JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 4, February 2007

System Management Services for High-Performance In-situ Aerospace Computing

Ian Troxel,[*] Eric Grobelny,[†] and Alan D. George[‡]
High-Performance Computing and Simulation (HCS) Research Laboratory,
University of Florida, Gainesville, Florida, 32611-6200

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 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 paper 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.

I. 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.[1] 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

Received 27 July 2006. The results presented in this submission substantially expand upon preliminary results presented in a submission to the 2006 International Conference on Embedded Systems and Applications entitled, "Reliable Management Services for COTS-based Space Systems and Applications." The Dependable Multiprocessor research project is funded in part under NASA NMP ST-8 contract NMO-710209 with partial matching funds from the state of Florida's High-tech Corridor Council.
[*] Ph.D. Candidate in the Electrical and Computer Engineering Department, AIAA Student Member, troxel@hcs.ufl.edu
[†] Ph.D. Student in the Electrical and Computer Engineering Department, grobelny@hcs.ufl.edu
[‡] Professor of Electrical and Computer Engineering, AIAA Member, george@hcs.ufl.edu

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 meet 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.[2] The DM system combines COTS processors and networking components (e.g., Ethernet) with a novel and robust middleware system that provides a means to customize application deployment and recovery features, and thereby maximize system efficiency while maintaining the required level of reliability by adapting to the harsh environment of space. In addition, the 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 paper presents and examines the performance of the DM system's management middleware and associated development and runtime libraries and is divided as follows. Section II presents past projects that have inspired the development of the DM system and other related research. Sections III and IV 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 V, and Section VI presents experimental performance analysis highlighting the scalability of the DM system. Conclusions and future work are highlighted in Section VII.

II. 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.[3] 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.[4] 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.[5] 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.[6] 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.[7] FedSat's payload consisted of a microprocessor, a reprogrammable FPGA, a radiation-hardened, antifuse-based FPGA to perform reconfiguration and configuration scrubbing, and memory. 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.[8] 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 which provides a simple mechanism for deploying replicated applications in an agent-based distributed computing environment.[9] 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.[10] While the central approach taken by the FTCT design reduces system complexity, hot-sparing managers can strain limited system resources and can become a bottleneck if not carefully designed. The Comprehensive Approach to Reconfigurable Management Architecture (CARMA), under development at the University of Florida, aims to make executing FPGA-accelerated jobs in an HPC environment as easy and reliable as it currently is to execute jobs in traditional HPC environments.[11] Elements of each of the past research projects previously discussed have influenced the DM middleware design; in particular, CARMA's ability to seamlessly integrate FPGA devices into the job scheduling system partially formed the basis of the DM scheduler. The next section provides an overview of the DM system architecture with high-level descriptions of the major components.

III. 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 Fig. 1) follows an integrated-payload concept whereby components can be incrementally added to a standard system infrastructure inexpensively.[12] 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.[13] 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.

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

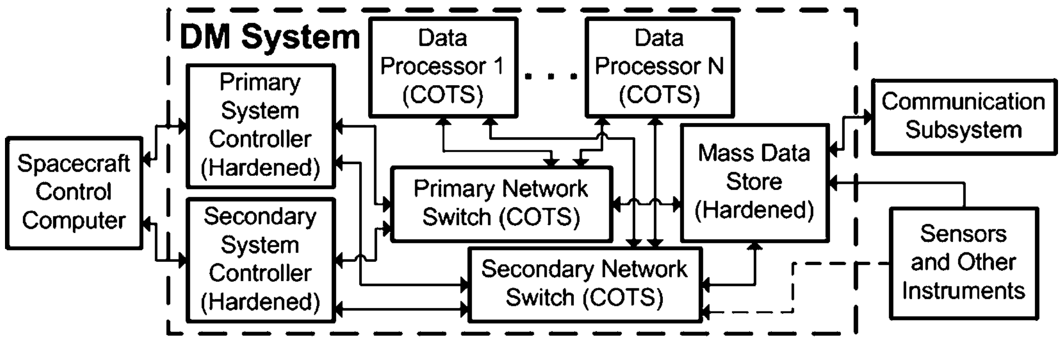

DM System
Data Processor 1 (COTS)
Data Processor N (COTS)
Primary System Controller (Hardened)
Secondary System Controller (Hardened)
Spacecraft Control Computer
Primary Network Switch (COTS)
Secondary Network Switch (COTS)
Mass Data Store (Hardened)
Communication Subsystem
Sensors and Other Instruments

Fig. 1 System hardware architecture of the Dependable Multiprocessor.

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 Mass Data Store, through the cluster, back to the Mass Data Store, 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 these results are presented in Section VI.D. The next section provides a detailed description of the middleware that ensures reliable processing in the DM system.

IV. 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 Fig. 2. 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 Fig. 2). 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.

