more detailed description of how CARMA achieves this goal is found in the remainder of this section and more information on application mappers and other synergistic tools is provided in Chapter 2.

5.1.3 System Interface

CARMA's *User Interface* provides a global view of the system for users and administrators to access a variety of services. User's can register job executables and hardware configuration files, submit jobs and query the status of resources and outstanding jobs. Due to the distributed nature of the CARMA infrastructure, user interfaces can be opened on any node in the system without loss of consistency or limited control. Also, multiple users are supported on any node in the system simultaneously as well. System administrators are provided a more full-featured interface where they may deploy, stop or restart individual CARMA agents on a per-node or aggregated basis. Also, administrators may set control levels to restrict access to certain resources in the system and may also remove jobs and users as required. A text-based interface was developed first so as to devote more effort to developing other more important CARMA components but graphical interfaces are under consideration. A preliminary version has been developed using the Simple Web Interface Link Library (SWILL) from the University

of Chicago [93] to analyze any potential roadblocks that may lay in store for such a future effort. It has been determined that hierarchical and aggregated views will be critical in such an interface, especially when resource information is displayed. Also, one of many extendible debug analysis tools must be integrated with CARMA in order to provide a simple to use yet powerful interface for online debug support that includes special-purpose devices in the analysis. More information on how CARMA supports such an analysis is provided in Section 5.1.6.

5.1.4 Job Scheduling and Management

Figure 5-2 shows the subset of related CARMA agents and operations that interact to process jobs. As previously described, users submit a collection of files to the local *Job Manager* including Directed-Acyclic Graphs (DAGs), executables and configuration files required by special-purpose devices. DAGs have long been used to describe the control flow and data dependencies between a collection of interrelated processes [94]. CARMA uses DAGs to describe multiple parallel and sequential tasks and also to describe special-purpose device requirements such as device type and configuration. The local job manager is responsible for ensuring the correct execution of the job and manages the scheduling, deployment, coordination between remote job managers and recovery in the case of job failures. Jobs are registered and tracked in the system by the local job manager to which it was submitted, and in this sense the job is bound to that manager. Job information is disseminated system-wide through the distributed performance monitoring system, and jobs that are bound to a failed manager will be reassigned to another job manager through a simplistic method (e.g., incrementing the manager ID).

In managing a job, the local job manager compares the dependencies outlined in the DAG to system-wide resource information provided by local and remote resource managers to match application needs to available resources. The CARMA performance monitoring system provides

a highly-efficient distributed monitoring system that allows local job managers to make effective system-wide scheduling decisions and more information on how information is collected in a scalable and effective manner is provided in the next two sections. The local job scheduler currently employs a custom opportunistic load-balancing scheduler, with gang scheduling for parallel jobs [95] as a simplistic baseline component. Again, this and other components can be updated in the future as user's needs require and development advances. However, the custom scheduler provides a means to include device-specific information along with the usual metrics such as processor and memory utilization in the decision-making process. For device-specific analysis, the scheduler searches the system resource information database to locate nodes that offers an advantage based on characteristics such as preferred device type or configuration.

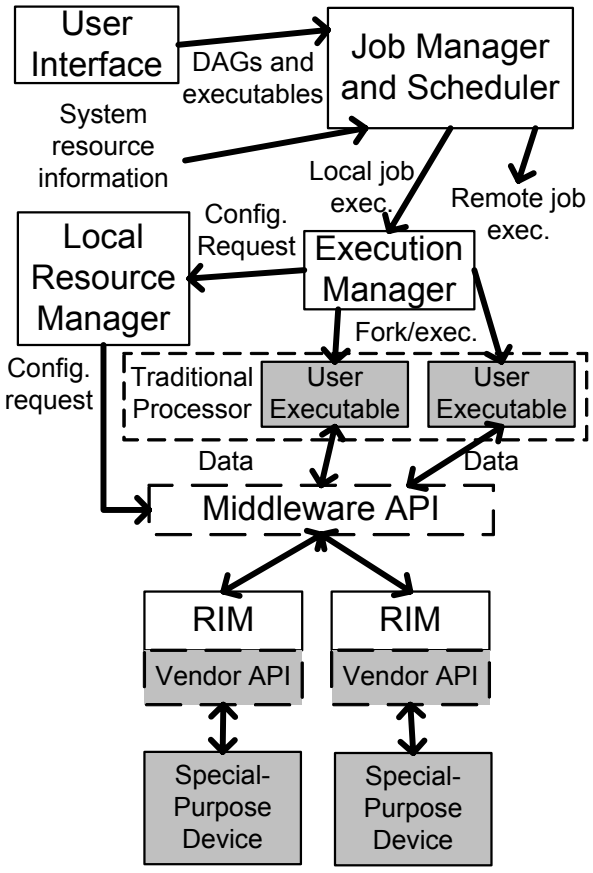


Figure 5-2. CARMA Job Management

When the local job manager determines the nodes on which the application will execute, the chosen local and remote *Execution Managers* are informed and given the required executables and other files. The execution manager manages the control flow for a task on a particular node and therefore requests the configuration and use of a resource and then forks and executes the user executable. The *Local Resource Manager* orchestrates the control of the special-purpose devices by serializing access to and configuring them for operation as required by the execution manager. This process is performed through the *Resource Interface Module (RIM)* which provides a standard interface to multiple device types. More information on how RIMs provide "resource normalization" follows.

5.1.5 Resource Normalization and Management

One of the key issues that must be addressed when managing heterogeneous systems is how to provide a generic access framework to special-purpose devices yet not limit a user's ability to exploit the special features they offer [96]. Typically, vendors develop an Application Programming Interface (API) that includes a standard set of function calls by which a user application may interact with a particular device. Such *Vendor APIs* are often custom-designed for a particular vendor's product line and therefore users are required to adapt their application if they choose to use a different vendor's device. This lack of portability is especially true for FPGA systems where many board vendors exist, each of which using a different API and associated runtime library. In addition to this inherent lack of code portability, most vendor APIs do not support a convenient method to provide multiple user access especially with regard to enforcing security. The vendor API approach (denoted *Option 1* in Figure 5-3) is the method by which users access special-purpose devices in systems deployed today.

