IMPROVING UTILIZATION AND AVAILABILITY OF HIGH-PERFORMANCE COMPUTING IN SPACE

By

RAJAGOPAL SUBRAMANIYAN

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

UNIVERSITY OF FLORIDA

2006

To my dad Mr. Subramanian and my mom Mrs. Vijayalakshmi

ACKNOWLEDGMENTS

I thank Dr. Alan George for having confidence in me and for allowing me to pursue this research. I also thank him for his support during this work. I thank Scott Studham at Oak Ridge National Laboratory for laying the foundation of this work and for providing great support and guidance.

I express my gratitude to my parents, who have been with me during the ups and downs of my life. I thank them for their moral, emotional and financial support throughout my career. I would be nowhere without them.

I thank my friends Giridhar and Yamini for being friendly, encouraging and very supportive in helping me to graduate. I also thank my friends Maakans, Anitha, Arun, Anand, Thanni, PD, and Kandy for making life at UF memorable.

I thank my colleagues at the HCS Lab for their technical support and peer reviews. I especially thank Casey, Adam, Eric and Ian for being friendly and more than just lab mates, making my term at the lab a memorable one. Finally, I thank all the sponsors of this work. This work was supported by NSF, ORNL, and NASA/JPL.

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

IMPROVING UTILIZATION AND AVAILABILITY OF HIGH-PERFORMANCE COMPUTING IN SPACE

By

Rajagopal Subramaniyan

December 2006

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

Space missions involving science and defense ventures have ever-increasing demands for data returns from their resources in space. The traditional approach of data gathering, data compression and data transmission is no longer viable due to the vast amounts of data. Over the past few decades, there have been several research efforts to make high-performance computing (HPC) systems available in space. The idea has been to have enough "on-board" processing power to support the many space and earth exploration and experimentation satellites orbiting earth and/or exploring the solar system. Such efforts have led to small-scale supercomputers embedded in the spacecraft and, more recently, to the idea of using commercial-off-the-shelf (COTS) components to provide HPC in space. Susceptibility of COTS components to Single-Event Upsets (SEUs) is a concern especially since space systems need to be self-healing and robust to survive the hostile environment. Fault-tolerant system functions need to be developed to manage the resources available and improve the availability of the HPC system in space. However, resources available to provide fault tolerance are fewer than traditional HPC systems on earth.

Several techniques exist in traditional HPC to provide fault tolerance and improve overall computation rate, but adapting these techniques for HPC in space is a challenge due to the

resource constraints. In this dissertation, this challenge is addressed by providing solutions to improve and complement HPC in space. Three techniques are introduced and investigated in three different phases of this dissertation to improve the effective utilization and availability of HPC in space. In the first phase, new model to perform checkpointing at an optimal rate is developed to improve useful computation time. The results suggest the requirement of I/O capabilities much superior to present systems. While the performance of several common HPC scheduling heuristics that can be used for effective task scheduling to improve overall execution time is simulatively analyzed in the second phase, availability is improved by designing a new lightweight fault-tolerant message passing middleware in the third phase. Analyses of applications developed with the fault-tolerant middleware show that robustness of the systems in space can be significantly improved without degrading the performance. In summary, this dissertation provides novel methodologies to improve utilization and availability in space-based high-performance computing, thereby providing better and effective fault tolerance.

CHAPTER 1
INTRODUCTION

Space research has advanced leaps and bounds in the past few decades with numerous satellites, probes and space shuttles launched to explore near-earth space and other astronomical bodies including earth itself. More and more of our universe is being explored and, with permanent space stations already in orbit around earth, space exploration is continuing to grow. There has been a significant increase in the capability of instruments deployed in space. With such high-tech instruments in place, space missions involving science and defense ventures have ever-increasing demands for data returns from their corresponding resources in space. The traditional implementation approach of data gathering, data compression, and data transmission is no longer viable. The amount of data being generated is becoming too large to be transmitted via available downlink channels in reasonable time. An industry-proposed solution to reduce the demand on the downlink is to move processing onto the spacecraft [1]. The idea has been to have enough on-board processing power to support the many space and earth exploration and experimentation satellites orbiting earth and/or exploring the solar system. However, this approach is mired by the limited capabilities of today's on-board processors and the prohibitive cost of developing radiation-hardened high-performance electronics.

Microelectronics designed for environments with high levels of ionizing radiation such as space have special design challenges. A single charged particle of radiation can knock thousands of electrons loose, causing electronic noise, signal spikes, and in the case of digital circuits, plainly incorrect results. The problem is particularly serious in the design of artificial satellites, spacecraft, military aircraft, nuclear power stations, and nuclear weapons. In order to ensure the proper operation of such systems, manufacturers of integrated circuits and sensors intended for the aerospace markets employ various methods of radiation hardening. The resulting systems are

said to be rad(iation)-hardened or rad-hard. Most rad-hard chips are based on their more mundane commercial equivalents, with manufacturing and design variations that reduce the susceptibility to radiation and electrical and magnetic interference. Typically, the hardened variants lag behind the cutting-edge commercial products by several technology generations due to the extensive development and testing required to produce a radiation-tolerant design [2].

As mentioned earlier, the approach of moving processing onto the spacecraft is hindered by the radiation effects influencing technological capabilities and cost of the high-performance electronics. This situation has encouraged researchers and industry to consider the use of commercial-off-the-shelf (COTS) components for on-board processing. Furthermore, the recent adoption of silicon-on-insulator (SOI) technology by COTS integrated foundries is resulting in devices with moderate space radiation tolerance [1]. Such an approach would make high-performance computing (HPC) possible in space. However, in spite of this progress, COTS components continue to be highly susceptible to Single-Event Upsets (SEU) that are consequences of radiation. SEU as defined by National Aeronautics and Space Administration (NASA) is "radiation-induced errors in microelectronic circuits caused when charged particles (usually from the radiation belts or from cosmic rays) lose energy by ionizing the medium through which they pass, leaving behind a wake of electron-hole pairs" [3]. High rate of SEUs lead to many component failures making HPC in space a significant challenge and certainly more complex than traditional HPC on ground. Technology must be developed that capitalize on the strengths of COTS devices to realize HPC in space while overcoming their susceptibility to SEUs without negating their benefits.

In general, computations are lost when there are failures and the full potential of the system is lost. Failures decrease the availability of the system. Availability gives a measure of the

readiness of usage of a system (e.g., a highly available system has a low probability of being down at any instant of time). Improving availability requires the system to be built with fault-tolerant features. But, these features might take additional resources reducing the effective utilization of the system. Effective utilization gives a measure of the time the system is used for actual computation. The impact of failures is worse in space environments compared to traditional HPC environments as the computational resources are limited. The chances of failures are high in harsh environments and the liberty in terms of resources available to provide fault tolerance is much less. HPC in space is not the same as traditional HPC on ground due to several constraints including harsh environments (e.g., high radiation, high rate of SEUs, high rate of component failures), limited resources attributed primarily to weight and volume constraints (e.g., processing speed, storage capacity, power availability), cost and timing constraints. Additionally, space computing also requires automated processing and repair to recover from faults if and when they occur in the system.

The traditional approach to SEU or transient error mitigation in soft, radiation-tolerant hardware is redundant self-checking and comparison, either in hardware or in software. In some applications, size, weight, and power constraints of the mission may preclude the use of redundant hardware, and the time constraints of the mission may preclude the use of redundant computation in software. In such cases, an alternative approach must be found which can provide adequate SEU protection with a single-string or single-execution implementation. For example, Samson et al. compare the overhead of traditional redundant self-checking for SEU mitigation with an application-specific fault tolerance method [4]. It is reported that self-checking configurations consume twice the power, twice the weight, twice the size, twice the cost, and roughly one half the reliability of the single-board solution.

Several techniques exist for traditional HPC to provide fault tolerance and improve overall computation rate. Some of the solutions include periodic backup of data (e.g., checkpointing), exposing computation to lesser faults by reducing overall execution time of applications (e.g., effective task scheduling) and reducing the impact of failure (e.g., fault-tolerant middleware). However, adapting these solutions for HPC in space is a challenge due to the resource constraints discussed earlier. In this dissertation, we address this challenge by investigating, developing and evaluating techniques to improve and complement HPC in space.

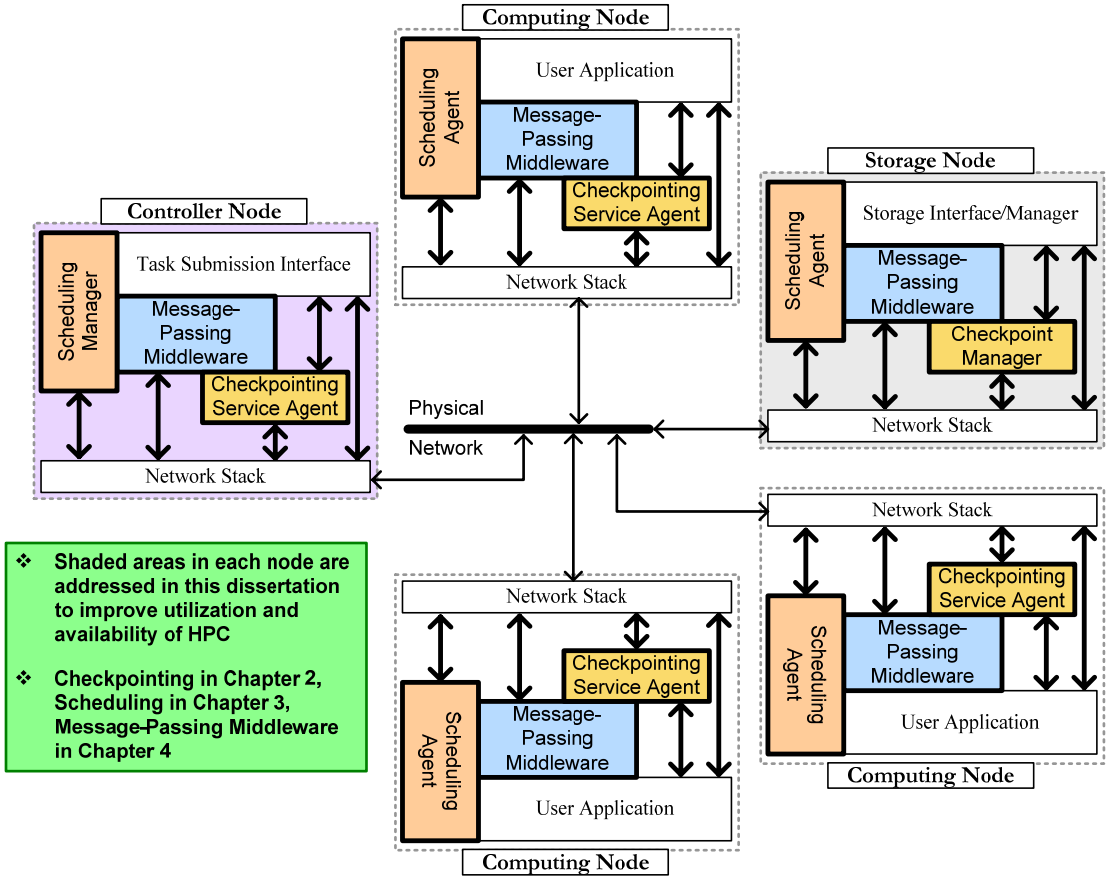


Figure 1-1. Typical high-performance computing system.

Figure 1-1 shows the various software agents involved in a typical HPC system either on ground or in space. The system has three types of nodes: a Controller Node, several Computing Nodes, and one or more Storage Nodes. The Controller Node is in charge of accepting tasks for

execution and scheduling the tasks on idle resources (i.e., Computing Nodes) and is generally radiation hardened in space systems. The Computing Nodes perform the actual execution of the application while the Storage Node is responsible for data storage and backup. The data could be input, output or system states (required for restarts on failures). The application processes executing on the Computing Nodes communicate via the Message-passing Middleware. In this dissertation, we address fault tolerance by focusing on the shaded areas in Figure 1-1, namely checkpointing, scheduling and message passing middleware.

Three techniques are discussed in three different phases of this research with the overall goal of improving the effective utilization and availability of HPC in space. In Phase 1, the useful computation time of the system is improved (increased) by optimizing checkpointing-related defensive I/O. We model the optimum checkpointing frequency for applications in terms of the mean-time-between-failures (MTBF) of the system, amount of memory checkpointed, and sustainable I/O bandwidth of the system. Optimal checkpointing maximizes useful computation time without jeopardizing the applications due to failures while minimizing the usage of resources. In Phase 2, the overall execution time of the application is improved (reduced) by simulatively analyzing scheduling heuristics for application task scheduling. We analyze techniques to effectively schedule tasks on parallel hardware reconfigurable resources that would be part of the HPC space system. Effective task scheduling reduces the overall execution time thereby exposing the application to lesser faults while reducing the resource usage as well. In Phase 3, availability of the system is improved by designing a new lightweight fault-tolerant message-passing middleware. The fault-tolerant middleware reduces the impact of failures on the system. The recovery time of the system is improved allowing unimpeded execution of

applications as much as possible. The techniques that we propose in this research are also applicable to traditional HPC on the ground but are critical for HPC in space.

This dissertation contains a discussion of the modeling of checkpointing process and optimization of defensive I/O. The dissertation also describes the simulative analysis of dynamic scheduling heuristics for effective task scheduling followed by the design and evaluation of a fault-tolerant message passing middleware. Conclusions and directions for future research are provided finally.

CHAPTER 2
OPTIMIZATION OF CHECKPOINTING-RELATED I/O (PHASE I)

In general, computation of large-scale scientific applications can be divided into three phases: start-up, computation, and close-down with I/O existing in all phases. A typical I/O pattern has start-up phase dominated by reads, close-down by writes and computation phase by both reads and writes. The primary questions of importance with relevance to I/O involved in the three phases are when, how much and how often? These questions can be addressed by segregating I/O into two portions: productive I/O and defensive I/O [5]. Productive I/O is the writing of data that the user needs for actual science such as visualization dumps, traces of key scientific variables over time, etc. Defensive I/O is employed to manage a large application executed over a period of time much larger than the platform's MTBF. Defensive I/O is only used for restarting a job in the event of application failure in order to retain the state of the computation and hence the progress since the last application failure. Thus, one would like to minimize the amount of resources devoted to defensive I/O and computation lost due to platform failure. As the time spent on defensive I/O (backup mechanisms for fault tolerance) is reduced, the time spent on useful computations will increase. This philosophy applies to high-performance distributed computing in the majority of environments ranging from supercomputing platforms to small embedded cluster systems, although the impact varies depending on the system and other resource constraints.

The impact of productive I/O on I/O bandwidth requirements can be reduced by better storage techniques and to some extent through improved programming techniques. It has been observed that defensive I/O dominates productive I/O in large applications with about 75% of the overall I/O being used for checkpointing, storing restart files, and other such similar techniques for failure recovery [5]. Hence, by optimizing the rate of defensive I/O, we can reduce the

overall I/O bandwidth requirement. Another advantage is that the optimizations used to control defensive I/O would be more generic and not specific to applications and platforms. However, reducing defensive I/O is a significant challenge.

Checkpointing of a system's memory, to mitigate the impact of failures is a primary driver of the sustainable bandwidth to high-performance filesystems. Checkpointing refers to the process of saving program/application state, usually to a stable storage (i.e., taking the snapshot of a running application for later use). Checkpointing forms the crux of rollback recovery and hence fault-tolerance, debugging, data replication and process migration for high-performance applications. The amount of time an application will tolerate suspending calculations to perform a checkpoint is directly related to the failure rate of the system. Hence, the rate of checkpointing (how often) is primarily driven by failure rate of the system. If the checkpointing rate is low, less resource are consumed but the chance of high computational loss (both time and data) is increased. If the checkpointing rate is high, resource consumption is greater but the chance of computational loss is reduced. It is important to strike a balance and an optimum rate of checkpointing is required. Finding a balance is a difficult problem even in traditional ground-based HPC with fewer failures and more resources. The problem is aggravated for HPC with embedded cluster systems in harsh environments such as space with more failures and less resources.