A. 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.

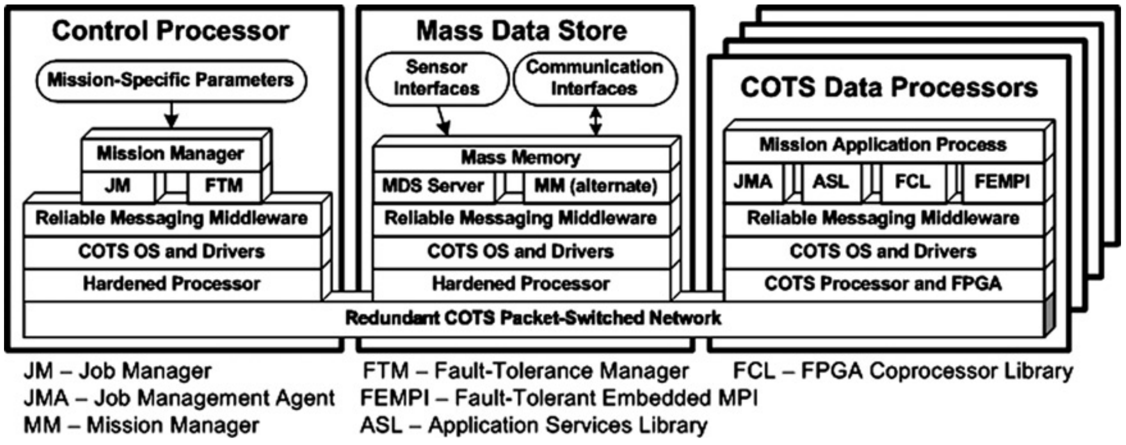

Control Processor
Mission-Specific Parameters
Mission Manager
JM
FTM
Reliable Messaging Middleware
COTS OS and Drivers
Hardened Processor
Mass Data Store
Sensor Interfaces
Communication Interfaces
Mass Memory
MDS Server
MM (alternate)
Reliable Messaging Middleware
COTS OS and Drivers
Hardened Processor
COTS Data Processors
Mission Application Process
JMA
ASL
FCL
FEMPI
Reliable Messaging Middleware
COTS OS and Drivers
COTS Processor and FPGA
Redundant COTS Packet-Switched Network
JM – Job Manager
JMA – Job Management Agent
MM – Mission Manager
FTM – Fault-Tolerance Manager
FEMPI – Fault-Tolerant Embedded MPI
ASL – Application Services Library
FCL – FPGA Coprocessor Library

Fig. 2 System software architecture of the Dependable Multiprocessor.

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 Section IV.B, 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.

B. 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. 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.

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. It is the central software component through which the embedded system provides liveliness information to the spacecraft. The spacecraft control computer (see Fig. 1) 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.

C. 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.[14] In addition, the scheduler optimizes the use of heterogeneous resources such as FPGA accelerators with strategies borrowed from the CARMA runtime job management service.[11] 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 IV.G.

Much like the FTM, the centralized JM employs distributed software agents to gather application liveliness information. The JM also relies upon these 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.[15] 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 Fig. 2) 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.

D. 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 Fig. 2). 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.

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. The MM uses a frame-based scheduling approach to application deployment in addition to enforcing application real-time deadlines. Also, the MM chooses an appropriate replication deployment strategy on a per-application basis based on environmental and system information provided by the FTM. Job replication is handled by the MM in a transparent, yet user-directed manner and replication options include double or triple replication with job processes either spatially or temporally replicated. The MM performs the vote of all job replicas and makes the decision to either rollback to that frame's input or abort the current instance of the job and roll forward to the next frame as defined in the mission description.

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 is represented by an independent executable most often executed on a separate processor. For jobs that include multiple tasks, FEMPI is used for inter-process communication between the tasks (see the next section). The job description file describes all information required to deploy a job to the JM 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 chain of control by which applications are deployed in the system is illustrated in Fig. 3. More information on the system monitoring capability and performance of the MM can be found in.[16] The next several sections describe the various application interface libraries provided by the DM system.

E. 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.[17] 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.

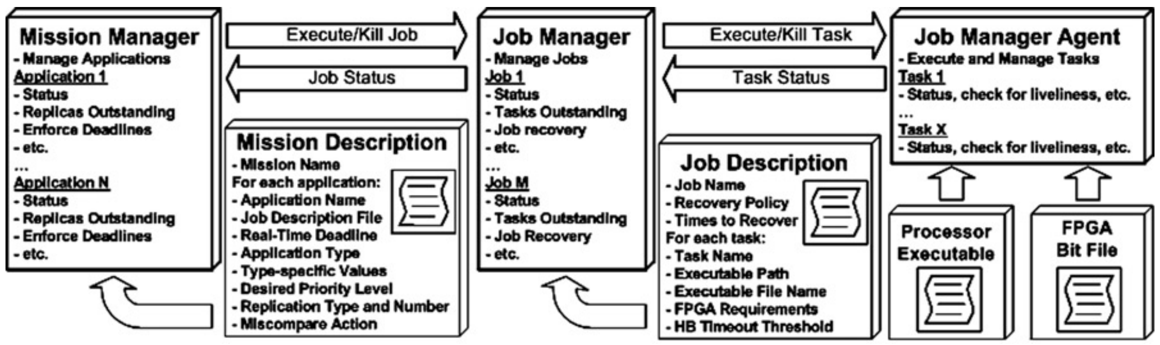


Fig. 3 DM system job deployment and management control flow.