In order to improve code portability and system usability and security, two additional options have been developed within the CARMA infrastructure as shown in Figure 5-3. Both

options include a *Middleware API* that provides a standard set of functions that is compatible across vendor APIs for a particular device type (e.g., FPGAs or GPUs). This API allows users to develop their applications in a "write-once-run-anywhere" philosophy that will improve code portability and allow them to focus on their applications rather than interfacing issues. In this manner, access to and management of special-purpose devices is normalized across all platforms drastically improving system usability and ease their integration into future systems. To improve system security, a means to enforce access protocols must be included in order to further decouple users from underlying devices. To support this need, *Option 3* includes a *Resource Interface Module (RIM)* that maintains full control of the device (e.g., always holds exclusive access to the device through the vendor API) and actively screens user requests for configuration, data transfer, etc. System administrators can set access permission on a per-user, per-resource and per-application basis and the RIM manages accesses to the device accordingly. RIMs also provide the additional advantage of gathering performance and health metrics for its respective device and these statistics are used for improved job scheduling and debug analysis. However, compared to *Option 2*, RIMs do impose additional overhead on the system in terms of access speed to the device but the improved security and other features may be worth the cost for some users. CARMA supports each of the three options where RIMs may be shutdown in order to release their control of the devices (and restarted later) and administrators can configure this policy on a per-node basis.

5.1.6 System Performance and Debug Monitoring

In addition to normalizing device management, CARMA components also normalize the information gathered from special-purpose devices. Such information is used to make job scheduling decisions and for application debug and system health analysis. Figure 5-4 shows the major components of the monitoring system that are active on each node. A *System Information*

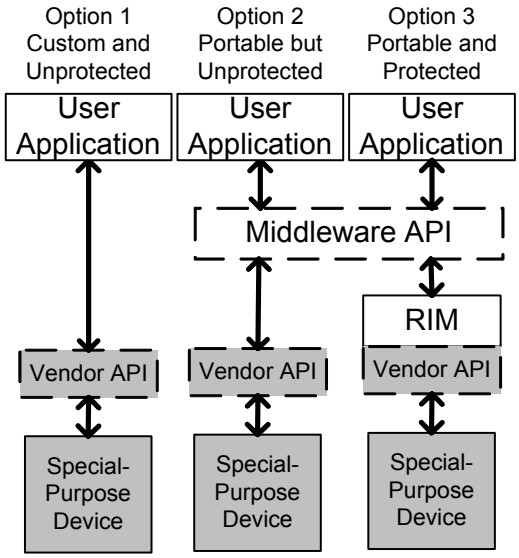


Figure 5-3. CARMA Resource Interface Options

Database is the core repository for gathering local and remote resource information. *Local Resource Managers* coordinate information gathered from probes in their node including the dynamic discovery of special-purpose devices and pass this information to the database. Information is gathered from *Remote Resource Managers* (i.e., on other nodes) through a messaging system and is then updated in the database. Also resource information updates are "pushed" to remote managers via the messaging system when appropriate. Great care must be taken to ensure the monitoring and distributed messaging system are scalable and do not become too burdensome to the system. CARMA employs the Gossip-Enabled Monitoring Service (GEMS) to ensure fast and efficient monitoring of distributed resources and more information on this tool is provided in a previous publication [25]. Also, an analysis of monitoring overhead is given in the next section. Beyond their use in distributed job scheduling decisions as previously described, application data and statistics gathered in the information database are subsequently used by various monitors and agents. These components aide users and administrators debug

applications and analyze the health and performance of the system. More information is provided in the next section.

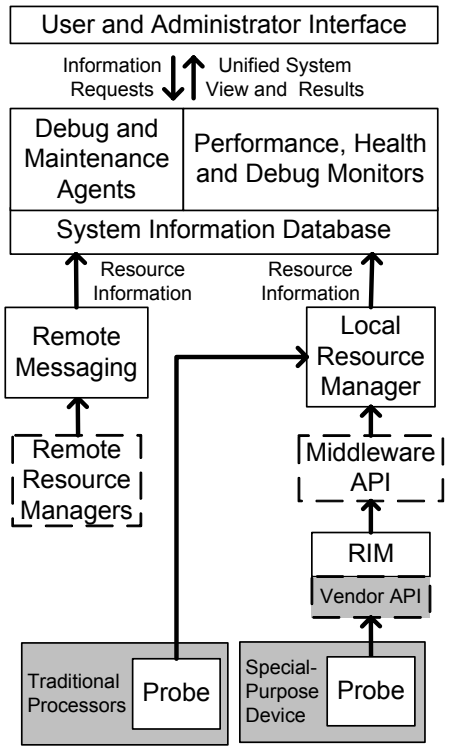


Figure 5-4. CARMA Resource Management and Monitoring

5.2 CARMA Deployment Details and Performance Analysis

CARMA is a framework that eases the integration of all manner of special-purpose devices into traditional high-performance computational systems. The initial deployment of the infrastructure focuses on hardware-adaptable Reconfigurable Computing (RC) devices. Several HPC platforms have demonstrated impressive performance improvements by incorporating RC coprocessors, but their effectiveness as production-quality machines has been limited due to a lack of a comprehensive programming model and runtime management system among other factors. As an example, all compilation and runtime management tools built for RC systems to date are based on the assumption of a single user executing individual tasks on a small-scale RC platform in an interactive fashion. Such tools inherently lack scalability, fault tolerance and are

typically designed custom for a vendor's particular device. Other special-purpose devices such as graphical-processing units have a well-defined interface, programming model and role in the system hierarchy. Due to this fundamental limitation, RC devices were chosen as the first focus area because they are seen as being the most critical and challenging of all potential coprocessor devices to integrate. It is assumed that a framework that can support both traditional processors and RC devices can be easily extended to support any device that lies between these two extremes. The remainder of this section presents an analysis of several of the key components and services that allow CARMA to ease the integration of RC devices into HPC clusters and describes the development and deployment activities that have been undertaken to date.

5.2.1 Deployment Details

The CARMA project has grown out of a necessity to ease the development, use and management of RC HPC systems at the University of Florida's High-performance Computing and Simulation (HCS) research laboratory. The HCS lab has a 300-node Beowulf cluster with a heterogeneous and growing collection of RC devices that it hopes to integrate into a unified development and runtime system. There are currently twelve different board types from eight different vendors in the system and executing applications with these unique resources today is no easy task. Without a suitable alternative, CARMA and other related projects are attempting to create a stopgap to meet the management needs of large-scale RC systems and also explore the potential roadblocks that must be overcome to produce a fully-featured commercial JMS system that will one day provide ubiquitous access to multi-paradigm machines.

To date, CARMA has been deployed on a 32-node cluster with 2.4GHz Intel Xeon processors interconnected with Gigabit Ethernet. The cluster is outfitted with a total of four boards with only two from the same vendor and jobs have been executed on each platform through the mixed deployment philosophy described in Section 5.1.5 (namely Options 2 and 3).