In this phase of the dissertation, we analytically model the process of checkpointing in terms of MTBF of the system, amount of memory checkpointed, sustainable I/O bandwidth and frequency of checkpointing. We identify the optimum frequency of checkpointing to be used on systems with given specifications, thereby making way for efficient use of available resources and gain maximum performance of the system without compromising on the fault tolerance

aspects. The useful computation time is increased thereby improving the effective system utilization. Further, we develop discrete-event models simulating the checkpointing process to verify the analytical model for optimum checkpointing. The simulation models are developed using the Mission-Level Designer (MLD) simulation tool from MLDesign Technologies Inc. [6]. In the simulation models, we use performance numbers that represent systems ranging from small cluster systems to large supercomputers.

The remainder of this chapter is organized as follows. Section 2.1 provides background on the growth trends of technologies to study the effectiveness of this research to improve I/O usage. Section 2.2 briefly highlights why checkpointing is the most common methodology of providing fault tolerance. In Section 2.3, the checkpointing process is analytically modeled to identify the optimum frequency of checkpointing to be used on systems with given specifications. Section 2.4 describes the simulation models that we develop to verify our analytical models, while the results derived from the analytical model are provided in Section 2.5. Section 2.6 provides conclusions for this chapter and directions for future research.

2.1 Background on Technology Growth Trends

In this section, we study the growth trend of technologies involved in HPC, highlighting the poor growth of sustainable I/O bandwidth. The poor growth of I/O bandwidth compared to other technologies substantiates our approach to reduce resource consumption and improve useful computation time by optimizing checkpointing-related defensive I/O. It is important to mention that although we have used performance numbers from traditional ground-based HPC and supercomputing platforms (due to lack of any standard representative platforms for HPC in space) to study the growth trends, the numbers and trends are representative of what would be coming for HPC in space. The performance of all the technologies however is relatively poorer for HPC in space due to radiation hardening.

Gordon Moore observed an exponential growth in the number of transistors per integrated circuit and predicted this trend would continue [7]. He made this famous observation in 1965, just four years after the first planar integrated circuit was discovered. This doubling of transistors every eighteen months, referred to as "Moore's Law", has been maintained and still holds true. Similar to other exponentially growing systems, Moore's law can be expressed as a mathematical equation with a derived compound annual growth rate. Let x be a quantity growing exponentially, in this case the number of transistors per integrated circuit, with respect to time t as measured in years. Then, $x(t) = x_0 e^{kt}$, where k is the compound annual growth rate, and the rate of change follows $\frac{dx}{dt} = kx$.

For Moore's law, the compound annual growth rate can be established as k_{moore}=0.46. When t=1.5, and $\frac{x}{x_0} = 2$; $k = \frac{\ln(2)}{1.5} = 0.46$. However, this compound annual growth rate is not the same for the other technologies involved in HPC. We briefly overview the growth of several technologies including CPU processing power, disk capacity, disk performance, and sustained bandwidth to/from disks in the remainder of this section.

2.1.1 CPU Processing Power

Modern HPC systems are made by tightly coupling multiple integrated circuits. In addition to being able to capitalize on the exponential growth observed in Moore's Law, these systems have also been able to increase the average number of processors in systems to have a peak performance that exceeds Moore's law. Figure 2-1 shows the LINPACK [8, 9] rating for the tenth most powerful computer in the world for several years as ranked by the Top500 [10] list. The compound annual growth rate can be established as k_{HPC}=0.58 (values of k for the technologies are calculated as shown for Moore's Law). We picked the tenth computer for no

special reason except that in our opinion, the computers at the very top might have been custom tuned and hence may not provide the general trend. The benchmark used to establish the Top500 list highlights the combined performance of the processors used to build the supercomputer. It is important to realize that other technologies such as disk performance, memory performance, and networking performance are not fully represented in the Top500 benchmark [10]. Growing the relative performance of these other technologies is equally important to the CPU processing power when looking to establish a well balanced system.

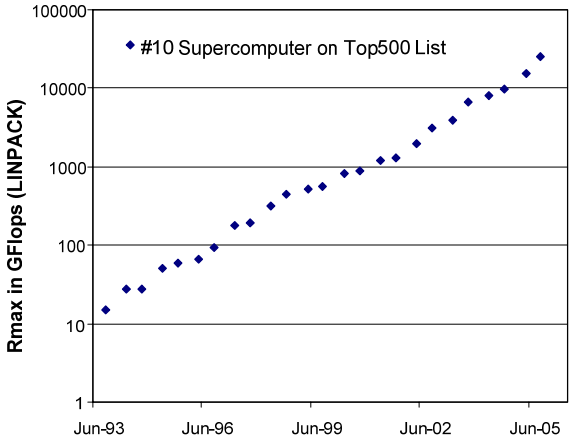


Figure 2-1. Supercomputer performance over time

2.1.2 Disk Capacity

Figure 2-2 shows the capacity of a 95mm 7,200RPM disk drive over time. Areal density of hard drives has grown at an impressive compound annual growth rate of k_{IO_cap}=0.62, and has accelerated to greater than a k_{IO_cap}=0.75 rate since 1999 as shown in the figure. We are now seeing the delivery of 120 GB/inch^2 for magnetic disk technology, with demonstrations over 240 GB/inch^2 routinely occurring in laboratories [11]. The first 1 TB (1,000GB) hard disk drives are expected to ship in 2007[11]. Heinze et.al [12] demonstrate the theoretical limit of physical media at approximately 256 TB/inch^2 using a single atomic layer to form a two-dimensional

antiferromagnet. In the next five to ten years, the likelihood of perpendicular recording using a patterned media may likely appear to further increase recording densities [11]. In addition to continued evolutionary advancement, there are new disruptive storage technologies nearly ready to enter the market place like Magnetic Random Access Memory (MRAM) [13] and Micro-Electro-Mechanical-Systems (MEMS) [14]. Thus we see a good growth in hard drive capacities.

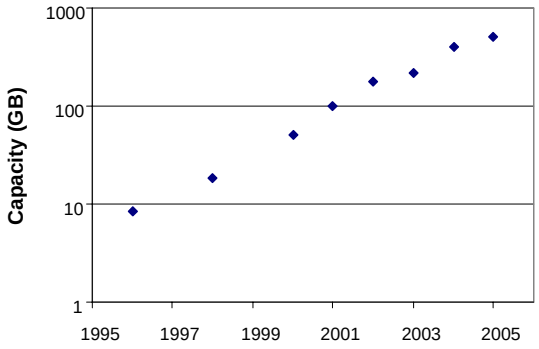


Figure 2-2. Hard drive capacity over time

2.1.3 Disk Performance Growth

Disk performance has not kept pace with the growth in disk capacities. More densely packed data means fewer disk actuators for a given amount of storage. While the compound annual growth rate of the areal density of magnetic disk recording has increased at an average of over 60 percent annually, the maximum number of random I/Os per second that a drive can deliver is improving at an annual compounded growth rate of less than $k_{IO\_perf\_io}$=0.20. Continual increases in capacity without corresponding performance improvements at the drive level create a performance imbalance that is defined by the ratio called access density. Access density is the ratio of performance, measured in I/Os per second, to the capacity of the drive, usually measured in gigabytes (access density = I/Os per second per gigabyte).

As seen in Table 2-1, access density has steadily declined while the capacity has increased substantially. Access density is becoming a significant factor in managing storage subsystem

performance and the tradeoffs of using higher-capacity disks must be carefully evaluated as lowering the cost-per-megabyte most often means lowering the performance. Future I/O-intensive applications will require higher access densities than are indicated by the current development roadmaps. Higher access densities may be achieved through lowering the capacity per actuator or dramatically increasing the I/Os-per-second capabilities of the drive. The latter is much harder to accomplish, particularly for random access applications where a seek (disk arm movement) is required.

Vendors are making small high-performance drives such as a 15,000 RPM 73.4 GB drive from Seagate with an advertised seek time of 3.6 ms leading to an access density of about 3.8 IO/s per GB. As of April 2006 these high performance drives cost $3.3 per GB [15], a factor of 11 higher than the $0.36 per GB for low cost commodity drives [16]. The factor of 9 increase in performance is offset by the factor of 9 increase in cost, leaving most architects to select commodity drives for the additional capacity.

Table 2-1. Disk performance growth over time

Year	Drive	Seek Time (ms)	I/Os per Sec	Capacity (GB)	I/Os per Sec per GB
1964	2314	112.5	8.9	0.029	306.50
1975	3350	36.7	27.2	0.317	86.00
1987	3380K	24.6	40.7	1.890	21.50
1996	3390-3	23.2	43.1	8.520	5.10
1998	Cheetah-18	18.0	55.6	18.200	3.10
2001	WD1000	8.9	112.4	100.000	1.10
2002	180GXP	8.5	117.6	180.000	0.70
2004	Deskstar 7K400	8.5	117.6	400.000	0.30
2005	Deskstar 7K500	8.5	117.6	500.000	0.24

2.1.4 Sustained Bandwidth to/from Disk

Sustained bandwidth from a disk relative to capacity of the disk has also continued to decline. The sustained bandwidth of a disk is dependent upon the physical location of the data on the disk. Due to a fixed rotational speed, the closer to the center of the disk platter the slower

the sustained read rate. Figure 2-3 shows the sustained transfer rate in MB/s for the 15,000 RPM, 37 GB disk drive from Seagate [17]. The first 5 gigabytes of data are transferred at a rate of 76 MB/s; meanwhile the final 3 GB of data only sustained 49 MB/s. Although vendors provide the peak performance number in general, the average and minimum sustained performance can be significantly lower. It is important to properly layout the data on the hard drive to achieve consistent performance.

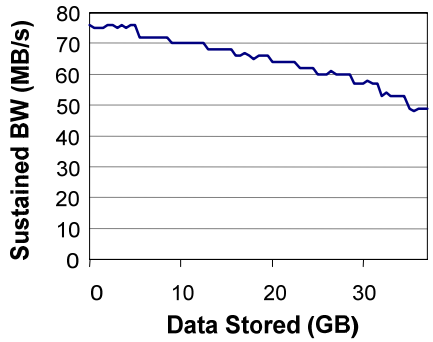


Figure 2-3. Performance of a single disk with respect to the amount of data stored

Figure 2-4 highlights the minimum, average and maximum performance for typical disk drives introduced between 1995 and 2004. The average sustainable bandwidth from a hard disk drive has grown at an annual compounded growth rate of $k_{IO\_perf\_BW}$=0.26 per year.

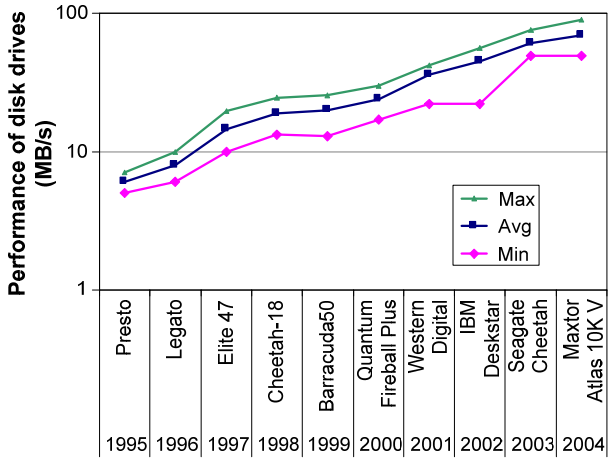


Figure 2-4. Growth of disk drive bandwidth over time

2.1.5 I/O Performance

The technologies pertinent to HPC discussed in this section thus far are all growing exponentially although some are growing substantially slower than others. Table 2-2 summarizes the rate of change for these technologies.

Table 2-2. Growth rate of computing technologies

Technology	Growth rate	
Transistors per integrated circuit	k_{moore}	0.46
LINPACK on Top10 supercomputer	k_{HPC}	0.58
Capacity of hard drives	k_{IO_cap}	0.62
Cost per GB of storage	k_{IO_cost}	-0.83
Performance of hard drive in I/Os per second	$k_{IO\_perf\_io}$	0.20
Performance of hard drive in bandwidth	$k_{IO\_perf\_BW}$	0.26

2.2 Optimal Usage of I/O

The performance of disk drives measured in both I/Os per second and sustained bandwidth is not keeping up with other technology trends. Assuming current systems meet I/O performance needs and for the balance of bandwidth per computational power to be maintained, we can use the formula $d(t) = e^{t(k_{HPC}-k_{IO\_perf})}$ to calculate number of more disk drives that will be needed in order to maintain that balance.

In order to show the importance of efficient usage of I/O resources, in Figure 2-5 we show how many disks will be required to work in perfect parallel to maintain system balance in the coming years. Given these growth trends we will need to use 46 times more disks in ten years in order to maintain the same balance of I/Os per second on a classical supercomputer. For example in 2004, a typical 10 TeraFLOP supercomputer may be capable of 500,000 theoretical I/Os per second, or 50 I/Os per GigaFLOP. Systems of this size normally have disk drives in the order of around 5,000. Given historical growth rates a comparable supercomputer would have 3.4 PetaFLOPs in 2014 and need to be capable of 170,000,000 theoretical I/Os per second to

keep the same balance. However, as disk performance is not growing at the same exponential rate as the computer performance, it would be necessary to have 46 times more disk drives, necessitating disk drives in the order of around 230,000. The gap might further widen in the years to follow necessitating a fundamental change to the way we approach storage.

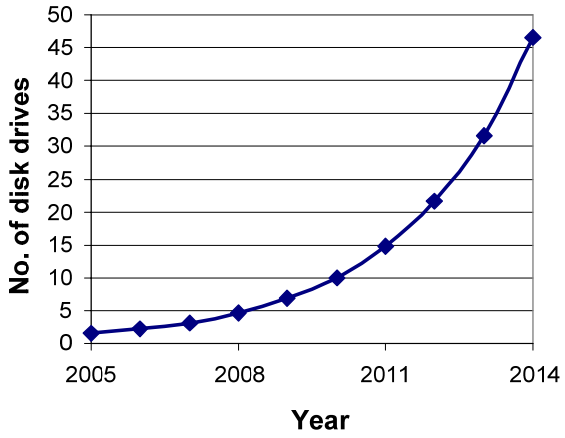


Figure 2-5. Number of disks required in future systems to maintain present performance levels

Based on the growth trend of the different technologies, it can be seen that I/O performance has fallen behind other technologies and the criticality of efficient usage of I/O resources can be realized. In the harsh space environment that is prone with failures and where the technologies are slower than their counterparts on ground, optimal usage of I/O resources is even more critical. As mentioned earlier, checkpointing is a critical process that drives the need to improve the I/O bandwidth with frequent and at times lengthy accesses to the disk. Hence, optimizing the rate of checkpointing will optimize the usage of I/O resources. An alternative would be improving the checkpointing process itself with new techniques to consume fewer resources.

2.2.1 Checkpointing Alternatives

Several strategies are used as alternatives to traditional disk-based checkpointing. Many researchers are working on diskless checkpointing (checkpoints are encoded and stored in a

distributed system instead of a central stable storage), for example [18], and suggest this strategy as an alternative to conventional disk-based checkpointing to reduce the I/O traffic due to checkpointing. The National Nuclear Security Administration (NNSA) issued a news release about the first full-system three-dimensional simulations of a nuclear weapon explosion (Crestone project), a significant achievement for Los Alamos National Laboratory and Science Applications International Corporation [19]. The simulation is essentially a 24-hour, seven-day-a-week job for more than seven months. For a highly important seven-month computation such as Crestone, the notion of checkpointing and a rolling computation go hand-in-hand. The small success stories of using diskless checkpointing fail in such cases of large applications. True disk-based incarnations are required over heavy I/O phases of the code. Moreover, diskless checkpointing is often application-dependent and not generic.