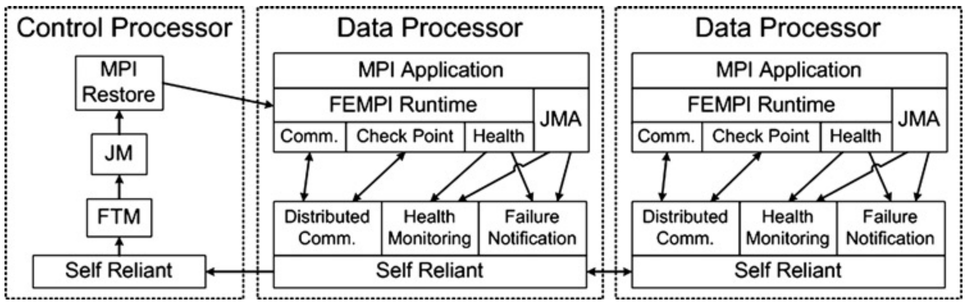


Fig. 4 Interaction between FEMPI and related software components of the Dependable Multiprocessor.

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 Fig. 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.[18]

F. 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.[19] 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 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.[20] 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.[21]

G. MDS Server and Checkpoint Library

The MDS Server process (shown in Fig. 2) 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.

H. 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.[22] 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 $O(n \log n)$.

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.

V. Experimental Setup

To investigate the performance of the 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.

A. Prototype Flight System

The prototype system developed for the current phase of the DM project, as shown in Fig. 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, and redundant Ethernet switches. 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 650 MHz PowerPC processor, memory, and dual 100 Mbps 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.

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

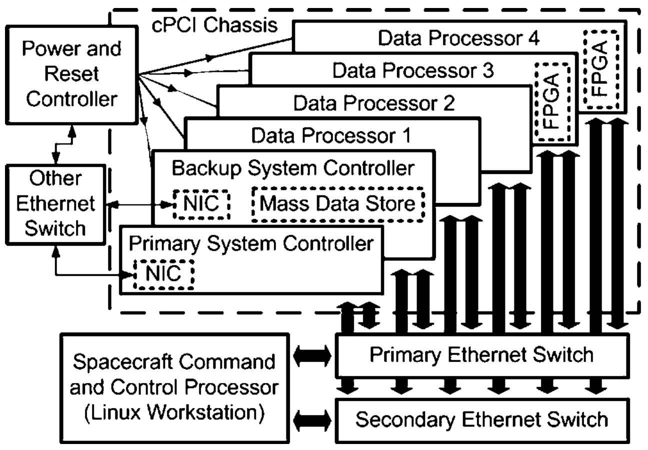


Fig. 5 System configuration of the prototype testbed.

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 the chassis while the system is executing parallel applications.

Various components (shown as boxes with dotted lines in Fig. 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 Fig. 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 40 GB PMC hard drive, while the flight system will likely include a radiation-hardened, solid-state storage device currently under development.

B. 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 2.4 GHz Intel Xeon processors, memory and a Gigabit Ethernet network interface. The DM middleware is 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.

VI. Prototype 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 VI.B and VI.C present and analyze the results of a job deployment and a scalability analysis of the DM software middleware respectively. Also, a case study application that highlights improvements required to ensure the scalability of the flight system's hardware architecture was investigated and quantified and the analysis of these results is presented in Section VI.D.

A. 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 1. All recovery times measure the time from when a fault is injected until the time at which an executing application recovers.

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[23] 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 250 ms 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 Fig. 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 Fig. 6. The results demonstrate that the time for the FTM to recover JMAs under the most stressing