The group is working toward integrating more platforms and refining the design and functionality of the middleware service with feedback from the growing NSF Center for High-performance Reconfigurable Computing (CHREC) and OpenFPGA consortia. A few application examples are highlighted in a previous publication [5]. A centralized version of the job management framework was developed for space environments as an offshoot of the CARMA project and will be deployed on a payload processing cluster aboard a spacecraft to be deployed on an upcoming NASA mission in 2009 as described in Chapter 3. A centralized approach was developed due to the relatively limited processing resources available in the cluster. In addition to the basic features provided by CARMA, the middleware's fault tolerance was greatly improved in the spacecraft version with additional checkpointing, agent progress counters and redundancy. The fault tolerance of the system was demonstrated with a battery of fault experiments including removing power from several of the cluster nodes while applications were executing. A future goal is also to migrate some of these features back to the distributed version. The remainder of this section focuses on the performance of the fully-distributed version of CARMA and overviews JM scheduling options and presents an analysis of the system's scalability.

5.2.2 Job Scheduling Study

The performance of several heuristics that can be used to schedule application tasks on parallel RC systems was analyzed using a custom discrete-event simulator. The heuristics compared in this work include Opportunistic Load Balancing (OLB), Minimum Execution Time (MET), Minimum Completion Time (MCT), and Switching Algorithm (SA). These heuristics have been studied widely in the literature for general-purpose systems but not RC systems [97]. We used typical HPC applications and RC resources as representative cases for a performance model used in the simulation by our scheduling heuristics to schedule the tasks by predicting the

overall execution time of tasks on RC resources. More information on the simulation analysis and algorithms used in the study is provided in a previous publication [98]. Among the inline scheduling heuristics examined, MCT and OLB have similar performance unless the system is nearly fully utilized. When the system is very heavily loaded, MCT outperforms OLB, however such high loads may not be realistic. In other cases, the performance of OLB equals MCT. MET performs poorly in all trials. SA which switches between MCT and MET has a performance better than MET but not as good as MCT and OLB. Based on this preliminary analysis, the CARMA job scheduler uses opportunistic load balancing because the MCT algorithm is more complex than OLB and does not provide a sufficient performance improvement to justify the added complexity.

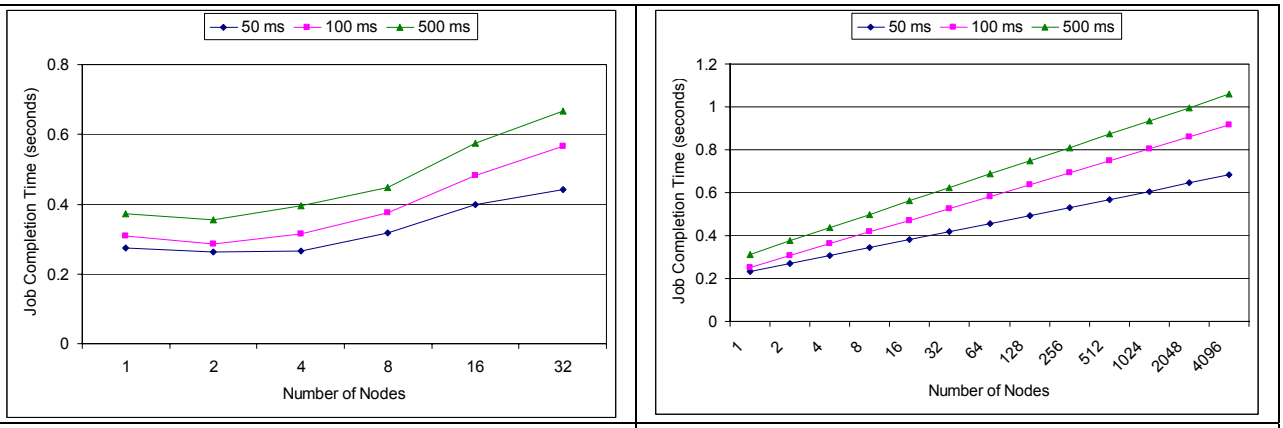
5.2.3 Scalability Analysis

To test the scalability of CARMA, the overhead imposed by the middleware was examined as system size increases. A set of 100 serial jobs are submitted (nearly instantaneously) to the JM on one CARMA node (to create a bottleneck) in the 32-node cluster described in Section 5.2.1. The jobs included in these tests were synthetic in nature and only performed a simple print to a file in order to ensure the overhead versus job execution time was at a maximum, thus stressing the system to the greatest extent. Tests were originally conducted using real serial and parallel jobs but were found to stress the monitoring and scheduling system much less because resources were occupied for longer periods of time. Also, serial jobs were found to be more taxing on the system than executing an equivalent number of parallel tasks in a single job because the scheduling and some of the deployment overhead is paid as a one-time penalty for any job regardless of the number of tasks. The results of these experiments therefore show the worst-case performance numbers for the middleware. Figure 5-5a shows the average job completion time for synthetic jobs executing on the 32-node cluster. Since these jobs do not

perform any meaningful work, the completion time is actually the maximum overhead latency required to schedule jobs on a cluster of a given size. The results show the job scheduling, deployment and cleanup overhead to be linearly bound (note the logarithmic scale on this and all figures) and therefore scalable. In order to further demonstrate the scalability of the service, the linear trend in the job completion values found in the previous experiments were linearly scaled up to a 4096-node cluster as shown in Figure 5-5b. These results show CARMA would likely provide a good job scheduling and deployment performance for large-scale systems due to the distributed nature of the infrastructure and the scalability of the GEMS monitoring system.

CARMA's impact on system resources was also investigated. In order to reduce processor utilization, CARMA agents periodically "sleep" for a set period of time. However, if these "sleep" durations are set too long, then agent response times become significantly delayed leading to potential performance penalties. A balance between response time and overhead imposed on the system must be struck and agent sleep times control this tradeoff. The optimal value for these sleep times may vary based on system performance and size, so it must be determined for each particular system. The effects of setting the agent sleep times to 50ms, 100ms and 500ms are also shown in Figure 5-5. The job completion times show a performance improvement when the agent sleep time is reduced but with a resultant increase in processor utilization (see Figure 5-6). For completeness, a sleep time of 25ms was also used but found to be nearly identical to the 50ms case (with an increase in processor utilization).