Other checkpointing alternatives include incremental checkpointing (i.e., checkpointing only the information that has changed since previous checkpoint) and distributed checkpointing (i.e., individual nodes in the system checkpoint asynchronously thereby reducing the simultaneous load on the network). Although these emergent schemes may have their advantages, more maturity is required for their use in large-scale, mission-critical applications. In the current scenario, checkpoints are mostly synchronous with all the nodes writing checkpoints at the same time. Additionally, the checkpoints ideally involve the storage of the full system memory (at least 70% to 80% of the full memory in general) [19]. This scenario is quite common when schedulers such as Condor [20] are used or when applications checkpoint using libraries such as Libckpt [21]. The frequency of checkpointing can be decreased for productive I/O to surpass defensive I/O, but not without the risk of losing more computation due to system failures.

2.2.2 Optimizing Checkpointing Frequency

In this phase of the research, we model the overhead in a system due to checkpointing with respect to the MTBF, memory capacity and I/O bandwidth of the system. In so doing, we identify the optimum frequency of checkpointing to be used on systems with given specifications. Optimal checkpointing helps to make efficient use of the available I/O and gain the maximum performance out of the system without compromising on its fault tolerance aspects. There have been similar efforts earlier to model and identify optimum checkpointing frequency for distributed systems [22-25]. However, these efforts have not been simulatively or experimentally verified, and the approaches as yet are too theoretical to be practically implemented on systems.

It should be mentioned that optimizing the frequency of checkpointing is just one method of reducing the impact of defensive I/O on I/O bandwidth requirements. Other methods might include improvement of the storage system (high-performance storage system, high-performance interconnects, etc.), novel methods and algorithms for checkpointing, etc. There are claims that we can hide the impact of defensive I/O and work around this problem rather than tackling it. However, there are no recorded proofs to substantiate such claims.

2.3 Analytical Modeling of Optimum Checkpointing Frequency

Distribution of the total running time t of a job is determined by several parameters including:

- Failure rate λ; $\lambda = \frac{1}{T_{MTBF}}$, where T_{MTBF} is the MTBF of the system
- Execution time without failure or checkpointing, T_x
- Execution time between checkpoints (checkpoint interval), T_i
- Time to perform a checkpoint (checkpointing time), T_c

- Operating system startup time, T_o
- Time to perform a restart (recovery time), T_r

Figures 2-6 and 2-7 show the various time parameters involved in the execution of a job without and with checkpointing respectively. Without checkpointing, the system has to be failure free for a duration of T_x for the computation to complete successfully. When a failure occurs, computation is restarted from its initial state after system recovery. With checkpointing, the system state is checkpointed periodically. When the system is recovered from a failure, computation is resumed from the latest stable checkpointed state on system recovery.

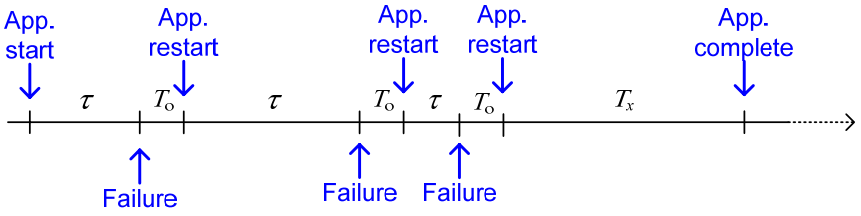


Figure 2-6. Job execution without checkpointing

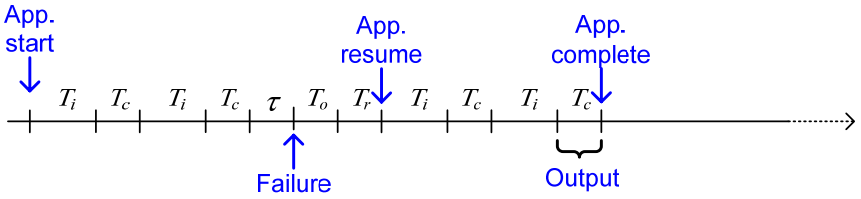


Figure 2-7. Job execution with checkpointing

The execution process with checkpointing and failures can be modeled as a Markov process as shown in Figure 2-8 where nodes 0,1,2,…,n represent stable checkpointed states and 0',1',2',….,n represent failed states. Let t_1, t_2, t_3,…,t_n be the random variables for the time spent in each cycle between two checkpointed states. These random variables are represented by τ in general.

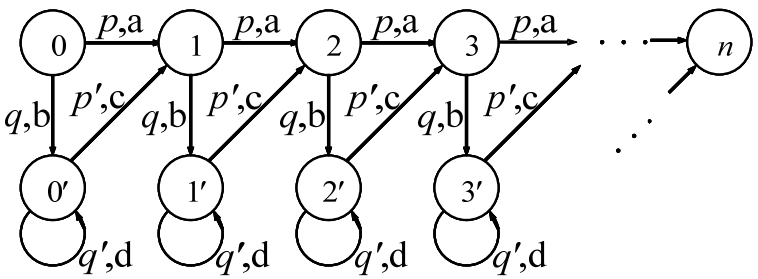


Figure 2-8. Execution modeled as a Markov process

The delays associated with each event in Figure 2-8 are as follows:

a: T b: $\tau \mid \tau \le T$ c: $T+T'$ d: $\tau \mid \tau \le T+T'$ here $T = T_i + T_c$ and $T' = T_o + T_r$

The probabilities associated with each event in Figure 2-8 are as follows:

$$p = e^{-\lambda T} \quad p' = e^{-\lambda(T+T')} \quad q = \text{prob}(\tau \le T) = 1-p \qquad q' = \text{prob}(\tau \le T+T') = 1-p'$$

It should be noted that the total running time is a sum of individual random variables representing individual checkpointing cycles. However, the random variables are similar and hence the mean total running time ($\bar{t}$) can be given as the sum of the mean running time of each cycle.

$$\bar{t} = E(t) = E(t_1) + E(t_2) + E(t_3) + ... + E(t_n)$$

$$\bar{t} = n \times E(t_1) \tag{2.1}$$

The mean running time of each cycle can be found by multiplying the probabilities associated with each path in the Markov chain and the corresponding time delay. There are several paths in the Markov chain that the process can actually take. The process can move from state 0 to state 1 with probability *p* and the delay associated with that transition is *a*. When there are failures in the system, the process does not directly move from state 0 to state 1. Instead, the process moves to state 0' with probability *q* and loops back in the same state with probability *q*'. The delays associated with these state transitions represented by *b* and *d* respectively. *b* and *d* are exponential random variables with upper limit of *T* and *T*+*T*' respectively. With probability

p', the system moves from the failed state to the stable state 1 and delay associated with the transition represented by c is equal to $T+T'$, the upper limit of d.

$$E(t_1) = p \times a + q \times p' \times (b + c) + q \times q' \times p' \times (b + c + d) + q \times q'^2 \times p' \times (b + c + 2d) + \ldots$$

$$= p \times a + q\left[(b + c) + \frac{q' \times d}{1 - q'}\right] \tag{2.2}$$

where

$$a = T; \quad b = e^{-\lambda T}\left(-T - \frac{1}{\lambda}\right) + \frac{1}{\lambda}; \quad c = T + T'; \quad d = e^{-\lambda(T+T')}\left(-(T + T') - \frac{1}{\lambda}\right) + \frac{1}{\lambda};$$

Substituting the corresponding time delays into Eq. 2.2 we get Eq. 2.3

$$E(t_1) = Te^{-\lambda T} + \left(1 - e^{-\lambda T}\right)\left[-e^{-\lambda T}\left(T + \frac{1}{\lambda}\right) + e^{-\lambda(T+T')}(T + T') + \frac{1}{\lambda}\left(\frac{\left(1 - e^{-\lambda(T+T')}\right)^2 + e^{-\lambda(T+T')}}{e^{-\lambda(T+T')}}\right)\right] \tag{2.3}$$

As can be seen from Eqs. 2.1 and 2.3, the expression for the mean total running time is complicated. To avoid further complexity, we followed a different method as follows. The Laplace transform of a function $f(x)$ is given by $\phi_x(s) = \int f(x)e^{-sx}dx$. We can find the Laplace transform of the pdf of the total running time. The negative of the derivative of the Laplace transform at s=0 gives the mean total running time.

$$\bar{\phi}_t(s) = \frac{d}{ds}\left(\int f(t)e^{-st}dt\right) = \int (-t)f(t)e^{-st}dt \tag{2.4}$$

$$\bar{\phi}_t(0) = \int (-t)f(t)dt = -E[t] \Rightarrow \bar{t} = -\bar{\phi}_t(0) \tag{2.5}$$

Laplace transform can be calculated by finding the Laplace transforms of individual transitions in the Markov process and then combining them together. For example, the Laplace transform on the pdf of the time required for the transition from state 0 to state 1 without a failure is given by e^{-sT} and the transition happens with probability p. The transform on the pdf

of the time required for transition from state 0 to state 0' that happens with probability q is given by $\varphi(s,T) = \frac{1}{1-e^{-\lambda T}} \int_0^T \lambda e^{-\lambda t} e^{-st} dt = \frac{\lambda}{\lambda + s}\left(\frac{1-e^{-(\lambda+s)T}}{1-e^{-\lambda T}}\right)$.

The Laplace transform on the pdf of the time required for the transition from state 0' to state 1 that happens with probability p' is given by $e^{-s(T+T')}$. If there is looping in state 0' due to repeated failures, the transform on the pdf of the time spent returning to state 0' (transition probability q') is given by $\varphi(s,T+T') = \frac{1}{1-e^{-\lambda(T+T')}} \int_0^{T+T'} \lambda e^{-\lambda t} e^{-st} dt = \frac{\lambda}{\lambda + s}\left(\frac{1-e^{-(\lambda+s)(T+T')}}{1-e^{-\lambda(T+T')}}\right)$.

The Laplace transform for the process with failures (state transition from 0 to 1 via 0') is found by doing a weighted sum of the Laplace transforms on the pdfs of the individual random variables. The transform is given by $q \times \varphi(s,T)\left[\sum_{i=0}^{\infty} p' e^{-s(T+T')} [q' \times \varphi(s,T)]^i\right]$ where i denotes the number of number of loops in state 0'. Hence, the Laplace transform of the pdf for one cycle of the Markov process is be given by $\phi_{t_1}(s) = pe^{-sT} + q \times \varphi(s,T)\left[\sum_{i=0}^{\infty} p' e^{-s(T+T')} [q' \times \varphi(s,T)]^i\right]$.

We get Eq. 2.6 from the above and differentiating Eq. 2.6 w.r.t s and substituting s=0, we get mean total running time given by Eq. 2.7. We verified the validity of the expressions for the mean total running time given by Eqs. 2.4 and 2.7 by running a Monte Carlo simulation with 10000 iterations and cross checking the results. We found that the time given by the expressions in the equations closely matched the simulation results. We will be using the expression in Eq. 2.7 for further development for simplicity reasons both in terms of representation and computation.

$$\phi_t(s) = \phi_{t_1}(s) \times \phi_{t_2}(s) \times \phi_{t_3}(s) \times ..... \times \phi_{t_n}(s) = \left(\phi_{t_1}(s)\right)^n$$

$$\phi_t(s) = \left(e^{-(s+\lambda)T} + \frac{\lambda\left(1 - e^{-(s+\lambda)T}\right) \times e^{-(s+\lambda)(T+T')}}{s + \lambda e^{-(s+\lambda)(T+T')}} \right)^n \tag{2.6}$$

$$\phi_t'(0) = -\frac{n}{\lambda}\left(\frac{1 - e^{-\lambda T}}{e^{-\lambda(T+T')}} \right)\left[\phi_t(0)\right]^{n-1}$$

$$\bar{t} = -\bar{\phi}_t(0) = \frac{n}{\lambda}\left(\frac{1 - e^{-\lambda T}}{e^{-\lambda T'}} \right) = \frac{n}{\lambda}\left(e^{\lambda(T_i + T_c + T_o + T_r)} - e^{\lambda(T_o + T_r)} \right) \tag{2.7}$$

We find the optimum checkpointing interval T_{i_opt} that gives the minimum total running time by differentiating the mean total running time with respect to T_i and equating to zero as shown in Eq. 2.8. We set n to be equal to the ratio of T_x and T_i

$$\frac{\partial \bar{t}}{\partial T_i} = \frac{\partial}{\partial T_i}\left(\frac{T_x\left(e^{\lambda(T_i + T_c + T_o + T_r)} - e^{\lambda(T_o + T_r)} \right)}{\lambda T_i} \right) = 0 \tag{2.8}$$

Solving for T_i in Eq. 2.8, we get the optimum checkpointing interval as follows:

$$T_{i\_opt} = \frac{1 - e^{-\lambda\left(T_{i\_opt} + T_c\right)}}{\lambda} \tag{2.9}$$

Eq. 2.9 can be represented in the form $\alpha = 1 - \beta e^{-\alpha}$ where $\alpha = \lambda T_{i\_opt}$ and $\beta = e^{-\lambda T_c}$. From this form of representation, it can be seen that Eq. 2.9 is a transcendental equation and it is impossible to find a solution for α except by defining a new function. There is no analytic solution to this equation to obtain a closed form solution. However, it can be seen on expansion of $e^{-\alpha}$ that α is bound by the limit $\alpha < 1$, which implies that T_{i_opt} is bound by the limit $T_{i\_opt} < \lambda^{-1}$ (i.e, T_{i_opt} < MTBF of the system). Also, since $e^{-\alpha} \le 1$ we have $1 - \beta \le \alpha$. Thus, we have a lower bound on optimum checkpointing interval as $T_{i\_opt} \ge \frac{1 - e^{-\lambda T_c}}{\lambda}$. Hence, we can use numerical methods to solve the equation for optimum checkpointing interval.

Impact of Checkpointing Overhead on I/O Bandwidth

We modeled the optimum checkpointing frequency that provides the minimum overall running time for applications. With the model developed, we study the impact of checkpointing overhead on the sustainable I/O bandwidth of systems in this section. The representative performance numbers used in this study are typical in HPC and supercomputing systems on ground.

In our view, given a system with a specific memory capacity and MTBF, it would be useful to study the I/O bandwidth requirements of the system with respect to the overhead that is imposed by the checkpointing done in the system and the subsequent performance loss in terms of the total execution time of the application. Eq. 2.10 obtains this performance loss as a function of λ, T_i and T_c.

$$F = \frac{T_x\left(e^{\lambda\left(T_{i\_opt}+T_c+T_o+T_r\right)} - e^{\lambda\left(T_o+T_r\right)}\right)}{T_x \times \lambda T_{i\_opt}} = \frac{\left(e^{\lambda\left(T_{i\_opt}+T_c+T_o+T_r\right)} - e^{\lambda\left(T_o+T_r\right)}\right)}{1-e^{-\lambda\left(T_{i\_opt}+T_c\right)}} \tag{2.10}$$

Let F denote the factor of increase in the total running time of the application due to checkpointing overhead and failures while running with optimum checkpoint interval. F is given by the ratio of $\bar{t}$ to T_x as given by Eq. 2.10.