Table 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

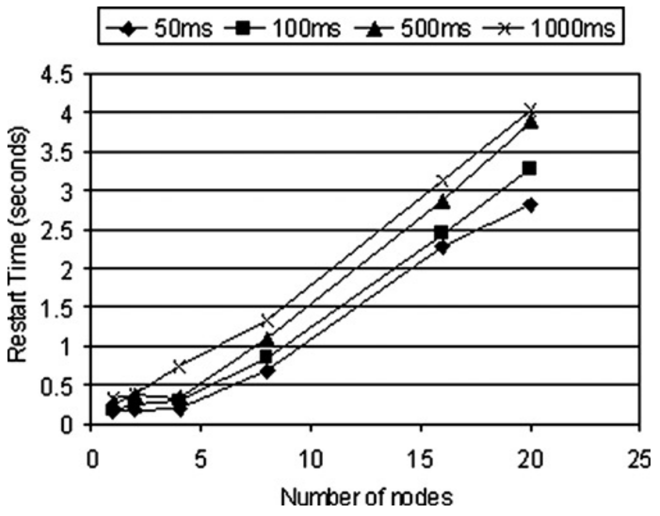


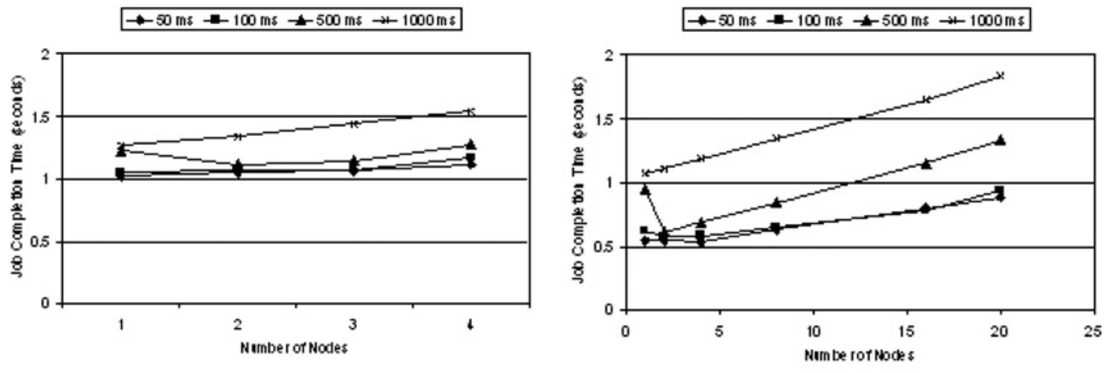
Fig. 6 JMA restart time for several FTM sleep times.

conditions possible is still linearly bound as the number of nodes in the system increases. Also, reducing the sleep time to 50 ms (and therefore decreasing the FTM's average response time) provides roughly a 13% performance improvement over setting the sleep time to 100 ms in the best case (i.e., 20 nodes shown in Fig. 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 25 ms case (not shown). Since the 25 ms FTM sleep time experiment provided approximately the same performance as that seen when setting the sleep time to 50 ms, the optimal FTM sleep time for both the testbed and cluster is 50 ms. 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.

B. 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 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 50 ms and 1000 ms with the JMA sleep times fixed at 100 ms.



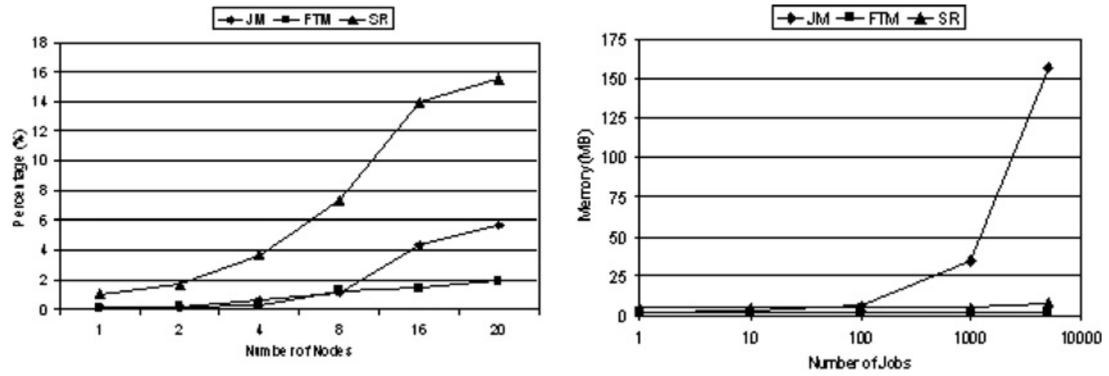
a) Testbed b) Cluster

Fig. 7 Job completion times for various agent sleep times.

For the testbed system, adjusting the JM sleep time provided at most a 3.4% performance improvement (i.e., agent sleep time of 50 ms versus 1000 ms 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 50 ms provides the best performance for the testbed. A sleep time of 25 ms was examined and was shown to provide the same performance as that of the 50 ms 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.

The results of the job completion time experiments on the ground-based cluster (Fig. 7b) show the same general trend as is found in the testbed results. Agent sleep times of 50 ms and 100 ms 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 500 ms 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 50 ms imposes an additional 20% processor utilization as compared to the system when set to 100 ms, though the overall processor utilization is relatively low for the DM middleware. The results suggest a JM sleep time of 100 ms is optimal because the overhead observed when the JM sleep time is above 100 ms imposes an overhead roughly equivalent to that observed when set to 100 ms. 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 Fig. 7b by two. The results demonstrate the scalability of the DM middleware in that the overhead imposed on the system by the DM middleware is linearly bounded.