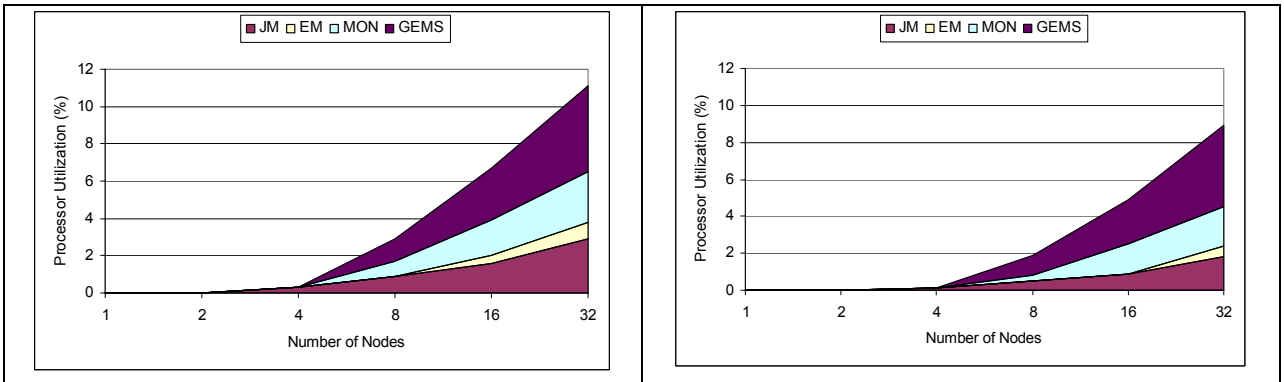
Figure 5-6 shows per-node processor utilization to be linearly bounded and thus scalable as confirmed by the projected values in Figure 5-7. Only agents that significantly contribute to the processor utilization are shown including the job manager (JM), execution manager (EM), resource performance monitor (MON) and GEMS. Future work will investigate ways to reduce



a) Measured | b) Projected

Figure 5-5. Job Deployment Overhead versus System Size

overhead by augmenting or replacing GEMS and expanding upon previous research into scalable device reconfiguration [99] and hardware probe design [100]. However, the results show the framework to be sound and likely to be scalable with relatively small imposed overhead. It should be noted that memory utilization for CARMA was observed to be roughly fixed at 5MB per node for all trials with the vast majority attributed to tables within GEMS.

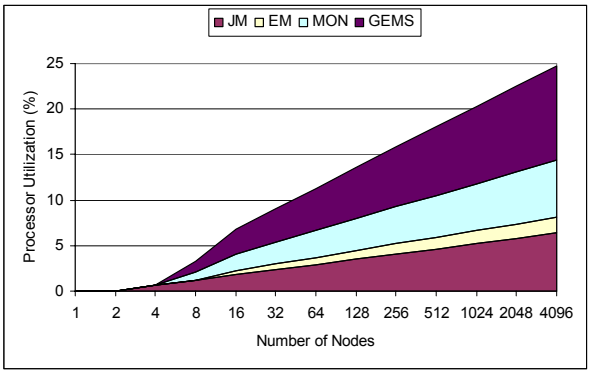


a) Agent Idle Time: 50ms | b) 100ms Agent Idle Time: 100ms

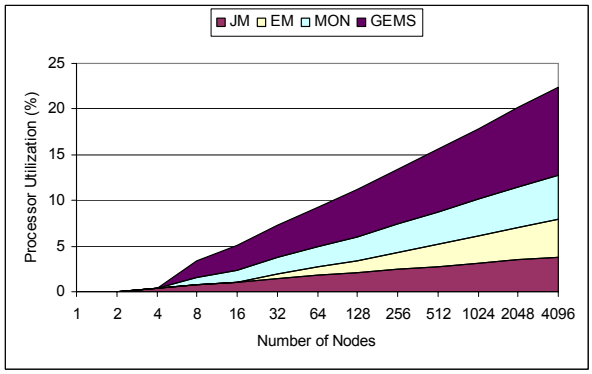
Figure 5-6. Measured Processor Utilization versus System Size

5.3 Conclusions

A trend toward infusing nontraditional and exotic resources into computational clusters has emerged as a way to overcome a predicted leveling off of Moore's Law. However, to achieve the full performance potential of these multi-paradigm HPC systems, special-purpose devices must be afforded as much deference in the system as traditional processors currently enjoy and



a) Agent Idle Time: 50ms



b) Agent Idle Time: 100ms

Figure 5-7. Projected Processor Utilization versus System Size

management systems and tools must be restructured with this goal in mind. To address this critical need, a distributed version of CARMA was investigated, designed and deployed on a Beowulf cluster and a detailed description of the CARMA framework was presented. Also, several experiments were undertaken to analyze the performance and scalability of the middleware.

First and foremost, the CARMA framework is composed of clearly delineated, modular services to provide a standard, systematic approach to building, customizing and distributing dual-paradigm management services, conceptually similar to other RC frameworks. This concept will lead to greater collaboration within the research community and increase extensibility and ease of integration. CARMA simply provides the infrastructure and corresponding services upon which any type of application can be executed while simultaneously incorporating many of the features found in traditional JMS tools. As noted earlier, there has been a good deal of projects working to build programming models, design capture tools and compilers for dual-paradigm systems but a scalable, fault-tolerant, fully-featured run-time management service is sorely lacking.

Several scheduling heuristics that can be used to schedule application tasks on parallel RC systems were analyzed to determine the best option for the CARMA JM. Based on this

preliminary analysis, the CARMA job scheduler uses opportunistic load balancing because the MCT algorithm is more complex than OLB and does not provide a large enough performance improvement to justify the added complexity. The overhead imposed by CARMA was examined as system size increases to test the scalability of the middleware. The results show the job scheduling, deployment and cleanup overhead to be linearly bounded and therefore scalable. In order to further demonstrate the scalability of the service, the job completion time values were scaled up to a 4096-node cluster and these results show CARMA provides a good job scheduling and deployment performance for large-scale systems. Processor utilization was also found to be linearly bounded and scalable, while memory utilization was found to be 5MB for all trials with most of the overhead attributable to GEMS.

Future work for the CARMA project includes building upon the foundation provided by the developed system and expanding its depth, scope and number of platforms supported. Feedback from the user and vendor community will be critical and will be incorporated through the CHREC and OpenFPGA consortia. More specifically, the fault tolerance features found in the centralized version will be migrated into the distributed version and expanding the user-configurable features in the performance monitoring portion of CARMA will be paramount to improve its usability. Also, ways to reduce processor utilization by augmenting or replacing GEMS and expanding upon previous research into scalable device reconfiguration methods will be investigated.

CHAPTER 6
CONCLUSIONS

In this dissertation, the background for reconfigurable computing, dual-paradigm systems and job management services has been studied. The CARMA framework and infrastructure has been presented and evaluated to meet the unique runtime management requirements of embedded systems and HPC clusters. A description of the architectural limitations and requirements for each of these two processing environments has been described as well as a brief description of how CARMA meets these requirements.

Due to recent successes in dual-paradigm system deployment, numerous researchers are beginning to use them to solve challenging HPC applications. While there has been much research and development to build application-mapping technology for programming dual-paradigm systems, robust and scalable job and resource management services, cognizant of both types of resources (i.e., traditional processors and FPGAs), are sorely lacking. There have been a few notable designs and simplistic tools developed to date with various limitations but a comprehensive management infrastructure providing a full-featured set of services has not been investigated, developed and evaluated. Therefore the study, design, development and analysis of CARMA on select HPC dual-paradigm systems is a novel, timely and necessary endeavor.