In Eq. 2.10, T_o can be considered negligible compared to the other times. Also, T_r can be considered equal to T_c as both represent the time to move the same amount of data through the same I/O channel. T_c can be given by the ratio of memory capacity to I/O bandwidth. Given a value of F, we can solve for T_{i_opt} by solving a quadratic equation on $e^{\lambda T_i}$.

$$T_{i-opt} = \frac{1}{\lambda} \times \ln\left(\frac{K \pm \sqrt{K^2 - 4M}}{2}\right) \tag{2.11}$$

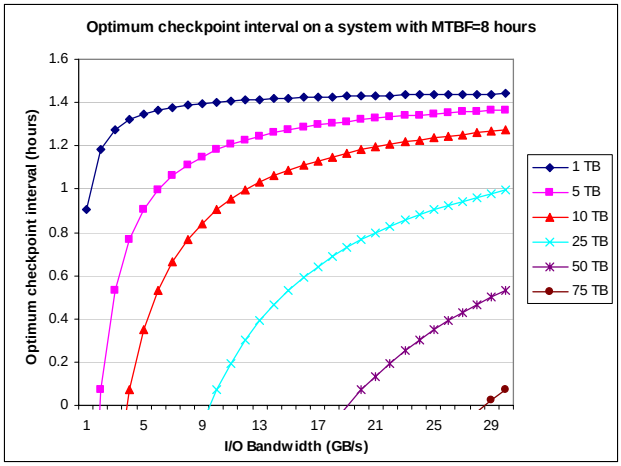
where $K = e^{-\lambda T_c} + Fe^{-2\lambda T_c}$ and $M = Fe^{-3\lambda T_c}$.

Figures 2-9(a) and 2-9(b) show T_{i_opt} and hence the impact of checkpointing on the overall system execution time in systems with MTBF of 8 hours and 24 hours respectively for varying system memory capacities and I/O bandwidth. The MTBF values used in the figures are typical in recent high-performance systems [5]. The value of F is fixed at 1.2 in the figures. We pick 1.2 because F represents the impact of checkpointing on overall execution time and lower values of F (as close to 1 as possible) are certainly desirable. It can be seen from the figures that for systems with low I/O bandwidth, optimum checkpointing is not even possible implying that the allowable overhead (represented by F) is not achievable. As I/O bandwidth of the system is increased, the optimum checkpointing interval also increases. Since the total execution time has been fixed (1.2 times the actual execution time), an increase in checkpoint interval means decrease in the number of checkpoints (n) during the course of the execution. Fewer checkpoints implies less overhead on the system's I/O.

It can be seen from the Figure 2-9 that sustainable I/O bandwidth is a key to obtain optimum overall execution time. As memory capacity of the system increases, so does the requirement of higher I/O bandwidth. For systems with higher memory capacities, it is impossible to find a solution for optimum checkpointing interval to obtain the optimum overall execution time. For example, in a system with a MTBF of 8 hours and memory capacity of 75 TB, there is no solution for the optimum checkpointing interval until the I/O bandwidth is increased to 29 GB/s. The impact is less in systems with higher MTBF values. For a system with similar memory capacity but a MTBF of 24 hours, there exists a solution starting with I/O bandwidth of 10 GB/s. However, an important factor to note is that although solutions exist for the optimum checkpointing interval that gives the minimum total running time, the system might be bogged down by too many checkpoints if the checkpoint interval is low. For example, in a

system with a memory capacity of 75 TB and MTBF of 24 hours, the optimum checkpointing interval is around 10 minutes. Performing large checkpoints at such a high frequency will certainly cause a great load on the system and is not desirable.

A



B

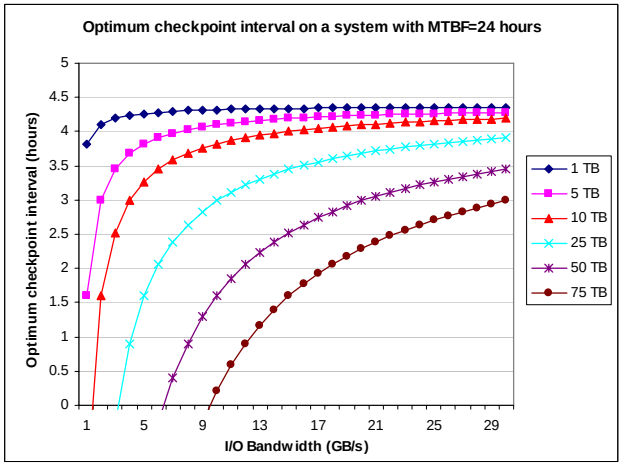


Figure 2-9. Impact of I/O bandwidth on the system execution time and checkpointing overhead A) System with T_{MTBF} = 8 hours. B) System with T_{MTBF} = 1 day.

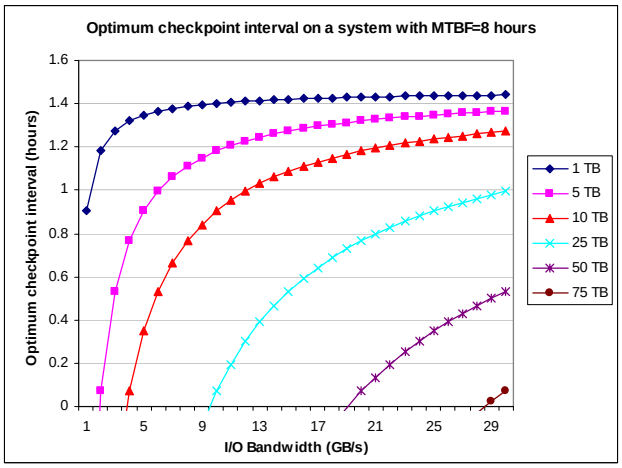
In certain scenarios or systems, the utility of the system within each cycle can be critical. In a given system, let R_1 denote the utility in a cycle (i.e., the ratio of time spent doing useful calculations to the overall time spent in a cycle).

$$R_1 = \frac{T_i}{T_i + T_c} \text{ and when } T_i = T_{i\_opt},$$

$$R_1 = \frac{1 - e^{-\lambda\left(T_{i\_opt} + T_c\right)}}{1 - e^{-\lambda\left(T_{i\_opt} + T_c\right)} + \lambda T_c} \tag{2.12}$$

Eq. 2.12 gives the utility in each cycle when the checkpointing is performed at the optimum checkpointing interval. The checkpointing time T_c can be again given by the ratio of C_{MEM} (memory capacity) and IO_{BW} (sustainable I/O bandwidth). Figures 2-10(a) and 2-10(b) give the utility in a cycle for several memory capacities and I/O bandwidth for systems with MTBF of 8 hours and 24 hours respectively. The value of F is fixed at 1.2. The trend of I/O bandwidth requirement is similar to that in Figure 2-9.

A



B

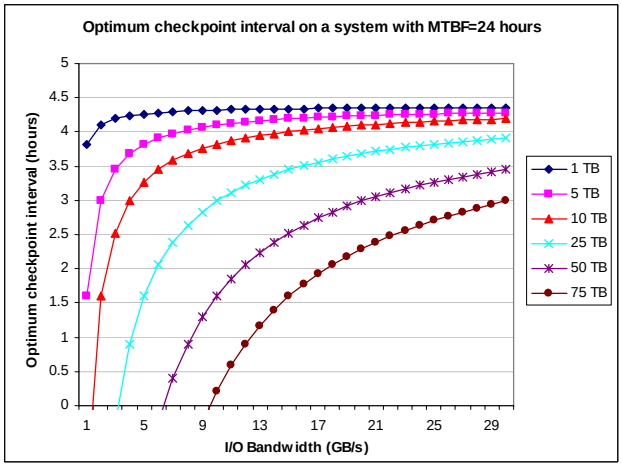


Figure 2-10. Impact of increasing memory capacity in systems on sustainable I/O bandwidth requirement A) System with T_{MTBF} = 8 hours. B) System with T_{MTBF} = 1 day.

It can be seen from the figures, in most of the cases, the utility in a checkpoint cycle is much less than 90% which implies that most of the time within a cycle is spent checkpointing and not doing useful calculations. Although not shown in the figure, it was found that for a system with MTBF of 8 hours and memory capacity of 75 TB, an I/O bandwidth of about 150 GB/s is required to utilize at least 90% of a checkpoint cycle on useful calculations. Such high values of I/O bandwidth are much higher than what is available for present systems. The fact that the system cannot even provide useful computations in many cases shows the gravity of the situation.

We see from Figure 2-10 that the utility within a cycle almost saturates beyond a certain I/O bandwidth. The I/O bandwidth at which the percentage utility begins to flatten out is what is desirable for the system both present and future. For example, I/O bandwidth of around 10 GB/s would be desirable for a system with a memory capacity of 1 TB and MTBF of 8 hours. For a similar system with MTBF of 24 hours even a lesser I/O bandwidth (around 5 GB/s) would suffice. But as the memory capacity of the system increases, the I/O bandwidth desirable is dramatically higher.

2.4 Simulative Verification of Checkpointing Model

Simulation is a useful tool to observe the dynamic behavior of a system as its configuration and components change. As preliminary verification, Monte Carlo simulations were performed on the analytical model. The simulation and analytical results matched closely verifying the correctness of the model mathematically. However, for more accurate verification of the model, we develop simulation models to mimic super and embedded computing environments using Mission-Level Designer (MLD), a discrete-event simulation platform developed by MLDesign Technologies Inc. Section 2.4.1 presents the system model used to gather results, and Sections 2.4.2-2.4.6 provide details about each major component model.

2.4.1 System Model

Figure 2-11 displays the system model that we employed to conduct the simulation experiments. The system consists of four key component models, multiple nodes, network, central memory, and fault generator. Each component has associated parameters that are user-definable in order to model different system settings. The components and their parameters are discussed in detail in the subsequent sections.

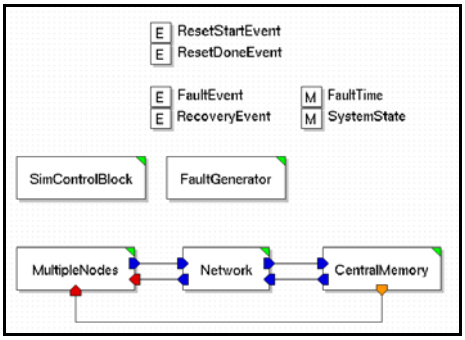


Figure 2-11. System model

2.4.2 Node Model

A node is defined as a device that processes and checkpoints data, and is prone to failures. Figure 2-12 shows the MLDesigner node model. The behavior of the node is modeled such that it checkpoints its entire main memory at a specified time period. The time spent between checkpoints represents useful computation, communication, and productive I/O time used to complete a specific task (see Computation section in Figure 2-12). After a checkpoint has been successfully completed, the nodes can continue processing data.

The node model was designed in order to provide an abstract representation of a clustered processing element that runs a parallel job along with the other nodes in the system. If a single node in the system fails, all the nodes must recover from the last successful checkpoint to ensure data integrity. The statistics gathered at the node level (see Statistics section in Figure 2-12) include completed computation time and lost computation time due to a failure. The user can

define numerous parameters for the node model including checkpoint interval, main memory size, and application completion time.

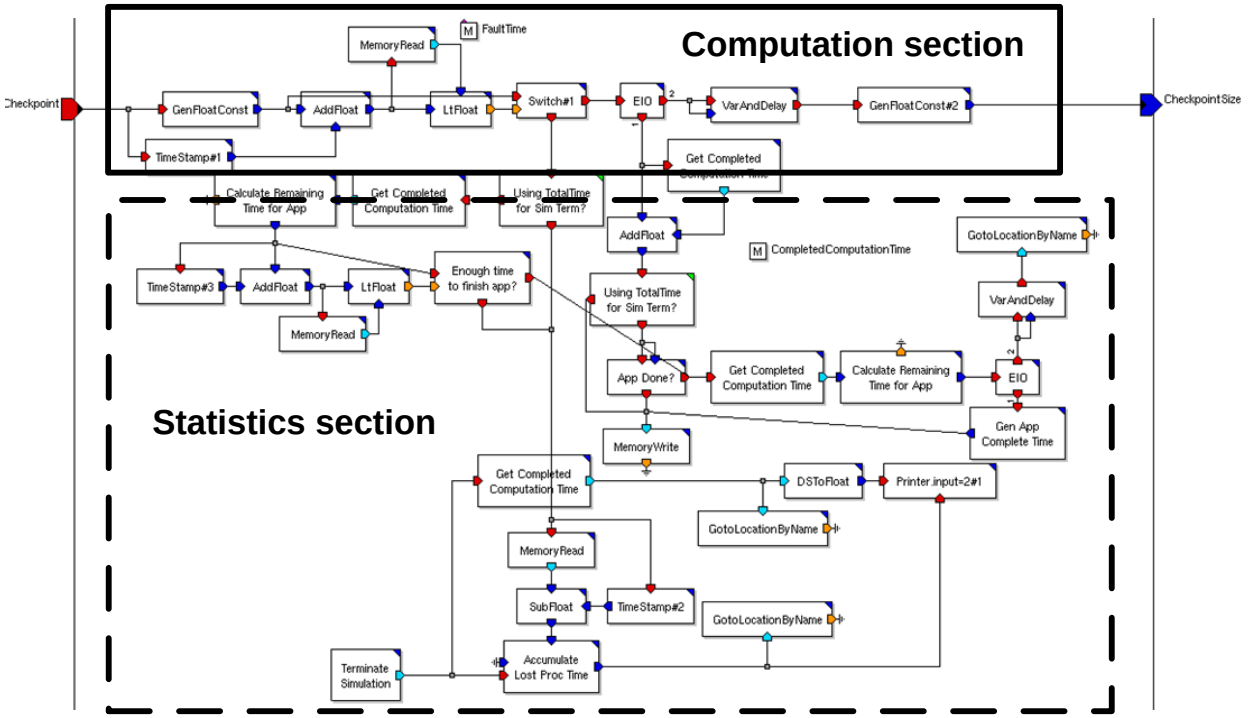


Figure 2-12. Node model

2.4.3 Multiple Nodes Model

The multiple nodes model, illustrated in Figure 2-13, uses a capability of the MLDesigner tool, dynamic instantiation, that allows a single block to represent multiple instances of a model. The node model described in Section 2.4.2 is dynamically instantiated in the multiple nodes model. The technique is used to ease the design and configuration procedure used to model large homogenous systems. The main function of the multiple nodes model is to ensure global synchronous checkpoints and to collect statistics. The statistics gathered include completed checkpoint time and total checkpoint time lost. That is, it records the amount of time taken to successfully complete a checkpoint and also the amount of time lost when a failure occurs during a checkpoint.

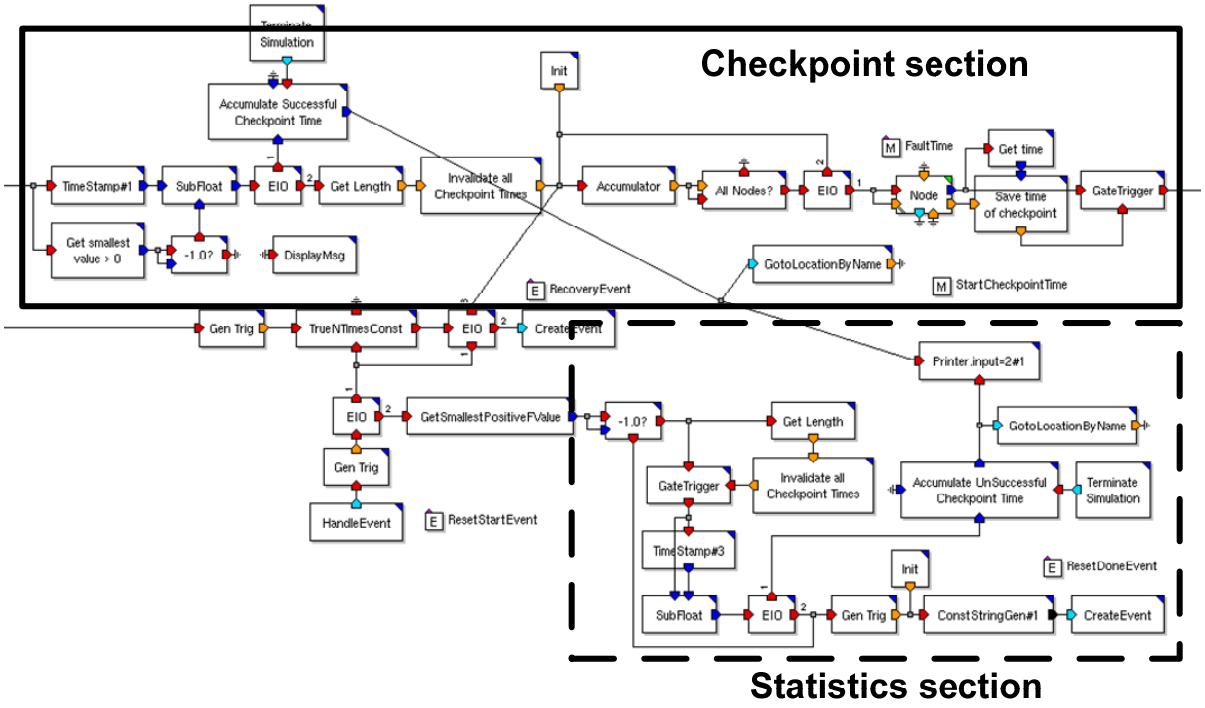


Figure 2-13. Multiple nodes model

2.4.4 Network Model

The network model is a coarse-grained representation of a generic switch- or bus-based network. The model uses user-defined latency and effective bandwidth parameters to calculate network delay based on the size of the transfer. Figure 2-14 illustrates the network model.