Figure 8a presents the measured processor utilization in the cluster when the agent sleep times are set to 100 ms and Fig. 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 Fig. 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 Fig. 8b were the same on both platforms for all experiments.



a) Processor Utilization (Cluster) b) Memory Utilization (Both Platforms)

Fig. 8 System overhead imposed by the DM middleware.

C. Performance and Scalability Summary

The optimal system parameter values and middleware performance and scalability results observed in the preceding experiments are summarized in Table 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 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).

As previously described, the optimal value for the FTM sleep time was observed to be 50 ms 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 50 ms was found to be optimal for the testbed but 100 ms 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 100 ms 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 the number of nodes increases. As previously described, the optimal JMA sleep time was found to be 100 ms for both system types and is independent of system 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 100 ms.

Table 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	0.05 s	0.05 s	0.05 s	0.05 s
Optimal JM Sleep Time	0.05 s	0.1 s	0.1 s	0.1 s
Optimal JMA Sleep Time	0.1 s	0.1 s	0.1 s	0.1 s
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	6.1 MB	6.6 MB	6.9 MB	7.1 MB
Data Processor Memory Utilization	1.3 MB	1.3 MB	1.3 MB	1.3 MB
JMA Recovery Time (typical)	0.25 s	0.25 s	0.25 s	0.25 s
JMA Recovery Time (extreme)	0.25 s	2.70 s	9.44 s	14.95 s
Job Deployment Overhead	1.1 s	0.95 s	2.05 s	2.95 s

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 2) was measured to be 250 ms 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 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 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 provides a case-study analysis of the system using a simple but important computation kernel used in numerous imaging applications, the 2D Fast Fourier Transform (FFT).

D. Case Study Analysis

The previous sections contain results that illustrate the scalability of the proposed middleware and system design. However, the experiments have not directly addressed the level of scalability intended for future systems beyond 20 nodes. Also, general characteristics of the system such as processor utilization and memory usage were measured rather than studying the system performance and scalability in terms of key application benchmarks. In order to analyze the proposed system for a larger range of system sizes and under more realistic workloads, models of critical system components were created using MLDesigner, a discrete-event simulation toolkit from MLDesign Technologies, Inc. The study focused on the 2D-FFT algorithm since many space applications use it as a key computation kernel. The baseline simulated system was configured as shown in Table 3 to mimic the flight version of the experimental testbed system described in Section V.

A fault-tolerant, parallel 2D FFT was modeled and represented the baseline algorithm. The parallel FFT distributes an image evenly over N processing nodes and performs a logical transpose of the data via a corner turn. A single

Table 3 Simulation model parameters.

Parameter Name	Value
Processor Power	650 MIPS
Network Bandwidth	Non-blocking 1000 Mb/s
Network Latency	50 μs
MDS Bandwidth	3.0 MB/s
MDS Access Latency	50 ms
Image File Size	1 MB

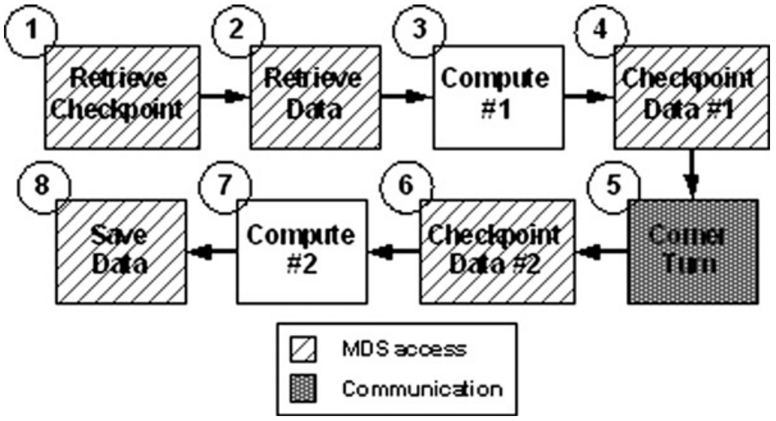


Fig. 9 Steps of parallel 2D FFT.

iteration of the FFT, illustrated in Fig. 9, includes several stages of computation, inter-processor communication (i.e., corner turn), and several MDS accesses (i.e., image read and write and checkpoint operations).

The results of the baseline simulation (see Fig. 10) show that the performance of the FFT slightly worsens as the number of data nodes increases. In order to pinpoint the cause of the performance decrease of the FFT, the processor, network, and MDS characteristics were greatly enhanced (i.e., up to 1000-fold). The results in Fig. 10 show that enhancing the processor and network has little effect on the performance of the FFT, while MDS improvements greatly decrease execution time and enhance scalability. The reason the FFT application performance is so directly tied to MDS performance is due to the high number of accesses to the MDS, the large MDS access latencies, and the serialization of accesses to the MDS.

After the MDS was verified as the bottleneck for the 2D FFT, several options were explored in order to mitigate the negative effects of the central memory. The options included algorithmic variations, enhancing the performance of the MDS, and combinations of these techniques. Table 4 lists the different variations.