Several scholarly contributions have been made by developing and analyzing the first comprehensive framework to date for job and resource management and services. CARMA is a first step toward developing an open standard with a modular design that will spur inventiveness and foster industry, government and scholarly collaboration much like Condor, Globus and the Open Systems Interconnect model have done for the cluster, grid and networking environments. Developing and evaluating components to fill out the CARMA infrastructure provided key insight into how future dual-paradigm runtime management services should be structured. This

insight was derived by testing the CARMA concept and design feasibility, refining components based on trial and error experiments and analysis to optimize the framework by analyzing strategies for reducing overhead and increasing performance, fault tolerance and functionality. In addition, deploying initial prototypes with baseline services will open HPC/RC systems to a wider community of users, shorten the RC development learning curve, increase the utilization and usability of existing and future systems, reduce the cost of integrating future RC technology, and enable a wide community of users to share in future design and development efforts. CARMA is meant to be an open standard with a modular design to promote flexibility and expandability in order to foster collaboration in the RC and traditional HPC community. CARMA builds upon a strong research base by combining lessons learned from years of traditional JMS research and preliminary concepts developed to date for RC systems. The CARMA project focuses on the needs of a wide range of users while not trying to maintain a "one-size-fits-all" design. CARMA also attempts to remain vendor agnostic to ensure heterogeneity, ubiquity and continued innovation.

For this dissertation, the CARMA framework was first developed, deployed, and evaluated in an embedded, space-computing environment. CARMA was adapted to meet the needs of a particular NASA satellite mission that seeks to replace custom components and instead deploy more inexpensive components that require improved software management. Individual CARMA services for the embedded-computing version increased or diminished in importance as dictated by the environment's requirements. Directed fault injection using NFTAPE has been performed to test the DM system's job replication and ABFT features. Nominal availability numbers (i.e., assuming a failure occurred when an application required the failed resource) of around 99.9963% have been determined to be a representative value via these analyses. Any

shortcomings discovered during these fault injection tests have been addressed and, after all repairs, the DM system has successfully recovered from every fault injection test performed to date. Tradeoff studies were undertaken to determine the optimal deployment of CARMA services and the middleware's viability in terms of scalability, response time and overhead were analyzed using benchmark applications. A balance between response time and overhead imposed on the system must be struck and the results suggest an agent sleep time of 50ms and 100ms for the FTM and JM respectively is optimal for large-scale systems. The results demonstrate the scalability of the DM middleware to 100 nodes and beyond with the overhead imposed on the system by the DM middleware scaling linearly with system size.

Also, the embedded version of CARMA was extended in another phase of this dissertation to further improve the overall fault tolerance of the system by including autonomic computing techniques. The system was then further analyzed and the adaptable meta scheduler was shown to provide additional fault-tolerance improvements for several realistic missions. The adaptable deployment scheme provides the most performance improvement (22.5% over a fixed scheme) when the spacecraft is subjected to high levels of radiation and also must quickly adapt to the radiation environment. Also, the DM system allows computation to continue through the SAA where most electronic systems must shutdown. CARMA will improve the cost-effectiveness of future spacecraft processing platforms and these savings will translate into more missions undertaken by NASA and thus expand society's understanding of and presence in space.

For the final phase of this research, the CARMA infrastructure was adapted for the HPC cluster-computing environment. This environment is typically marked by nodes composed of moderate- to high-performance processors, limited per-node storage capacity, and typically interconnected with a message-passing, switched network. Clusters are best suited for general-

purpose and scientific applications with moderate computation and communication requirements. Therefore the CARMA framework was augmented to be fully distributed to maximize the system's fault tolerance and scalability. A tradeoff analysis was undertaken to assess and quantify the middleware's imposed system and application overhead and scalability. The projected results show the middleware's overhead to be linearly bound, imposing an application performance overhead of less than 1 second and a processor utilization of less than 25% when scheduling individual tasks on 4000 nodes. The experiments demonstrated the design's viability and provided key insight to direct future infrastructure research. This research will be extended in the future through a standardization effort and this project will impact the usability and cost-effectiveness of future large-scale HPC systems.

Future research can proceed in several specific directions. As CARMA is meant to be an initial step toward providing insight into how best to develop a dual-paradigm system, application and resource manager, additional research and development will be required to develop a "production-level" version of the framework, infrastructure and related services. To extend the framework's supported platforms, CARMA's RIM and other components can be developed for other systems mentioned in this dissertation (e.g., Cray's XD1) or on future systems not yet developed. An expansion onto many more platforms will examine the framework's ability to provide hardware virtualization and simultaneously provide the benefits of the infrastructure to a broader community of researchers. Also, the initial set of components and features deployed in the initial versions will likely be improved upon to provide additional functionality in the future. In addition, once the initial framework and prototypes are deployed and shared with the community, other researchers will likely collaborate on modules for their

unique needs. Furthermore, CARMA should be deployed on large-scale production systems to experimentally validate the infrastructure's scalability.

LIST OF REFERENCES

[1] M. Frank, "Physical Limits of Computing," *Computing in Science and Engineering*, **4**(3), May/June 2002, pp. 16-25.

[2] D. Frank, "Power-constrained CMOS scaling limits," *IBM Journal of Research and Development in Computers & Technology*, **46**(2/3), February 2002, pp. 235-244.

[3] C. Hsu and W. Feng, "A Power-Aware Run-Time System for High-Performance Computing," *Proc. Supercomputing*, Seattle, WA, November 12-18, 2005.

[4] E. DeBenedictis, "Will Moore's Law Be Sufficient?," *Proc. Supercomputing*, Pittsburg, PA, November 6-12, 2004.

[5] B. Holland, J. Greco, I. Troxel, G. Barfield, V. Aggarwal and A. George, "Compile- and Run-time Services for Distributed Heterogeneous Reconfigurable Computing," *Proc. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 26-29, 2006.

[7] J. Kahle, M. Day, P. Hofstee, C. Johns, T. Maeurer and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development in Computers & Technology*, **49**(4/5), April 2005, pp. 589-604.

[8] S. Williams, J. Shalf, L. Loiker, S. Kamil, P. Husbands and K. Yelick, "The Potential of the Cell Processor for Scientific Computing," *Proc. Conference on Computing Frontiers (CF)*, Ischia, Italy, May 3-5, 2006.

[9] M. Blom and P. Follo, "VHF SAR Image Formation Implemented on a GPU," *Proc. International Geoscience and Remote Sensing Symposium (IGARSS)*, Seoul, South Korea, July 25-26, 2005.

[10] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU," *Proc. Conference on Information Visualization (IV)*, London, England, July 4-7, 2006.

[11] J. Bolz, I. Farmer, E. Grinspun and P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics*, **22**(3), September 2003, pp. 917-924.

[12] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, "GPU Cluster for High Performance Computing," *Proc. Supercomputing*, Pittsburg, PA, November 6-12, 2004.