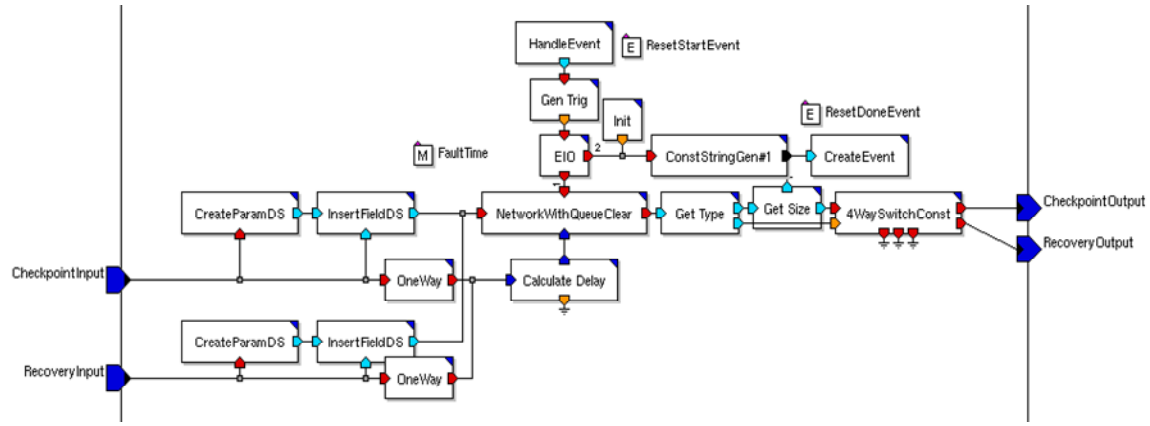


Figure 2-14. Network model

2.4.5 Central Memory Model

The central memory model is another coarse-grained model that is used as a mechanism to represent the storing and restoring of checkpointed data. The model accepts each checkpoint request and sends a reply when the checkpoint is completed. When a node fails, each node is sent its last successful checkpoint after some user-definable recovery time which represents the time needed for the system to respond to the failure. No actual data is stored in the central memory since the simulated system does not actually process real data. Also, the central memory is assumed to be large enough to hold all checkpoint data, therefore overflow is not considered. Figure 2-15 illustrates the central memory model.

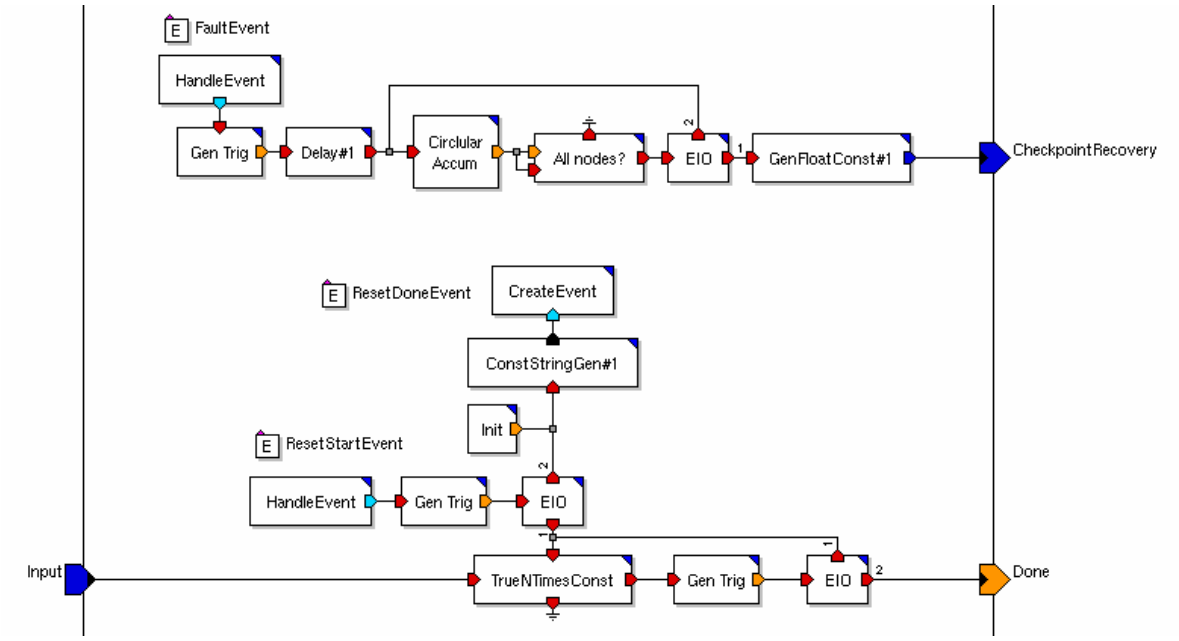


Figure 2-15. Central memory model

2.4.6 Fault Generator Model

When a failure occurs in a component, the system must recover using a predefined procedure. This procedure starts by halting each working node followed by the transmission of the last checkpoint by the central memory model. When the failed node recovers and all nodes receive their last checkpoint, the system begins processing once again. The fault generator

model shown in Figure 2-16 controls this process by creating a “failure event” at some random time based on an exponential distribution with a user-definable time (i.e., MTBF) and orchestrating the recovery process of the components in the system. For example, when a failure occurs, the fault generator will pass a notice to the node models to let them know that they must recover. It also tracks the status of each node regarding their recovery status (e.g. recovered or recovering).

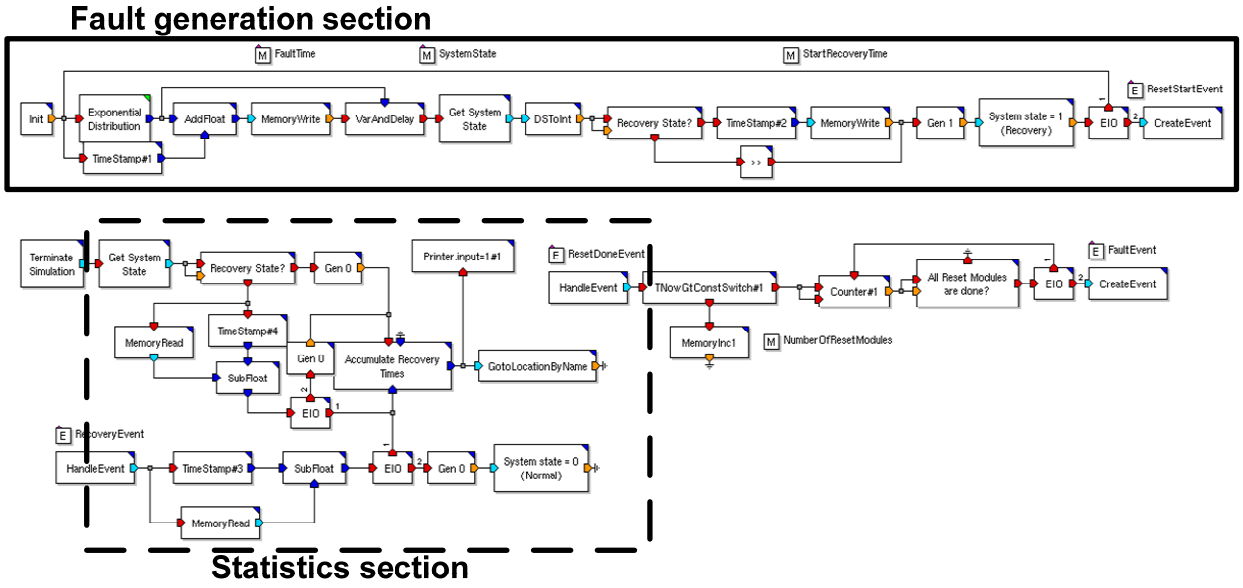


Figure 2-16. Fault generator model

2.4.7 Simulation Process

Total execution time of the application, MTBF of the system, sustainable I/O bandwidth, amount of memory to be checkpointed and frequency of checkpointing are input parameters to the simulation model and are variable. Faults are generated in the system based on the MTBF value input. After every checkpointing period, data (equal to the size of memory specified) is checkpointed to the central memory. The time to checkpoint is dependent on the amount of checkpoint data and the I/O bandwidth values. While recovering from a failure, the nodes in the system collect the data from the central memory and the transfer time is again dependent on the

data size and the I/O bandwidth value. If a failure occurs in the system during the data recovery process, the transfer is reinitiated and the process is repeated until successful.

2.4.8 Verification of Analytical Model

In order to verify the correctness of our analytical model, we simulated the checkpointing process in several systems with memory ranging from 5 TB to 75 TB and I/O bandwidth ranging from 5 GB/s to 50 GB/s, running applications with execution times ranging from 15 days to 6 months (180 days). For each combination of values for the above parameters, the system was simulated for MTBF values of 8 hours, 16 hours and 24 hours. The numbers listed above are typical of high-performance computers found in several research labs and production centers. In this section, we pick a few systems and show the simulation results to verify the analytical model for optimum checkpointing frequency.

Figure 2-17 shows the results from simulations of the system running applications with fault-free execution time of 15 days (15×24=360 hours). For a given value of checkpointing interval (T_i), the y-axes in the figure give the application completion time (i.e., total time for the application to complete with faults occurring in the system). Hence, the optimum checkpoint interval is the value for which the application completion time is the least with the given system resources. Based on the completion time given by Figure 2-17(a), for a system with 5 TB memory and 5 GB/s sustained I/O bandwidth, the optimum checkpoint intervals are around 2 hours, 3 hours and 4 hours respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. Similarly, for a system with 5 TB memory and 50 GB/s sustained I/O bandwidth, the optimum checkpoint intervals as shown by Figure 2-17(b) are 0.5 hours, 1 hour and 1 hour respectively when the MTBF values are 8 hours, 16 hours and 24 hours.

In order to verify the correctness of the analytical model, Eq. 2.9 needs to be solved to find T_{i_opt} given the same system parameters used for simulation in Figure 2-17. We use the fixed

point numerical method [26] to solve Eq. 2.9. Such strategies are quite common in solving equations as not all problems yield to direct analytical solutions [27]. Fixed point iteration is a method for finding roots of an equation $f(x) = 0$ if it can be cast into the form $x = g(x)$ as is Eq. 2.9. A fixed point of the function $g(x)$ is the solution of the equation $x = g(x)$ and the solution is found using an iterative technique. A value is supplied for the unknown variable in the equation and the process is repeated until an answer is achieved.

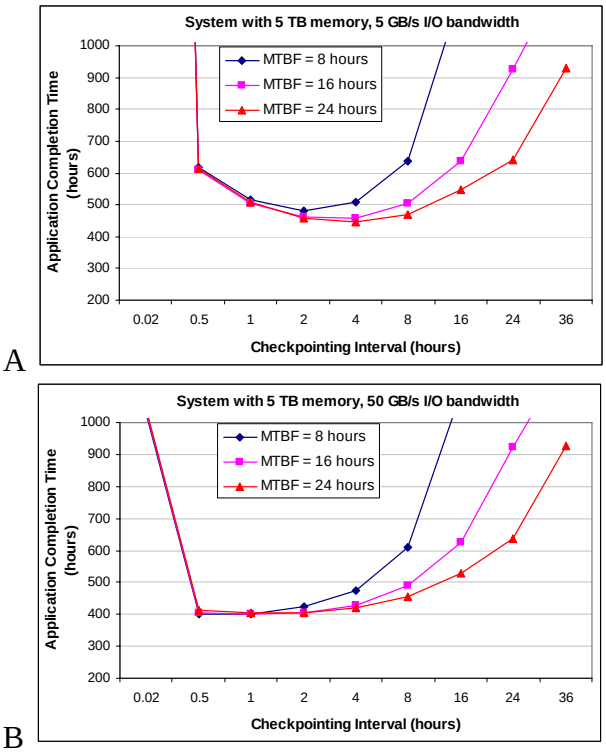


Figure 2-17. Optimum checkpointing interval based on simulations of systems with 5 TB memory A) System with /O bandwidth of 5 GB/s. B) System with I/O bandwidth of 50 GB/s.

In order to show the validity of using numerical method to solve our analytical model, we show the results from the fixed point iterative method that we used in Figure 2-18. The system used for illustration in the figure is the one with 5 TB of memory and 5 GB/s of I/O bandwidth. For Eq. 2.9 to have a solution (i.e., for a given checkpoint interval to be the optimum), the left-

hand side (LHS) is equal to the right-hand side (RHS) of the equation. For a given value of checkpointing interval (T_i), the y-axis in Figure 2-18 referred to as the error in solution give the difference between the LHS and the RHS of Eq. 2.9. Hence, the optimum checkpoint interval is the one for which the error is zero. The checkpointing interval values used in our iterative technique are bound by the lower and upper limits discussed in Section 2.3. The optimum checkpoint intervals identified using the numerical method in Figure 2-18 for the system are 2 hours, 3 hours and 3.5 hours respectively when the MTBF is 8 hours, 16 hours, and 24 hours. The results of the analytical model are then cross compared with the solutions from the simulation model shown in Figure 2-17. It can also be seen that the solutions are well below the upper bound (MTBF) derived in Section 2.3. Section 2.5 provides the results from the analytical model for a wide range of systems.

Solving our analytical model for a system with 5 TB of memory and 5 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 2 hours, 3 hours and 3.5 hours respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. For a similar system with 5 TB of memory but 50 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 0.5 hours, 0.75 hours and 1 hour respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours.

Based on the simulation results for optimum checkpoint intervals in Figure 2-17, it can be seen that the analytical results match that of simulation. According to both analytical and simulation models, for a system with 5 TB of memory and 5 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 2 hours, 3 hours and 4 hours respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. Likewise for a system with 5 TB of memory but 50 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 0.5

hours, 1 hour and 1 hour respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. The analytical model is thus verified by the simulation model.