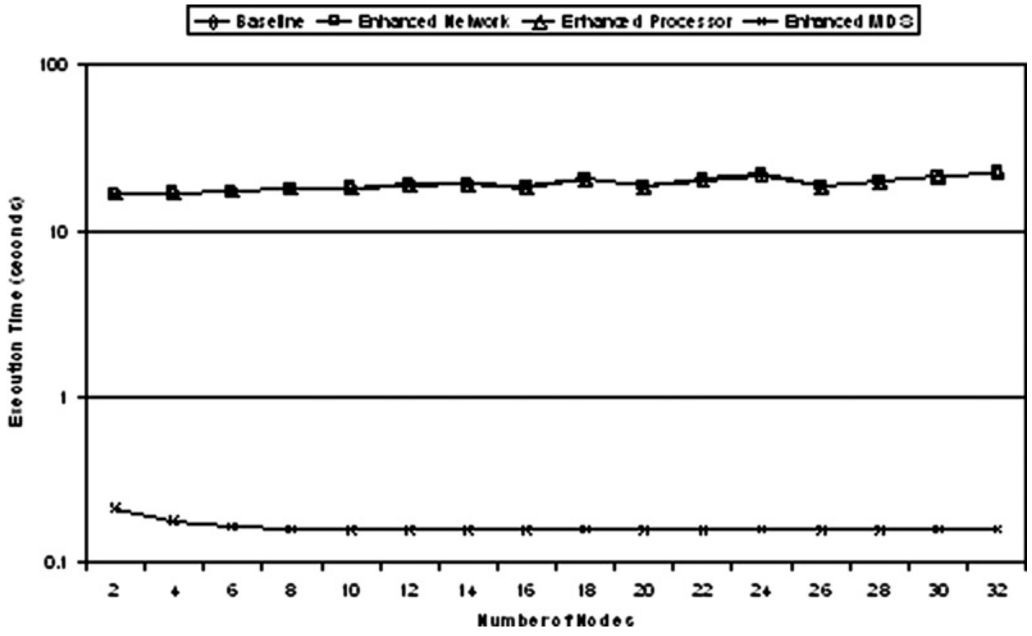


Fig. 10 Execution time per image for baseline and enhanced systems.

Table 4 FFT algorithmic variations and system enhancements.

Algorithm/Technique	Description	Label
Parallel FFT	Baseline parallel 2D FFT.	P-FFT (Baseline)
Parallel FFT with distributed checkpointing	Parallel 2D FFT with "nearest neighbor" checkpointing—data node $\boldsymbol{i}$ saves checkpoint data to data node $(\boldsymbol{i}+1)$ mod $\boldsymbol{N}$, where $\boldsymbol{i}$ is a unique integer $(0 \leq \boldsymbol{i} \leq \boldsymbol{N}-1)$ and $\boldsymbol{N}$ is the number of tasks in a specific job.	P-FFT-DCP
Parallel FFT with distributed data	Parallel 2D FFT with each node collecting a portion of an image for processing thus eliminating the data retrieval and data save stages.	P-FFT-DD
Parallel FFT with distributed checkpointing and distributed data	Combination of both distribution techniques described above.	P-FFT-DCP-DD
Parallel FFT with MDS enhancements	Parallel 2D FFT using a performance-enhanced MDS. The MDS bandwidth is improved 100-fold and the access latency is reduced by a factor of 50.	P-FFT-MDSe
Distributed FFT	A variation of the 2D FFT that has each node process an entire image rather than a part of the image.	D-FFT
Distributed FFT with distributed data	Distributed 2D FFT algorithm with each node collecting an entire image to process.	D-FFT-DD
Distributed FFT with MDS enhancements	Distributed 2D FFT algorithm using a performance-enhanced MDS.	D-FFT-MDSe

Each technique offers performance enhancements over the baseline algorithm (i.e., P-FFT). Figure 10 shows that the parallel FFT with distributed checkpointing and distributed data provides the best speedup (up to 740×) over the baseline because it eliminates all MDS accesses. Individually, the distributed checkpointing and distributed data techniques result in only a minimal performance increase since the time taken to access the MDS still dominates the total execution time. MDS performance enhancements reduce the execution of the parallel FFT by a factor of 5. Switching the FFT algorithm (see Fig. 12) to the distributed version achieves a 2.5× speedup over the baseline which can then be further increased to 14× and 100× by employing MDS improvements and distributed data, respectively. It is noteworthy to mention that the distributed FFT is well suited for larger systems sizes since the number of MDS accesses remains constant as system size increases.