[13] J. Tripp, A. Hanson, M. Gokhale and H. Mortveit, "Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study," *Proc. Supercomputing*, Seattle, WA, November 12-18, 2005.

[14] D. Fouts, K. Macklin, D. Zulaica and R. Duren, "Electronic Warfare Digital Signal Processing on COTS Computer Systems with Reconfigurable Architectures," *AIAA Journal of Aerospace Computing, Information, and Communication*, **2**(10), October 2005.

[15] P. Faes, B. Minnaert, M. Christiaens, E. Bonnet, Y. Saeys, D. Stroobandt and Y. Van de Peer, "Scalable Hardware Accelerator for Comparing DNA and Protein Sequences," *Proc. Conference on Scalable Information Systems (InfoScale)*, Hong Kong, China, May 30-June1, 2006.

[16] S. Fahmy, P. Cheung and W. Luk, "Hardware Acceleration of Hidden Markov Model Decoding for Person Detection," *Proc. Design Automation, and Test in Europe Conference and Exhibition (DATE)*, Munich, Germany, March 7-11, 2005.

[17] J. Beeckler and W. Gross, "FPGA Particle Graphics Hardware," *Proc. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April 18-20, 2005.

[18] B. Cope, P. Cheung, W. Luk and S. Witt, "Have GPUs made FPGAs redundant in the field of Video Processing?," *Proc. Conference on Field-Programmable Technology (FPT)*, National University of Singapore, Singapore, December 11-14, 2005.

[19] S. Mohanty and V. Prasanna, "A Hierarchical Approach for Energy Efficient Application Design Using Heterogeneous Embedded Systems," *Proc. Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, San Jose, CA, October 30-November 1, 2003.

[20] V. Ross, "Heterogeneous HPC Computing," *Proc. GOvernment Microcircuits Applications and Critical TECHnologies (GOMACTech)*, Tampa, FL, March 31-April 3, 2003.

[21] T. El-Ghazawi, K. Gaj, N. Alexandridis, F. Vroman, N. Nguyen, J. Radzikowski, P. Samipagdi and S. Suboh, "A Performance Study of Job Management Systems," *Journal of Concurrency: Practice and Experience*, **16**(13), October 2004, pp. 1229-1246.

[22] Altair Grid Technologies, 2003, "OpenPBS," http://www.openpbs.org/main.html (last accessed October 2006).

[23] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su and D. Zagorodnov, "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. on Parallel and Distributed Systems*, **14**(4), April 2003, pp. 369-382.

[24] R. Wolski, N. Spring and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems*, **15**(5-6), October 1999, pp. 757-768.

[25] R. Subramaniyan, P. Raman, A. George and M. Radlinski, "GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems," *Cluster Computing Journal*, **9**(1), January 2006, pp. 101-120.

[26] D. Thain, T. Tannenbaum and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Journal of Concurrency and Computation: Practice and Experience*, **17**(2-4), February-April 2005, pp. 323-356.

[27] Platform Computing Corporation, *Load Sharing Facility: User's and Administrator's Guide*, User's Guide, Toronto, Canada, 1994.

[28] L. Revor, *DQS Users Guide*, User's Guide, Argonne National Laboratory, Argonne, IL, September 1992.

[29] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel and B. Toonen, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus," *Proc. Supercomputing*, Denver, CO, November 10-16, 2001.

[30] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, **11**(2), February 1997, pp. 115-128.

[31] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, **34**(2), June 2002, pp. 171-210.

[32] A. Staicu, J. Radzikowski, K. Gaj, N. Alexandridis and T. El-Ghazawi, "Effective Use of Networked Reconfigurable Resources," *Proc. International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, Laurel, MD, September 12-13, 2001.

[33] K. Gaj, T. El-Ghazawi, N. Alexandridis, J. Radzikowski, M. Taher and F. Vroman, "Effective Utilization and Reconfiguration of Distributed Hardware Resources Using Job Management Systems," *Proc. Reconfigurable Architecture Workshop (RAW)*, Nice, France, April 22, 2003.

[34] J. Lehrter, F. Abu-Khzam, D. Bouldin, M. Langston and G. Peterson, "On Special-Purpose Hardware Clusters for High-Performance Computational Grids," *Proc. International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Cambridge, MA, November 4-6, 2002.

[35] D. Thomas and W. Luk, "Framework for Development and Distribution of Hardware Acceleration," *Proc. International Society for Optical Engineering (SPIE)*, **4867**, 2002, pp. 48-59.

[36] N. Shirazi, W. Luk, P. Cheung, *Run-Time Management of Dynamically Reconfigurable Designs*, in R. Hartensteing and A. Keevallik, eds., *Field-Programmable Logic and Application*s, Springer, NewYork, NY, 1998, pp. 59-68.

[37] J. Burns, A. Donlin, J. Hogg, S. Singh and M. de Wit, "A Dynamic Reconfiguration Run-Time System," *Proc. IEEE Symposium on Field-programmable Custom Computing Machines (FCCM)*, Napa Valley, CA, April 16-18, 1997.

[38] J. Morrison, P. O'Dowd and P. Healy, "Searching RC5 Keyspaces with Distributed Reconfigurable Hardware," *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, Nevada, June 23-26, 2003.

[39] J. Morrison, P. O'Dowd and P. Healy, "LinuxNOW: A Peer-to-Peer Metacomputer for the Linux Operating System," *Journal of Parallel and Distributed Processing Techniques and Applications*, (In press).

[40] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot and E. Komp, "The Case for High Level Programming Models for Reconfigurable Computers," *Proc. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 26-29, 2006.

[41] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," *Proc. IEEE Symposium on Field-programmable Custom Computing Machines (FCCM)*, Napa, CA, April 20-23, 1999.

[42] Honeywell Technology Center, *An Integrated Co-design Environment for Heterogeneous Configurable Computing Systems*, White Paper, Minneapolis, MN, 1999.

[43] P. Sundararajan, S. Guccione and D. Levi, "XHWIF: A Portable Hardware Interface for Reconfigurable Computing," *Proc. Reconfigurable Technology Session at ITCom*, Denver, CO, August 19-24, 2001.

[44] D. Lahn, R. Hudson and P. Athanas, "Framework for Architecture-Independent Run-Time Reconfigurable Applications," *Proc. International Society for Optical Engineering (SPIE)*, **4212**, November 2000, pp. 162-172.

[45] E. Caspi, M. Chu and R. Huang, *Scheduling on a Reconfigurable Processor with Virtual Pages*, Technical Report, UC Berkeley, Berkeley, CA, 2001.

[46] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek and A. DeHon, "Analysis of Quasi-Static Scheduling Techniques in a Virtualized Reconfigurable Machine," *Proc. ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, February 24-26, 2002.

[47] R. Maestre, F. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh and H. Singh, "A Framework for Reconfigurable Computing: Task Scheduling and Context Management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **9**(6), December 2001.