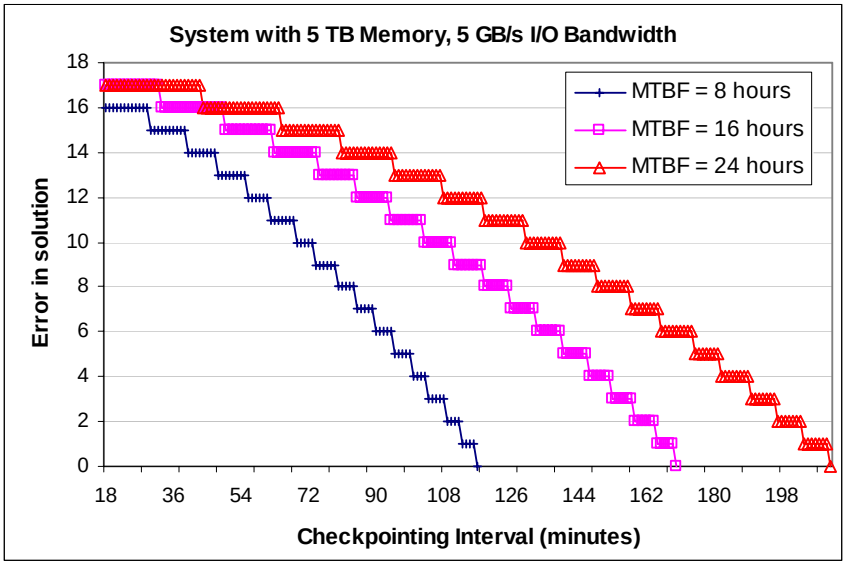


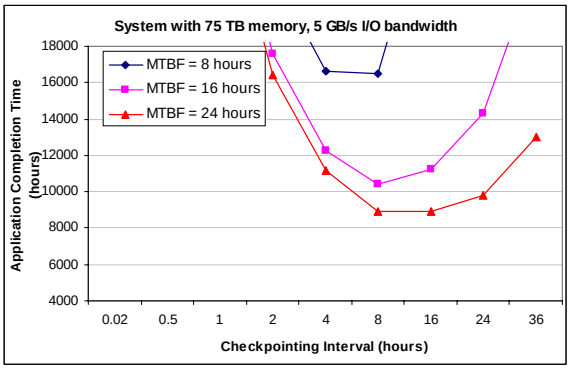
Figure 2-18. Numerical method solution for the analytical model

Eq. 2.9 is independent of the total execution time of the application. In order to verify this behavior of the analytical model, we ran simulations for applications with larger execution times. Figure 2-19 shows the results from simulations of the system running applications with fault-free execution time of 180 days (180×24=4320 hours). We also wanted to verify that the analytical model holds true for systems with parameters different from those used. Hence the system shown in Figure 2-19 has 75 TB of memory which is close to the high end of presently existing supercomputers.

Simulation results in Figure 2-19(a) show that for a system with 75 TB of memory and 5 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 6 hours, 9 hours and 12 hours respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. For the same system, the analytical model also provides the same optimum checkpointing intervals as can be seen from figures in Section 2.5.

Simulation results in Figure 2-19(b) show that for a system with 75 TB of memory but 50 GB/s of I/O bandwidth, the optimum checkpointing intervals are around 2 hours, 3 hours and 4 hours respectively when the MTBF of the system is 8 hours, 16 hours and 24 hours. Analytical results for the same system matches the simulation results. Thus, it can be seen from Figures 2-17 through 2-19 that the analytical results match closely with the simulation results, further verifying the correctness of the analytical model.

A



B

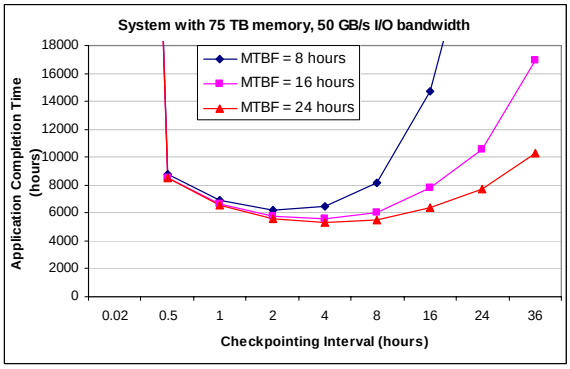


Figure 2-19. Optimum checkpointing interval based on simulations of systems with 75 TB memory A) System with /O bandwidth of 5 GB/s. B) System with I/O bandwidth of 50 GB/s.

2.5 Optimal Checkpointing Frequencies in Typical Systems

The results shown in Section 2.5 were primarily used to verify the analytical model. In this section, we show results derived from the analytical model for a wide range of systems with parameter values representing large supercomputers traditionally developed for HPC. The results provide insight about the checkpoint intervals that should be used in typical scenarios. In

order to analyze the applicability of our model to a broader spectrum of systems, we also studied small systems intended for high-performance embedded computing (HPEC) in space. The results provide insight about the checkpoint intervals that should be used in typical scenarios.

It should be mentioned here that at present there are no existing systems for HPC in space. However there are initiatives for developing such systems such as the one at the High-performance Computing and Simulation Research Laboratory at UF in collaboration with NASA and Honeywell Inc [1]. The values supplied for the parameters (resources) in our studies are inspired by the system at University of Florida and hence fall in that range.

2.5.1 Traditional Ground-based HPC Systems

We show the optimal checkpointing frequency results for three different cases with parameter values representing traditional ground-based HPC systems in Figure 2-20. The optimal checkpointing frequencies are calculated using the same method discussed in the previous section (i.e., fixed point iteration method). Figure 2-20(a) gives the results for a system at the lower end with 5 TB memory while Figure 2-20(b) gives the results for a system at the other extreme with 75 TB memory. These higher-end resources are similar to what an application such as the Crestone project discussed earlier would solicit. The results for a midrange system with 30 TB memory is shown in Figure 2-20(c). The optimum checkpointing interval for typical HPC systems is in the order of hours.

Increase in MTBF implies lesser chances of faults and hence it is not required to checkpoint often. It can be seen from Figure 20 that as the MTBF of the system increases the optimum checkpointing interval also increases. However, the difference is much pronounced only in systems with less I/O bandwidth. As the I/O bandwidth of the system is increased, the difference in optimum checkpointing intervals for varying MTBF decreases. It can also be observed from the figure that for a system with a given memory capacity and MTBF, the

optimum checkpointing interval decreases as the I/O bandwidth of the system is increased. This trend implies that it is better to checkpoint at a higher frequency if the time to checkpoint lowers. This observation can also be ascertained by the fact that for the same I/O bandwidth the optimum checkpointing interval increases as the memory capacity of the systems increases (implying that it is optimal to checkpoint at a lesser frequency if checkpointing time increases).

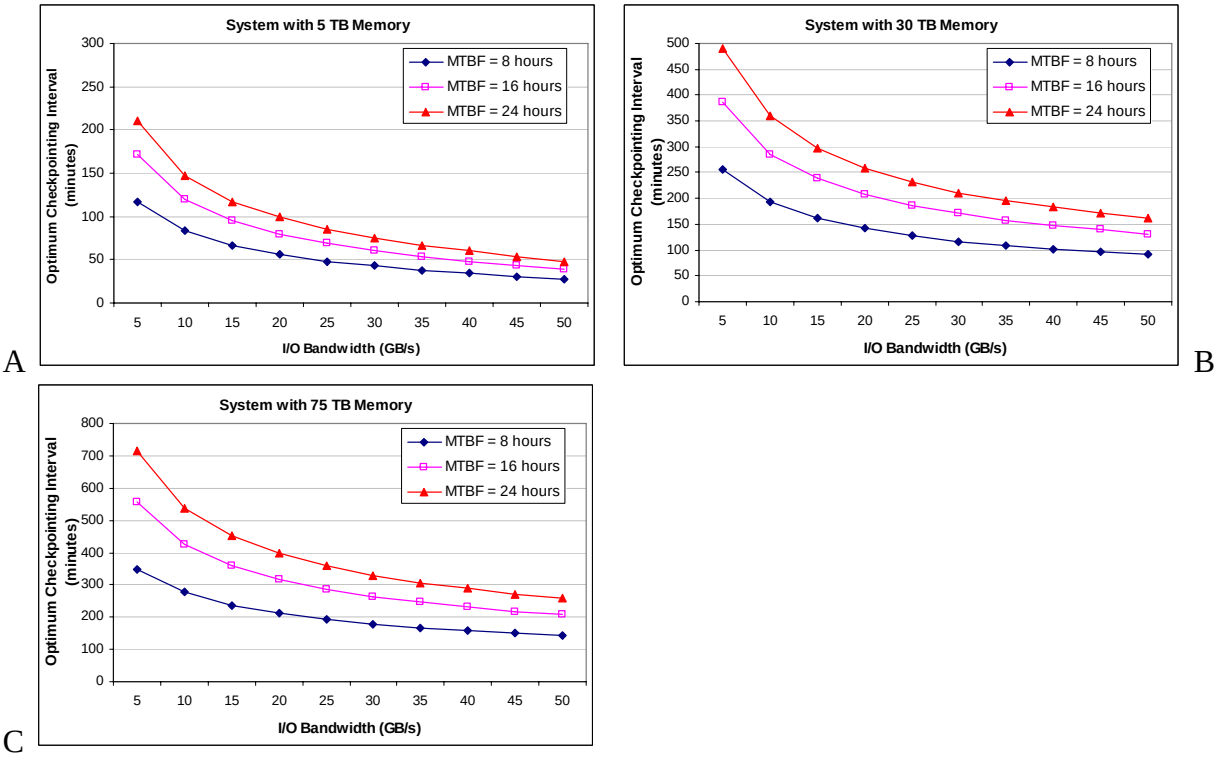


Figure 2-20. Optimum checkpointing interval for various systems A) System with 5 TB memory. B) System with 30 TB memory. C) System with 75 TB memory.

2.5.2 Space-based HPC Systems

Figure 2-21 shows the optimal checkpointing frequencies for systems with parameter values in the range of HPC systems in space. Figure 2-21(a) gives the results for a system at the lower end with 512 MB memory while Figure 2-21(b) gives the results for a system at the other extreme with 8 GB memory. The results for a midrange system with 2 GB memory is shown in

Figure 2-21(c). The optimum checkpointing interval for typical HPEC systems is in the order of minutes.

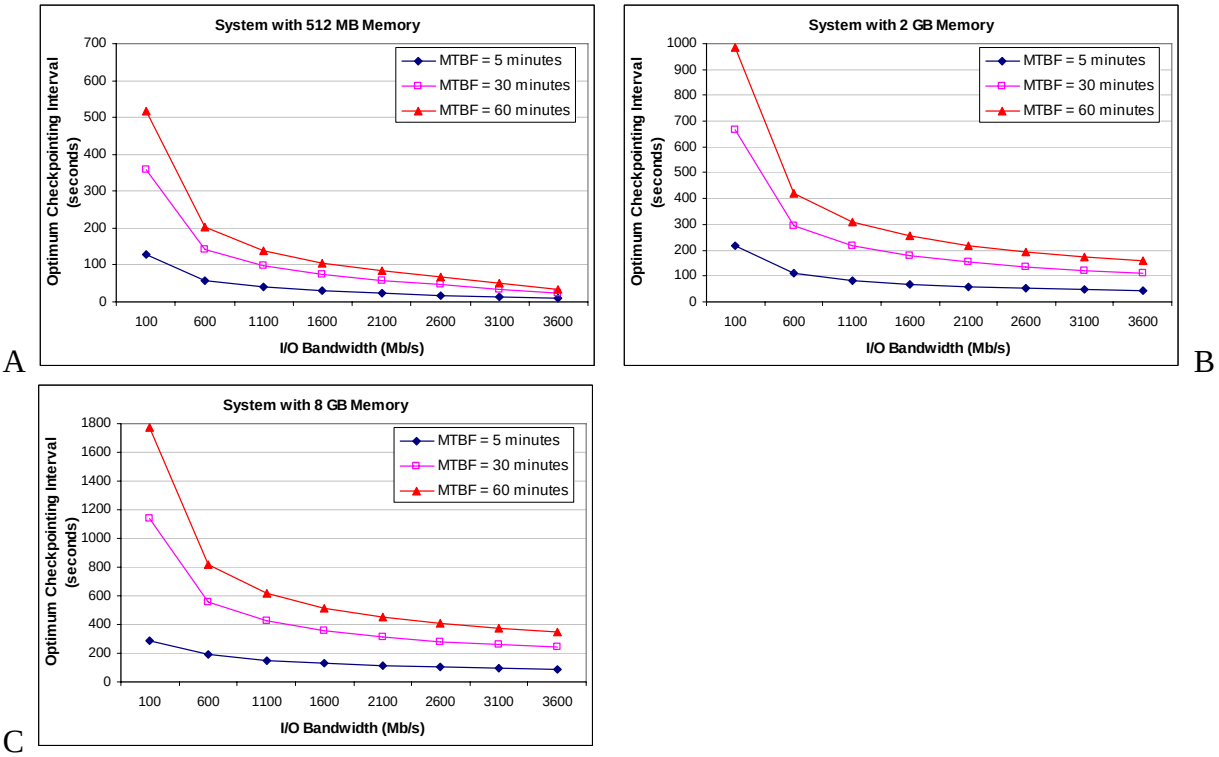


Figure 2-21. Optimum checkpointing interval for various systems in space A) System with 512 MB memory. B) System with 2 GB memory. C) System with 8 GB memory.

It can be seen from Figure 2-21 that the behavior of the space-based HPEC systems is similar to the ground-based HPC systems shown in Figure 2-20 although HPEC systems have much less resources and high failure rate. For a given system, the optimal checkpoint interval increases as the MTBF value is increased. Likewise for a system with a given MTBF and I/O bandwidth, the optimum checkpoint interval increases as the memory of the system increases. As I/O bandwidth of the system is increased with other system parameters kept constant the optimum checkpoint interval decreases.

2.7 Conclusions and Future Research

In this phase of the research, the useful computation time of the system is improved (increased) by optimizing checkpointing-related defensive I/O. We studied the growth in performance of the various technologies involved in high-performance computing, highlighting the poor performance growth of disk I/O compared to other technologies. Presently, defensive I/O that is based on checkpointing is the primary driver of I/O bandwidth, rather than productive I/O that is based on processor performance. There are several research efforts to improve the process of checkpointing itself but they are generally application-specific. To our knowledge, there have been no successful attempts to optimize the rate of checkpointing, which is our approach to reduce the overhead of fault-tolerance. Such an approach is not application- or system-specific and is applicable to both HPC systems in space and on ground. The primary contribution of this phase of research is the development of a new model for checkpointing process and in so doing identify the optimum rate of checkpointing for a given system. Checkpointing at an optimal rate reduces the risk of loosing much computation on a failure while increasing the amount of useful computation time available.

We developed a mathematical model to represent the checkpointing process in large systems and analytically modeled the optimum computation time within a checkpoint cycle in terms of the time spent on checkpointing, the MTBF of the system, amount of memory checkpointed and sustainable I/O bandwidth of the system. Optimal checkpointing maximizes useful computation time without jeopardizing the applications due to failures while minimizing the usage of resources. The optimum computation time leads to the minimal wastage of useful time by reducing the time spent on overhead tasks involving checkpointing and recovering from failures.

The model showed that the optimum checkpointing interval is independent of duration of application execution and is only dependent on system resources. In order to see the impact of checkpointing overheads and the overhead to recover from failures on the I/O bandwidth required in the systems, we analyzed the overall execution time of applications including these overhead tasks relative to the ideal execution time in a system without any checkpointing or failures. Results pertaining to the time spent on useful calculations between checkpoints suggest the need for a very high sustainable I/O bandwidth for future systems. A system with a projected MTBF of 1 day (24 hours) and a memory capacity of 75 TB, which may be typical for a system in the near future, would require a sustainable I/O bandwidth of about 53 GB/s to effectively use 90% of time on useful computations. For a similar system with a projected MTBF of 8 hours, the required I/O bandwidth would be 150 GB/s for the same performance.

In order to verify our analytical model, we developed discrete-event simulation models to simulate the checkpointing process. In the simulation models, we used performance numbers that represent systems ranging from small embedded cluster systems (space-based) to large supercomputers (ground-based). The simulation results matched closely with those from our analytical model. Finally, we also showed the optimum checkpointing interval for a wide range of HPC and HPEC systems. The results derived from the analytical model showed that irrespective of the duration of the application's fault-free execution, the optimum checkpointing interval is in the orders of hours for HPC systems and in the order of minutes for HPEC systems. For a system with a given MTBF, the checkpointing time determined by the ratio of memory capacity and I/O bandwidth is the primary factor influencing the optimum checkpointing interval.