The results for the parallel 2D FFT (see Fig. 11) magnify the affects of the MDS on the system's performance. Though the parallel FFT's general trend shows worse performance as system size scales, the top four lines show numerous anomalies where the performance of the FFT actually improves as the number of nodes in the system increases. These anomalies arise from the total number of MDS accesses needed to compute a single image for the entire system. For example, a dip in execution time occurs in the baseline parallel FFT algorithm when moving from 18 to 19 nodes. The total number of MDS accesses of the parallel FFT using 18 nodes is 90 while the number of accesses decreases to 76 for the 19-node case. Since the MDS is the system's bottleneck, the execution time of the algorithm benefits from the reduction of MDS accesses. Only in the parallel FFT with distributed data and distributed checkpointing option do we see the "zig-zags" disappear due to no data transfers occurring between the nodes and the MDS. The distributed FFT (see Fig. 12) also does not show any performance anomalies due to the nature of the algorithm. That is, the number of MDS accesses remains constant per image since only one node is responsible for computing that image.

The results in Figs. 11 and 12 corresponded to 1 MB images, thus we conducted simulations to analyze the affects of larger image sizes. Our results showed that the algorithms and enhancements reversed the trend for the parallel FFT. That is, the execution times improved as the system size grew, though the improvements were very minimal. Also, the sporadic performance jumps were amortized due to the large number of MDS accesses as compared to the variance in the number of accesses. The distributed FFT with distributed data was the only option that showed a large improvement because more processing could occur when data was more readily available for the processors.

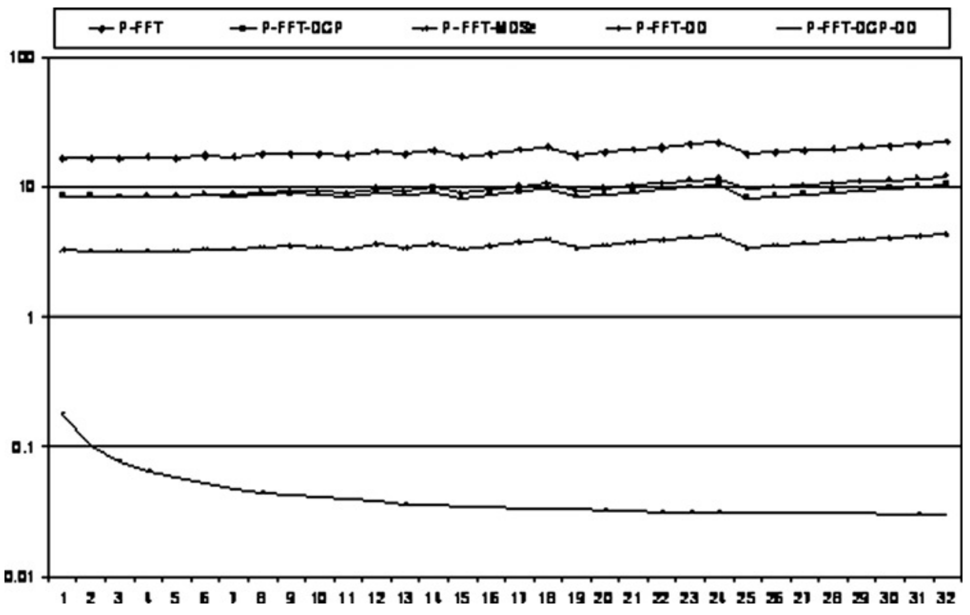


Fig. 11 Parallel 2D FFT execution times per image for various performance-enhancing techniques.

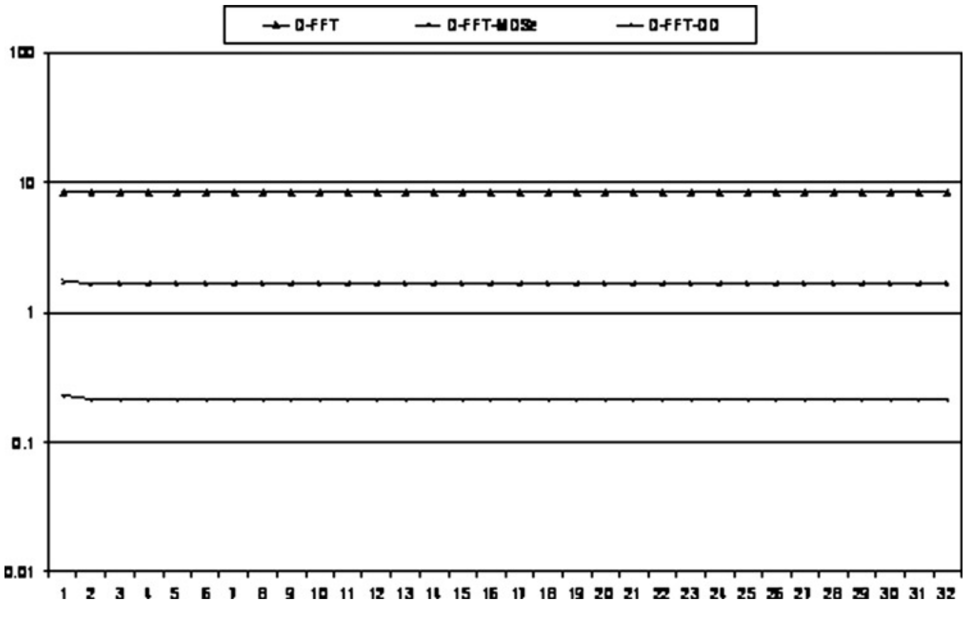


Fig. 12 Distributed 2D fft execution times per image for various performance-enhancing techniques.