[48] K. Purna and D. Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers," *IEEE Transaction on Computers*, **48**(6), June 1999.

[49] S. Hauck and G. Borriello, "Logic Partition Orderings for Multi-FPGA Systems," *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, February 12-14, 1995.

[50] International Organization for Standardization, *Open Systems Interconnection -- Basic Reference Model*, Standard ISO/IEC 7498-1, Geneva, Switzerland, 1994.

[51] M. Griffin, *NASA 2006 Strategic Plan*, National Aeronautics and Space Administration, NP-2006-02-423-HQ, Washington, DC, February, 2006.

[52] J. Samson, J. Ramos, I. Troxel, R. Subramaniyan, A. Jacobs, J. Greco, G. Cieslewski, J. Curreri, M. Fischer, E. Grobelny, A. George, V. Aggarwal, M. Patel and R. Some, "High-Performance, Dependable Multiprocessor," *Proc. of IEEE/AIAA Aerospace Conference*, Big Sky, MT, March 4-11, 2006.

[53] D. Dechant, "The Advanced Onboard Signal Processor (AOSP)," *Advances in VLSI and Computer Systems*, **2**(2), October 1990, pp. 69-78.

[54] M. Iacoponi and D. Vail, "The Fault Tolerance Approach of the Advanced Architecture On-Board Processor," *Proc. Symposium on Fault-Tolerant Computing*, Chicago, IL, June 21-23, 1989.

[55] F. Chen, L. Craymer, J. Deifik, A. Fogel, D. Katz, A. Silliman Jr., R. Some, S. Upchurch and K. Whisnant, "Demonstration of the Remote Exploration and Experimentation (REE) Fault-Tolerant Parallel-Processing Supercomputer for Spacecraft Onboard Scientific Data Processing," *Proc. International Conference on Dependable Systems and Networks (ICDSN)*, New York, NY, June 2000.

[56] K. Whisnant, R. Iyer, Z. Kalbarczyk, P. Jones III, D. Rennels and R. Some, "The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications," *IEEE Transactions on Software Engineering*, **30**(4), April 2004, pp. 257-277.

[57] J. Williams, A. Dawood and S. Visser, "Reconfigurable Onboard Processing and Real Time Remote Sensing," *IEICE Trans. on Information and Systems, Special Issue on Reconfigurable Computing*, **E86-D**(5), May 2003, pp. 819-829.

[58] J. Williams, N. Bergmann, and R. Hodson, "A Linux-based Software Platform for the Reconfigurable Scalable Computing Project," *Proc. International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, Washington, DC, September 7-9, 2005.

[59] M. Bertier, O. Marin and P. Sens, "A Framework for the Fault-Tolerant Support of Agent Software," *Proc. Symposium on Software Reliability Engineering (ISSRE)*, Boulder, CO, November 17-20, 2003.

[60] M. Li, D. Goldberg, W. Tao and Y. Tamir, "Fault-Tolerant Cluster Management for Reliable High-performance Computing," *Proc. International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Anaheim, California, August 21-24, 2001.

[61] E. Prado, P. Prewitt and E. Ille, "A Standard Approach to Spaceborne Payload Data Processing," *IEEE Aerospace Conference*, Big Sky, Montana, March 2001.

[62] S. Fuller, *RapidIO - The Embedded System Interconnect*, John Wiley & Sons, Hoboken, NJ, January 2005.

[63] Ian Troxel, Eric Grobelny and Alan D. Geroge, "System Management Services for High-Performance In-situ Aerospace Computing," *AIAA Journal of Aerospace Computing, Information and Communication* (In press).

[64] D. Feitelson, and L. Rudolph. "Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control," *Journal of Parallel and Distributed Computing*, **35**(1), May 1996, pp. 18-34.

[65] G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovsky, and J. Dongarra, "Fault Tolerant Communication Library and Applications for HPC," *Los Alamos Computer Science Institute (LACSI) Symposium*, Santa Fe, NM, October 27-29, 2003.

[66] Message Passing Interface Forum, *MPI: a message-passing interface standard*, Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1, 1994.

[67] R. Subramaniyan, V. Aggarwal, A. Jacobs, and A. George, "FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems ," *Proc. International Conference on Embedded Systems and Applications (ESA)*, Las Vegas, NV, June 26-29, 2006.

[68] J. Villarreal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar, "Improving Software Performance with Configurable Logic," *Journal of Design Automation for Embedded Systems*, **7**(4), November 2002, pp. 325-339.

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

[70] K. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Transactions on Computers*, **C-33**(6), June 1984, pp. 518-528.

[71] D. Scott, P. Jones, M. Hamman, Z. Kalbarczyk, and R. Iyer, "NFTAPE: Networked Fault Tolerance and Performance Evaluator," *Proc. International Conference on Dependable Systems and Networks (DSN)*, Bethesda, MD, June 23-26, 2002.

[72] E. Frachtenberg, D. Feitelsoná, F. Petrini, and Juan Fern andez, "Adaptive Parallel Job Scheduling with Flexible CoScheduling," *IEEE Transactions on Parallel and Distributed Systems*, **11**(16), November 2005, pp. 1066-1077.

[73] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, C. Conger, J. Curreri, and A. George, "Hardware/software Interface for High-performance Space Computing with FPGA Coprocessors," *Proc. IEEE Aerospace*, Big Sky, MN, March 4-11, 2006.

[74] M. Parashar, S. Hariri, "Autonomic Computing: An Overview," *Proc. International Workshop on Unconventional Programming Paradigms (UPP)*, Le Mont Saint Michel, France, September 15-17, 2004.

[75] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, "A Concise Introduction to Autonomic Computing," *Journal of Advance Engineering Informatics*, **19**(3), 2005, pp. 181-187.

[76] R. Sterritt, D. Bustard, and A. McCrea, "Autonomic Computing Correlation for Fault Management System Evolution," *Proc. Conference on Industrial Informatics (INDIN)*, Banff, Canada, August 20-24, 2003.

[77] K. Breitman and W. Truszkowski, "The Autonomic Semantic Desktop: Helping Users Cope with Information System Complexity," *Proc. International Conference on Engineering of Autonomic Systems (EASe)*, Columbia, MD, April 24-28, 2006.

[78] P. Eaton, H. Weatherspoon and J. Kubiatowicz, "Efficiently Binding Data to Owners in Distributed Content-Addressable Storage Systems," *Proc. IEEE Security in Storage Workshop (SISW)*, San Francisco, CA, December 13, 2005.

[79] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank – A Heterogeneous Scalable SAN File System," *IBM Systems Journal*, **42**(2), 2003, pp. 250-267.

[80] R. Detter, L. Welch, B. Pfarr, B. Tjaden, and E. Huh, "Adaptive Management of Computing and Network Resources for Spacecraft Systems," *Proc. International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, Laurel, MD, September 26-28, 2000.