As future work, the model can be experimentally verified with real systems running scientific applications. When successfully verified, the model can be used to find the optimal checkpointing frequency for various HPC and HPEC systems based on their resource capabilities.

CHAPTER 3
EFFECTIVE TASK SCHEDULING (PHASE II)

Several techniques exist for traditional HPC to provide fault tolerance and improve overall computation rate. We discussed the option of periodic backup of data at an optimum frequency during the first phase of this dissertation. In this next phase of the dissertation we focus on exposing computation to lesser faults by reducing overall execution time of applications through effective task scheduling of applications for HPC systems in space. We propose to analyze techniques to effectively schedule tasks on parallel hardware reconfigurable resources that could be part of the HPC space system.

3.1 Introduction

Recently, the processing speed and efficiency requirements of HPC applications have driven research efforts in a different direction toward systems augmented with customizable hardware such as Field-Programmable Gate Arrays (FPGAs). Application-Specific Integrated Circuits (ASICs) almost always outperform General-Purpose Processors (GPPs) in terms of raw performance for a given application but tend to have a much higher development cost and cannot be adapted once developed. By contrast, RC offers flexibility at multiple levels by exposing the raw building blocks to form complex logic operations optimized for the task at hand. In addition, when a task is completed, the same hardware can be reconfigured to perform a completely different task. FPGA-based dynamically reconfigurable computing can be more cost-effective and flexible than ASIC solutions and simultaneously faster than GPP-based conventional computing. RC has gained in popularity recently and there are numerous ongoing RC-related research efforts at various institutions in addition to a large following in the HPC vendor community with products from Cray, SRC and SGI already deployed. Empowering clusters and supercomputers with customizable and dynamically reconfigurable hardware

resources is a potential solution to address the requirements of HPC applications in terms of enormity, processing speed and efficiency. The strengths of GPPs and FPGAs compliment each other and an efficient merger can produce a powerful combination. However, this area of research is still relatively new and much research is required to bring the tools and runtime environment to a level of maturity seen in traditional HPC systems.

HPC systems in space might potentially involve such FPGA-based reconfigurable computing alongside traditional HPC based on GPPs. The reason is two-fold: the first one being that FPGAs fit well with the embedded systems mentality of space computing and the second being the improved speedup in the execution of tasks. We followed a modeling-based approach in the first phase to improve the useful computation time. In this phase, we reduce the execution time of tasks thereby exposing them to lesser number of faults. However, reducing execution time would require effective scheduling of tasks. Since FPGA-based parallel computing is a relatively new field of research, there are not many standardized and well researched scheduling mechanisms for scheduling RC applications.

In this phase of the dissertation, we explore some initial steps toward the goal of effective task scheduling in space-based HPC systems enhanced with reconfigurable hardware accelerators. Developing a robust service for automated scheduling, execution, and management of tasks including RC tasks on network-based clusters, or any scalable distributed system, is a major challenge. Performance modeling, dynamic scheduling, and management of distributed parallel RC applications and systems have not been well studied to our knowledge. We take a first step towards the solution of automated job management by analyzing via simulation the performance of several traditional scheduling heuristics on the computing resources as part of a future automated job management service for parallel RC systems such as [28]. We test these

heuristics by executing typical applications used in the evaluation of Department of Energy (DOE) systems on simulated RC resources in space. We also develop a performance model to predict the overall execution time of tasks on RC resources used by our scheduling heuristics to schedule these tasks. With this analysis, we progress towards our primary goal of fault tolerance. Additionally studying the scheduling of tasks on RC processing elements has similarities to and hence helps task scheduling on traditional processing elements involved in HPC in space.

The rest of the chapter is organized as follows. A brief background on RC and information on representative RC resources and applications used in our simulations is provided in the second section. In the third section, we develop the performance model used for our scheduling heuristics and the various heuristics that we compare in this phase of research. We discuss the simulation results and compare the different heuristics in the fourth section. Finally, the conclusions and contributions are presented in the fifth section.

3.2 Background

A general background of RC is not included here. The topic has been well presented by many sources such as Compton and Hauck [29] and Enzler, Plessl and Platzner [30]. In this section, we provide background information on the RC resources and HPC applications used to drive our simulations.

3.2.1 Representative RC Systems

As part of this work, we studied several RC systems including PCI-based boards in clusters of workstations, the Cray XD1 with a fast message-passing interconnect between processors and FPGAs, and SGI's RASC for the Altix350 which features a global shared-memory system between all system components. The speedups that the RC systems provide in our simulations are based on empirical measurements of their characteristics and performance on sample applications. A brief description of the RC systems studied follows.

We studied three boards attached to host machines as peripheral components via a PCI slot: BenNUEY, RC1000, and CPX2100. The BenNUEY [31] RC board from Nallatech hosts a Virtex-II 6000 FPGA and an attached BenBLUE-II daughter card with two Virtex-II 6000s for a total of three Virtex-II 6000s. For a PCI-based board, the BenNUEY provides a fairly substantial amount of processing power. The architecture allows for DIME module expansion of up to six Virtex-II 8000s in addition to the FPGA on the BenNUEY in a tightly-coupled package. The RC1000 from Celoxica [32] contains a Virtex 2000E FPGA. This PCI-based board includes a front-side memory interface that can provide significant storage and direct-memory access capability while using minimal FPGA resources in contrast with boards having off-chip memory behind the FPGA. The CPX2100 [33] from Tarari contains three Virtex-II 2000s, although only two are available for user designs. The third FPGA is used for memory management, PCI bus management, and other control processes allowing the other two FPGAs to be used for user designs more efficiently.

SGI's Altix family, built upon Intel Itanium 2 processors and the Linux OS, scales up to thousands of processors via their NUMAflex global shared-memory architecture. The Altix line also employs the NUMAlink message-passing interconnect with 1 μs MPI short-message latency and 3.2 GB/s unidirectional bandwidth per link. SGI has integrated a Virtex-II 6000 FPGA, known as the Reconfigurable Application-Specific Computing (RASC) module [34], directly to the NUMAlink fabric, allowing the RASC hardware to integrate to systems up to 512 processors and beyond. Cray has also created a system with RC capabilities in the XD1 line by integrating an FPGA directly onto each 2- or 4-way symmetric multiprocessor node's local RapidArray connection. Two system flavors exist today, one including Virtex-II Pros and another including Virtex-4 FPGAs. The FPGA considered in our simulations belongs to the Virtex-4 category.

3.2.2 Representative Applications

The representative applications that were used in our simulations were chosen to possess heterogeneous characteristics in terms of execution time of individual tasks, memory locality, computational intensity (ratio of computation to memory access), etc. The choice of applications and their performance numbers are based on the performance evaluation performed by Vetter et al. [35]. It should be mentioned that we do not consider the in-depth characteristics of the applications and only consider their comparative performance pattern. Also, the applications chosen are traditional HPC applications that have well studied before and are not specific to space computing. The reason is that space applications on HPC and RC environments are not well studied yet and there are no standard performance evaluations of these applications to our knowledge. We use seven benchmark applications (many of them are used on ASC[1] platforms of the Department of Energy) including sPPM, SMG2000, SPHOT, IRS, UMT, AZTEC, and SWEEP3D. The applications are truly scalable, scaling to thousands of processors, and some have executed on platforms using as many as 8000 processors. A coarse-grained, distributed memory model is used by all the applications. Below is a brief description of the applications as described by [35].

- **sPPM** [36] solves three-dimensional gas dynamics problem (compressible Euler equations) using a simplified Piecewise Parabolic Method (PPM). PPM is a finite volume technique in which each grid point uses the information at the four nearest neighbors along each spatial dimension to update the values of its variables.
- **SMG2000** [37] is a parallel semicoarsening multigrid solver for linear systems arising from finite difference, volume, or element discretizations of the diffusion equation on logically rectangular grids.

1. Advanced Simulation and Computing (ASC) Program earlier known as Accelerated Strategic Computing Initiative (ASCI) is the National Nuclear Security Administration (NNSA) collaborative program among DOE's several national laboratories.

- **SPHOT** is a two-dimensional photon transport code to track particles through a logically rectangular 2-D mesh. Monte Carlo transport solves the Boltzmann transport equation by directly mimicking the behavior of photons.

- **Sweep3D** [38] solves a three-dimensional, time-independent, particle transport equation on an orthogonal mesh using a multidimensional wavefront algorithm for deterministic particle transport simulation.

- **IRS** [39] is an implicit radiation solver code to solve radiation transport by the flux-limited diffusion approximation using an implicit matrix solution.

- **UMT** is a three-dimensional, deterministic, multigroup, photon transport code for unstructured meshes solving first-order form of the steady-state Boltzmann transport equation.

- **AZTEC** [40] is a parallel iterative library for solving linear systems.

3.3 Task Scheduling

Scheduling is the process of assigning tasks to required resources. Efficient task scheduling is a critical part of automated management and is imperative to maximize application performance and system utilization. Scheduling can be broadly classified as static scheduling wherein tasks are assigned resources prior to the start of application execution, and dynamic scheduling wherein tasks are assigned resources dynamically while applications are executing. In general, dynamic scheduling often implies that tasks are migrated between resources while executing based on the availability and performance of the various resources at hand. Static scheduling typically simplifies runtime management requirements but often leads to inefficient resource utilization. Dynamic scheduling can improve overall application execution times but requires a more complicated runtime management mechanism, especially in a heterogeneous environment. Another form of scheduling which blends the two previous schemes schedules tasks as and when they arrive (no fixed static schedules) but without any dynamic task migration capabilities. We choose this intermediate approach for our simulations to provide the balance between efficiency and complexity.

Task scheduling can be further categorized into divisible workload scheduling and independent task scheduling. In divisible workload scheduling, the workload can be divided into arbitrary sized chunks for scheduling. This type of division of workload enables flexible scheduling and hence can be more efficient. However, in general, it may be difficult to divide a workload into chunks of any arbitrary size. By contrast, independent task scheduling schedules the workload that has been already divided into independent tasks and this is the scheme we adopt for our simulations. Our simulations only focus on resource-level scheduling (i.e., a task scheduled to a resource owns the resource until completed, at which time other tasks may use the resource).

3.3.1 Scope of the Scheduler

Jobs represented by a task-data dependency graph such as a directed acyclic graph (DAG) are submitted to a Job Manager (JM) component of the automated management service. Figure 3-1 shows DAGs for two sample jobs.

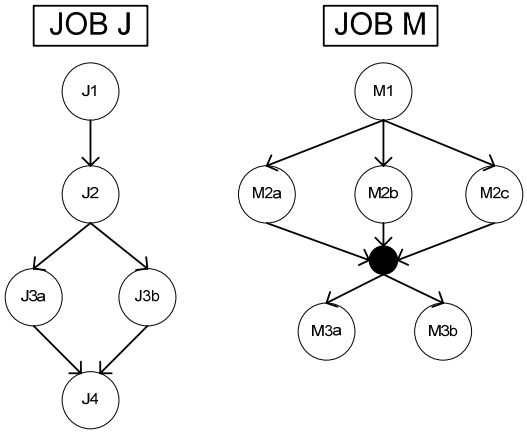


Figure 3-1. Task dependence using directed acyclic graphs for sample jobs

The JM provides the scheduler with the task details (i.e., performance requirements and the order of execution of tasks). The tasks are scheduled in the same order as they are represented in the DAG. If multiple tasks can be executed simultaneously (i.e., if the tasks are in the same level

on the DAG), then the tasks can be scheduled in a batch mode. The communication dependencies between the tasks (that might be executing simultaneously) are not considered in the performance model that we use to schedule the tasks.

3.3.2 Performance Model

Tasks are scheduled on a set of machines (we use the terms machines and resources interchangeably) based on the performance of the tasks on the machines. Generally, in any scheduler, the job of predicting task performance is based on a model which when supplied with the parameters representing both tasks and machines gives the expected performance of the tasks on the machines. Many such performance models exist for traditional HPC systems but few for RC systems. The models that exist are too complex for simulative studies of scheduling heuristics [41]. Hence, in this section, we develop a performance model to predict the overall execution time of a task on a machine given the characteristics of the task and the machine. We use this performance model for our simulations. Eq. 3.1 through 3.5 give the performance model that we develop for our system.

$$OET = T_{HCOMM} + T_{HCOMP} + T_{BCOMM} + T_{BCOMP} \tag{3.1}$$

OET : Overall execution time
T_{HCOMM}, T_{HCOMP} : Communication and computation time involving the host machine
T_{BCOMM}, T_{BCOMP} : Communication and computation time involving the RC board

$$T_{BCOMP} = T_{CONF} + T_{BOARD} \tag{3.2}$$
$$T_{BCOMM} = T_{BCF} + \left(S_{DATA} \times N_{TRANSFER}\right) \tag{3.3}$$
$$T_{HCOMP} = T_{HCOMP\_NO} \tag{3.4}$$
$$T_{HCOMM} = T_{HCF} + T_{EXE} + T_{HDATA} \tag{3.5}$$

T_{CONF} : Board configuration time
T_{BOARD} : Board execution time
T_{BCF} : Configuration file transfer time to the board from the host
S_{DATA} : Average size per data transfer b/w board and host
$N_{TRANSFER}$: Average number of data transfers per execution
T_{HOMP_NO} : Computation time in hosts that is non-overlapping with computation in board

T_{HCF} : Configuration file transfer time to host
T_{EXE} : Host execution file transfer time to host
T_{HDATA} : IO data transfer time to host

In general, a portion of the task executes on the GPP machine which hosts the FPGA resource and controls the FPGA portion of the task. We call applications with such tasks dual-paradigm applications, one paradigm being computing with the GPP and the other the FPGA-based RC. The overall execution time of a dual-paradigm application's task can be broadly divided into computation time and communication time corresponding to the host machine and the RC board. Communication time corresponding to the host includes the time to move the host executable, configuration file for the FGPA and I/O data to the host from their respective source (remote) locations. Computation time corresponding to the host involves modeling the host and has been extensively researched by many conventional computing researchers. For our simulations, we are interested only in the RC part of the tasks. Hence we ignore the communication part of the host by assuming that the executables and I/O data are available locally in the host and the computation part of the host by assuming that the host part of execution overlaps with the RC (hence hidden).

The communication time corresponding to the RC board includes the time to move the configuration file and I/O data from the host to the RC board. The computation time corresponding to the RC board includes the time to configure the FPGA and the actual execution time of the task. The communication time (corresponding to both the host and the RC board) involved in the overall execution time is critical only when the data transfer time is very large. Such cases are common in computational grids or other such distributed computing environments over long-haul networks. Our assumption to neglect the communication time

corresponding to the host is made because we are only interested in systems that are not distributed over wide-area networks in this research.

3.3.3 Scheduling Heuristics

In general, scheduling decisions are affected by the relations between the tasks scheduled, heterogeneity of the resources on which the tasks are scheduled, and the performance model that is used to predict the performance of the tasks on the available resources. The quality of scheduler is greatly impacted by the performance model. We discussed the performance model developed for our simulations previously. Given a set of independent tasks and a set of available resources, our goal is to minimize the overall execution time of the entire task set. Based on the model, we can predict the time it takes for each task to execute on each resource. Assuming there are N tasks to be scheduled and there are M machines available, an $N \times M$ matrix can be created with each element in the matrix representing the performance of a task on the corresponding machine in terms of a certain metric. The metric could be anything required by the user such as execution time, memory used, etc. However, it is not always possible to assign the best machine to each task due to resource contention. So we use heuristics to identify the machine to which each task should be assigned.