The results demonstrate that a realistic application can be effectively executed by the DM system if the mass memory subsystem is improved to allow for parallel memory accesses and distributed checkpoints.

VII. 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 the 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 50 ms versus 1000 ms 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 50 ms provides the best performance for the testbed. For the ground-based cluster, agent sleep times of 50 ms and 100 ms 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 50 ms imposes an additional 20% processor utilization as compared to the system when set to 100 ms. A balance between response time and overhead imposed on the system must be struck and the results suggest an agent sleep time of 50 ms and 100 ms 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.

Performance and scalability studies beyond the 20-node ground-based cluster were performed using simulation to discover weaknesses in the current system architecture. The results showed that the centralized MDS can be a performance bottleneck when executing jobs that frequently access the MDS. Various techniques were explored to mitigate the MDS bottleneck including distributed checkpointing, distributing interconnections between sensors and data processors (i.e., distributed data), algorithm variations, and improving the performance of the MDS. The study showed that eliminating extraneous MDS accesses was the best option though enhancing the MDS memory was also a good option for increasing performance. With regards to scalability, changing the algorithm from a parallel to a distributed approach and including distributed checkpointing provides the best performance improvement of all the options analyzed. For large image sizes (i.e., 64 MB), the distributed FFT with distributed data was the only option that showed a large improvement because more processing could occur when data was more readily available for the processors. Based upon these results, a distributed mass-memory system will be further explored and developed in the future.

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 DM system and show the management service to be scalable and provides an adequate level of performance and fault tolerance with minimal overhead. Future work for the DM project includes additional fault-tolerance and availability analysis for failure rates likely to be observed in deep space missions.

Acknowledgement

The authors wish to thank Honeywell Inc., NASA NMP and NASA JPL for their continued support of this project, and a special thanks to Vikas Aggarwal at Tandel Inc. for developing benchmarks used in our experiments and to Rajagopal Subramaniyan, Grzegorz Cieslewski, Adam Jacobs, John Curreri and Mike Fischer of the HCS Lab at Florida for their support on the project.

References

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

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

[3]Dechant, D., "The Advanced Onboard Signal Processor (AOSP)," *Advances in VLSI and Computer Systems*, Vol. 2, No. 2, October 1990, pp. 69–78.

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

[5]Chen, F., Craymer, L., Deifik, J., Fogel, A., Katz, D., Silliman, A., Jr., Some, R., Upchurch S., and Whisnant, K., "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.

[6]Whisnant, K., Iyer, R., Kalbarczyk, Z., Jones III, P., Rennels D., and Some, R., "The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications," *IEEE Transactions on Software Engineering*, Vol. 30, No. 4, April 2004, pp. 257–277.

[7]Williams, J., Dawood A., and Visser, S., "Reconfigurable Onboard Processing and Real Time Remote Sensing," *IEICE Trans. on Information and Systems*, *Special Issue on Reconfigurable Computing*, Vol. E86-D, No. 5, May 2003, pp. 819–829.

[8]Williams, J., Bergmann, N., and Hodson, R., "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.

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

[10]Li, M., Goldberg, D., Tao W., and Tamir, Y., "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.

[11]Troxel, I., Jacob, A., George, A., Subramaniyan R., and Radlinski, M., "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.

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

[13]Fuller, S., "RapidIO - The Embedded System Interconnect," John Wiley & Sons, January 2005.

[14]Feitelson, D., and Rudolph. L., "Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control," *Journal of Parallel and Distributed Computing*, Vol. 35, No. 1, May 1996, pp. 18–34.

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

[16]Troxel I., and George, A., "Adaptable and Autonomic Mission Manager for Dependable Aerospace Computing," *Proc. IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, Indianapolis, IN, September 29-October 1, 2006 (to appear).

[17]Message Passing Interface Forum, "MPI: a message-passing interface standard," Technical Report CS-94-230, Computer Science Department, University of Tennessee, April 1, 1994.

[18]Subramaniyan, R., Aggarwal, V., Jacobs, A., and George, A., "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.

[19]Villarreal, J., Suresh, D., Stitt, G., Vahid, F., and Najjar, W., "Improving Software Performance with Configurable Logic," *Journal of Design Automation for Embedded Systems*, Vol. 7, No. 4, November 2002, pp. 325–339.

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

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

[22]Huang K., and Abraham, J., “Algorithm-Based Fault Tolerance for Matrix Operations”, *IEEE Transactions on Computers*, Vol. C-33, No. 6, June 1984, pp. 518–528.

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

Stanley Nissen
Associate Editor