[81] R. Sterritt, C. Rouff, J. Rash, W. Truszkowski, and M. Hinchey, "Self-* Properties in NASA Missions," *Proc. International Conference on Software Engineering Research and Practice (SERP)*, Las Vegas, NV, June 27-29, 2005.

[82] E. Wyatt, H. Hotz, R. Sherwood, J. Szijjarto and M. Sue, "Beacon Monitor Operations on the Deep Space One Mission," Proc. *Symposium on Space Mission Operations and Ground Data Systems (SpaceOps)*, Tokyo, Japan, June 1-5, 1998.

[83] C. Rouff, M. Hinchey, J. Rash, W. Truszkowski, and R. Sterritt, "Towards Autonomic Management of NASA Missions," *Proc. of International Workshop on Reliability and Autonomic Management in Parallel and Distributed System (RAMPDS) at the International Conference on Parallel and Distributed Systems (ICPADS)*, Fukuoka, Japan, July 20-22, 2005.

[84] S. Curtis, W. Truszkowski, M. Rilee, and P. Clark, "ANTS for the Human Exploration and Development of Space," *Proc. IEEE Aerospace Conference*, Big Sky, MT, March 8-15, 2003.

[85] J. Baldassari, C. Kopec, E. Leshay, W. Truszkowski, and D. Finkel, "Autonomic Cluster Management System (ACMS): A Demonstration of Autonomic Principles at Work," *Proc. IEEE International Conference on the Engineering of Computer-Based Systems (ECBS)*, Greenbelt, MD, April 4-7, 2005.

[86] R. Sterritt and D. Bustard, "Autonomic Computing – a Means of Achieving Dependability?," *Proc. IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, Huntsville, Alabama, April 7-11, 2003.

[87] G. Badhwar, "Radiation Measurements on the International Space Station," *Proc. of International Workshop on Space Radiation Research at the Annual NASA Space Radiation Health Investigators' Workshop*, Arona, Italy, May 27-31, 2000.

[88] M. Sirianni and M. Mutchler, "Radiation Damage in HST Detectors," *Proc. of Scientific Detector Workshop (SDW)*, Taormina, Italy, June 19-25, 2005.

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

[90] G. Swift, *Virtex-II Static SEU Characterization*, Xilinx, Inc., Single Event Effects Consortium Report, San Jose, CA, January 2004.

[91] P. Saganti, F. Cucinotta, J. Wilson, L. Simonsen, and C. Zeitlin, "Radiation Climate Map for Analyzing Tisks to Astronauts on the Mars Surface from Galactic Cosmic Rays," *Space Science Reviews*, **110**(1), 2004, pp. 143-156.

[92] I. Troxel, A. Jacob, A. George, R. Subramaniyan and M. Radlinski, "CARMA: A Comprehensive Management Framework for High-Performance Reconfigurable Computing," *Proc. International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, Washington, DC, September 8-10, 2004.

[93] S. Lampoudi and D. Beazley, "SWILL: A Simple Embedded Web Server Library," *Proc. USENIX Annual Technical Conference*, Monterey, CA, June 10-15, 2002.

[94] Q. Hua and Z. Chen, "Efficient Granularity and Clustering of the Directed Acyclic Graphs," *Proc. Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Chengdu, China, August 27-29, 2003

[95] D. Feitelson, and L. Rudolph. "Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control," *Journal of Parallel and Distributed Computing*, **35**(1), May 1996, pp. 18-34.

[96] G. Winters and T. Teorey, "Challenges in Distributed Systems: Managing Heterogeneous Distributed Computing Systems: Using Information Repositories," *Proc. Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing (CASCON)*, Toronto, Canada, October 24-28, 1993.

[97] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund, "Dynamic Matching and Scheduling of a Class of Independendent Tasks onto Heterogeneous Computing Systems," *Proc. Heterogeneous Computing Workshop*, San Juan, Puerto Rico, April 12, 1999.

[98] R. Subramaniyan, I. Troxel, A. George and M. Smith, "Simulative Analysis of Dynamic Scheduling Heuristics for FPGA-based Reconfigurable Computing of Parallel Applications," *Proc. Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, February 22-24, 2006.

[99] R. DeVille, I. Troxel and A. George, "Performance Monitoring for Run-time Management of Reconfigurable Devices," *Proc. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 27-30, 2005.

[100] A. Jacob, I. Troxel and A. George, "Distributed Configuration Management for Reconfigurable Cluster Computing," *Proc. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, June 21-24, 2004.

BIOGRAPHICAL SKETCH

Mr. Troxel received two B.S. degrees (one in electrical engineering and one in computer engineering) and an M.E. degree in electrical and computer engineering from the University of Florida. Mr. Troxel is a senior research assistant at the High-performance Computing and Simulation (HCS) Research Laboratory and has led the Reconfigurable Computing Research Group focusing on infrastructure challenges and application acceleration for four years. The HCS lab has been cited by the National Security Agency (NSA) as a Center of Research Excellence in High-performance Computing and Networking and the reconfigurable computing work at the HCS lab has recently led to the awarding of the NSF Center for High-performance Reconfigurable Computing (CHREC) with the University of Florida as lead institution.

Mr. Troxel has also completed the Advanced Space Computing Research Group which focuses on a variety of topics related to high-performance embedded computing in space. He has been active on research and development in high-performance architectures, networks, services, and systems for applications in parallel, reconfigurable, embedded, distributed, and fault-tolerant computing for six years for sponsors such as Honeywell, NASA, DoD and Rockwell Collins among others. Mr. Troxel gained extensive knowledge in an industry setting while on internship at Honeywell where he filed two patents on related processing topics.

Mr. Troxel's professional service includes co-organizer for the 2006 and 2007 IEEE Aerospace Conference session on Reconfigurable Computing Technologies and technical committee member for the Military and Aerospace Programmable Logic Devices (MAPLD) Conference since 2004. He has also served as a reviewer for numerous conferences and journals including: IEEE Local Computer Networks (LCN) Conference, the IEEE International Performance, Computing, and Communications Conference (IPCCC), the International Conference on Vector and Parallel Processing (VECPAR), the ACS/IEEE International

Conference on Pervasive Services (ICPS), the International Conference on Parallel Processing (ICPP), *IEEE Transactions on Very Large Scale Integration Systems*, the *Journal of Circuits, Systems, and Computers*, the *Cluster Computing Journal*, the *Journal of Parallel and Distributed Computing (JPDC)*, and the *Journal of Microprocessors and Microsystems*. Mr. Troxel has published two journal and 22 conference papers to date with three more journal submissions pending. Upon graduation, Mr. Troxel will seek employment at an industry, government or academic institution where he may continue cutting-edge computer engineering research.