In this dissertation, we compare the performance of six such heuristics for scheduling of tasks [42] on parallel RC systems. The heuristics compared in this work include opportunistic load balancing (OLB), minimum execution time (MET), minimum completion time (MCT), switching algorithm (SA), MIN-MIN, and MAX-MIN. These heuristics have been studied widely before for general-purpose systems but not RC systems. The following is a brief description of each heuristic.

- **MCT**: Assign each task to the machine with minimum completion time (machine available time + estimated time of computation).

- **MET**: Only consider the expected execution time of each task on the machines and select the machine that provides the minimum execution time for the task.

- **SA**: Switch between MCT and MET based on load imbalance (if load imbalance increases above a threshold, revert back to MCT from MET). The ratio of completion time when scheduled using MCT to that when scheduled using MET is used to switch between the two heuristics.

- **OLB**: Assign each task to the next available machine without considering the expected execution time on that machine. The performance of the machine for the task being scheduled is not considered.

- **MIN-MIN**: First, select a "best" MCT machine for each task. Second, from all tasks, schedule the one with minimum completion time (send a task to the machine which is available earliest and executes the task fastest).

- **MAX-MIN**: Same first step as MIN-MIN but schedule the task with maximum completion time (tasks with long completion time are scheduled first on the best available machines and executed in parallel with other tasks, hence better load-balancing).

The first four heuristics (i.e., MCT, MET, SA, and OLB) are categorized as inline scheduling heuristics because they are used for scheduling individual tasks as they arrive. The last two heuristics (i.e., MIN-MIN and MAX-MIN) are categorized as batch scheduling as they are used to schedule multiple tasks simultaneously. The batch heuristics are commonly used to schedule massively parallel applications with many parallel tasks.

3.4 Simulative Analysis

The characteristics of the applications used in our simulations are given in Table 3-1 based on the performance evaluation performed by Vetter et al.[35]. The tests in [35] were run on an IBM SP system, located at Lawrence Livermore National Laboratory and composed of 68 IBM RS/6000 NightHawk-2 16-way SMP nodes using 375 MHz IBM 64-bit POWER3-II CPUs [43]. The POWER3 provides unit-stride hardware prefetching and hence applications with higher memory locality benefit more in this architecture. The computational intensity gives the ratio of operations to number of memory accesses (loads and stores).

Based on our study of the characteristics of the RC systems and the speedups that they provide for applications in general, we have summarized the characteristics of the RC systems used in our simulations in Table 3-2. It should be noted that the base speedup given in the table is relative and not absolute speedup. For example, if a task speedup is 1.3 times on the RC1000 compared to a GPP machine, then the speedup for the same task is 1.8 times on the Tarari. The speedup numbers are merely representative and may vary widely depending on the specific algorithm but these values represent a form of general behavior for each system.

Table 3-1. Characteristics of applications

Application	Execution Time (sec)	Computational Intensity	Memory Locality (Spatial)
sPPM	19.0	1.45	VERY HIGH
SMG2000	4.0	0.08	LOW
SPHOT	14.0	1.70	HIGH
Sweep3D	3.5	0.75	MEDIUM
IRS	135.0	0.45	VERY HIGH
UMT	350.0	1.50	HIGH
Aztec	14.0	0.45	VERY HIGH

Table 3-2. Characteristics of RC systems

Board/System	Estimated Base Speedup	Configfile Size (KB)	Bus Speed
RC1000	1.3	1200	PCI(64,66)
TARARI	1.8	500	PCI(64,66)
NALLATECH	2.5	2600	PCI(64,33)
SGI	2.8	2733	NumaLink
CRAY	3.0	2377	RapidArray

We know the performance of the tasks on a GPP from Table 3-1. For the performance model that we use for task scheduling in our simulations, we require the execution time (predicted) of the tasks on the RC systems. We calculate this execution time as

$$RCExecutionTime = \frac{BaseExecutionTime \times MemoryLocality}{BaseSpeedup \times ComputationalIntensity}$$ where *BaseExecutionTime*, *MemoryLocality* and *ComputationalIntensity* are provided in Table 3-1 while *BaseSpeedup* is

provided in Table 3-2. We assume that the speedup of a task is higher in an RC system when the computational intensity is higher. If the memory locality of a task is higher, it implies that POWER3's unit stride hardware prefetching has benefited the task. The task will not be able to take advantage of this facility in RC system and hence the speedup will decrease. The numerical values used for *MemoryLocality* in the calculation of *RCExecutionTime* are as follows: 0.99 for VERY HIGH, 0.94 for HIGH, 0.87 for MEDIUM, and 0.80 for LOW.

3.4.1 Simulation Setup

Scheduling is mainly influenced by the execution time of the tasks on the specific RC resource. Hence the other parameters in our performance model are kept constant. Configuration file management is a research area in itself that tries to identify the best ways to store a configuration file, location of storage, replication of heavily accessed files, etc. to optimally retrieve files and configure the FPGAs [44]. Analyzing these and many such tradeoffs in detail would be good research for the future. In this research, we restrict our scope to the analysis of the scheduling heuristics alone. We assume that the configuration files specific to any application task are available on the machine hosting the RC resource. Only the time to move the configuration file from the host machine to the FPGA and configure the FPGA is considered in our simulations.

The following describes the four different simulation setups investigated:

- **Homogeneous tasks on homogeneous machines**: All the machines in the cluster are the same. The tasks that arrive for scheduling are the same.
- **Heterogeneous tasks on homogeneous machines**: All the machines in the cluster are the same. The tasks that arrive are a mix of the 7 tasks listed earlier. To maintain complete heterogeneity in the simulation the tasks are chosen in a round-robin fashion.
- **Homogeneous tasks on heterogeneous machines**: The cluster is composed of a mix of the RC resources listed. In a heterogeneous setup with 12 machines in the system, there are 3 RC1000s and Tararis each, and 2 Nallatechs, SGIs and Crays each. The tasks that arrive to be scheduled are the same.

- **Heterogeneous tasks on heterogeneous machines**: The cluster is composed of a mix of the RC resources as described in the previous setup. The tasks that arrive are also a mix of the seven tasks listed above.

3.4.2 Simulation Results on Large-Scale Systems

In this section, we study the simulation results on large-scale systems typically found in HPC systems on ground. The task arrival to the scheduler is modeled as a Poisson distribution. The total time for which the tasks arrive is fixed at 5000 seconds. The mean value of the Poisson distribution is fixed at 5 sec, 10 sec, and 20 sec to simulate the arrival of 1000 tasks, 500 tasks and 250 tasks, respectively. The number of machines in the system is fixed at 12 for all simulations. The average makespan (i.e., time taken to complete all the tasks that arrive) and average sharing penalty (i.e., difference between the completion time for a task in the present simulation setup and when the task is executed alone without any other task in the system) are measured. However, only makespan is shown in this dissertation. The results are an average of five trials.

Figure 3-2 shows the performance of inline scheduling heuristics when homogeneous tasks (all SPPM in this case) are scheduled on a homogeneous system. There is no difference in the makespan of the tasks for MCT and OLB as long as the tasks and machines are homogeneous. Since the tasks execute very fast, the makespan is driven only by the arrival rate of the tasks. Since there are 12 machines in the system, it will take at least 60 seconds on average, with the fastest arrival rate (mean of 5 sec), for a machine to be scheduled with the next task. But the SPPM task does not execute for 60 sec on any of the machines. Hence the difference in speedup between the machines does not affect the makespan of the tasks across the machines for MCT and OLB. However, for the MET heuristic, all the tasks can potentially be scheduled on the first machine since MET schedules a task on a machine that provides the least execution time for the task. With a homogeneous system, each machine provides the same execution time and hence

the first machine is always picked, hence the large performance difference between MET, and MCT and OLB. As the machine's speed increases, the difference between MET, and MCT and OLB decreases.

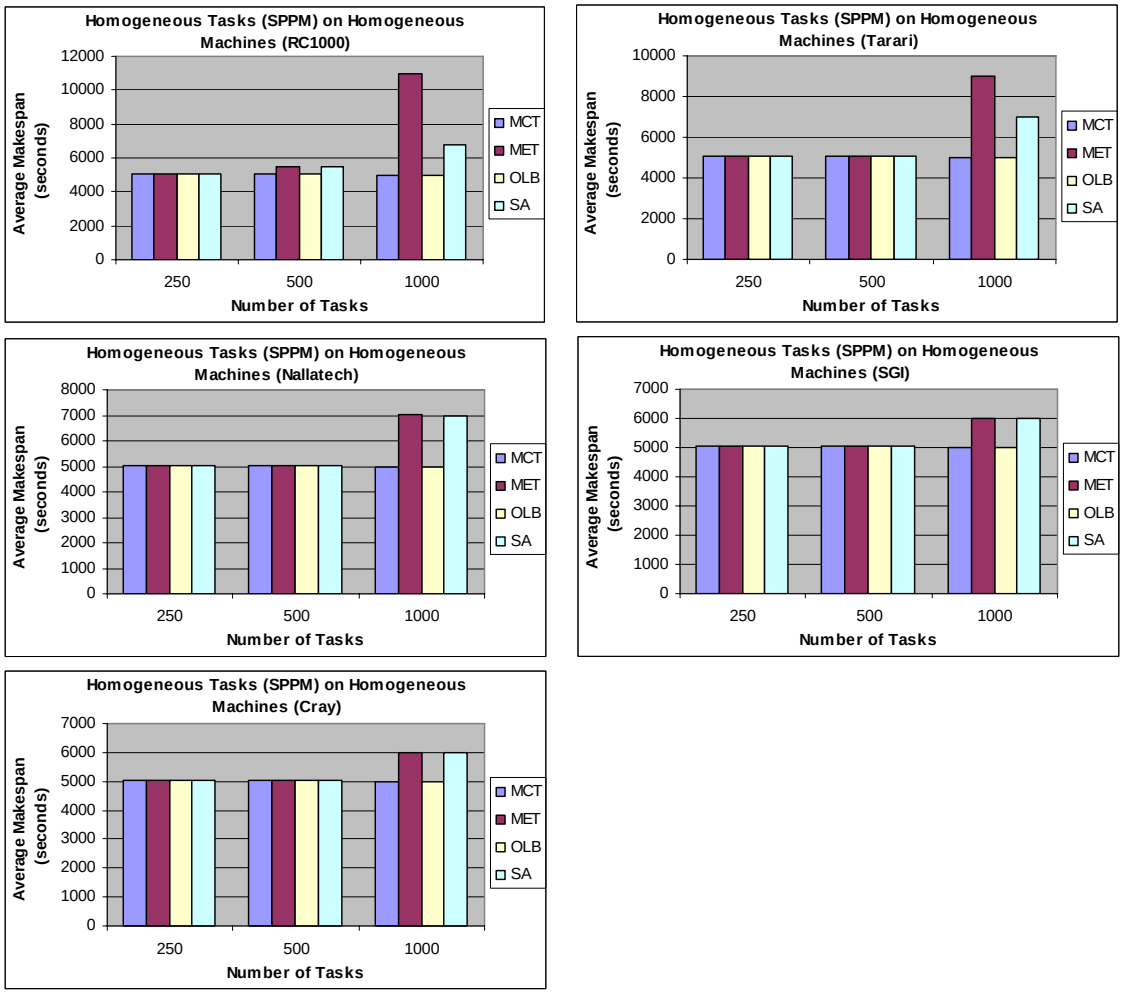


Figure 3-2. Scheduling of SPPM on various RC systems

Figure 3-3 shows the performance of inline scheduling heuristics for the same setup as Figure 3-2 but for UMT, which has a longer execution time compared to SPPM. Both IRS and UMT have long execution times but we have shown only the results for UMT here due to space restrictions. It can be seen from Figure 3-3 that for tasks with longer execution times, the average makespan is driven by the actual execution times and not by the arrival rate of tasks.

Makespan increases with the number of tasks and decreases with faster machines in the system. Again, as the tasks and machines are homogeneous, MCT and OLB perform the same. MET performs poorly compared to MCT and OLB because, with MET, there is a possibility of only a few machines that perform well being loaded with tasks. With MCT and OLB, the tasks are well distributed among the machines in general.

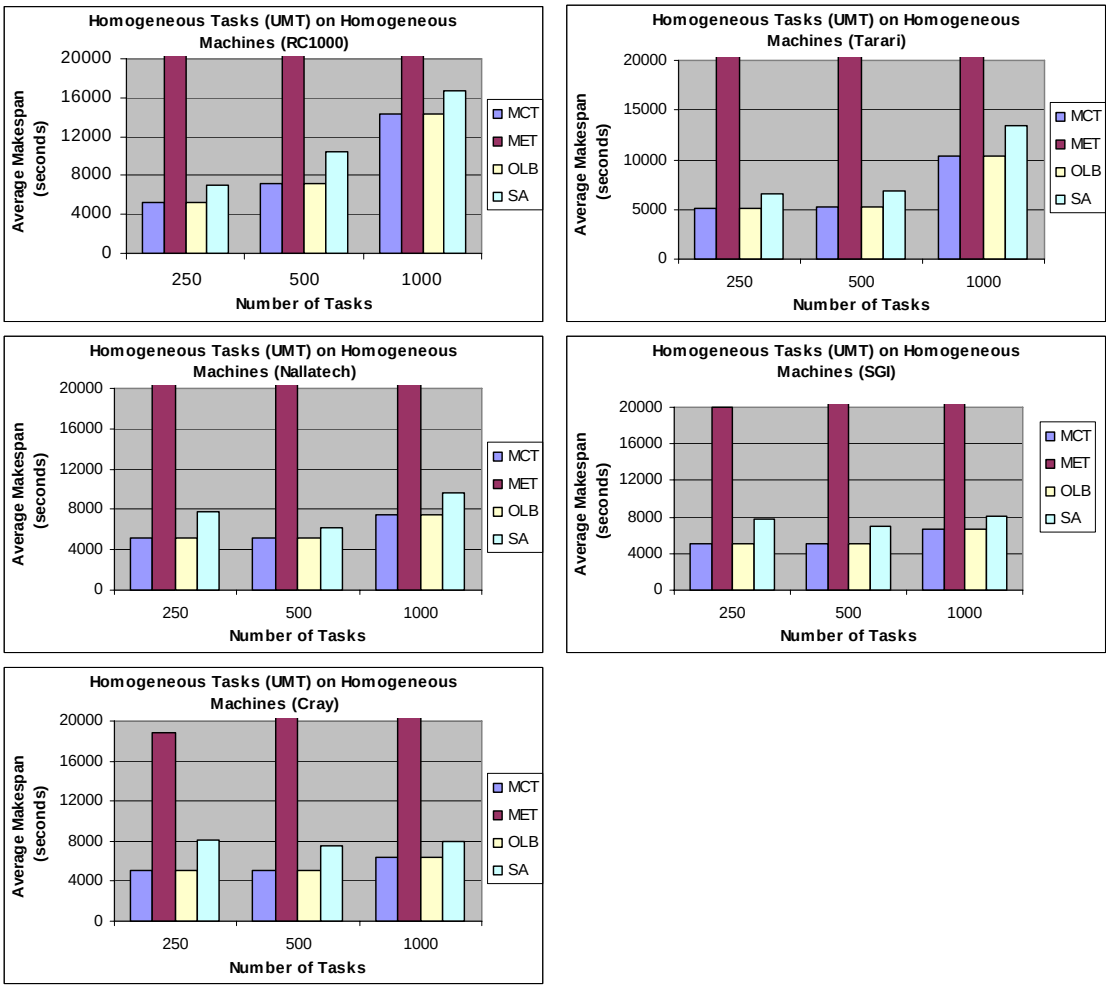


Figure 3-3. Scheduling of UMT on various RC systems

Figure 3-4 shows the performance of inline scheduling heuristics when heterogeneous tasks are scheduled on a system with homogeneous machines. Since the machines are homogeneous, MCT and OLB perform the same. The makespan for MCT and OLB does not

differ between machines for the same reason previously stated (i.e., not enough load on the machines).

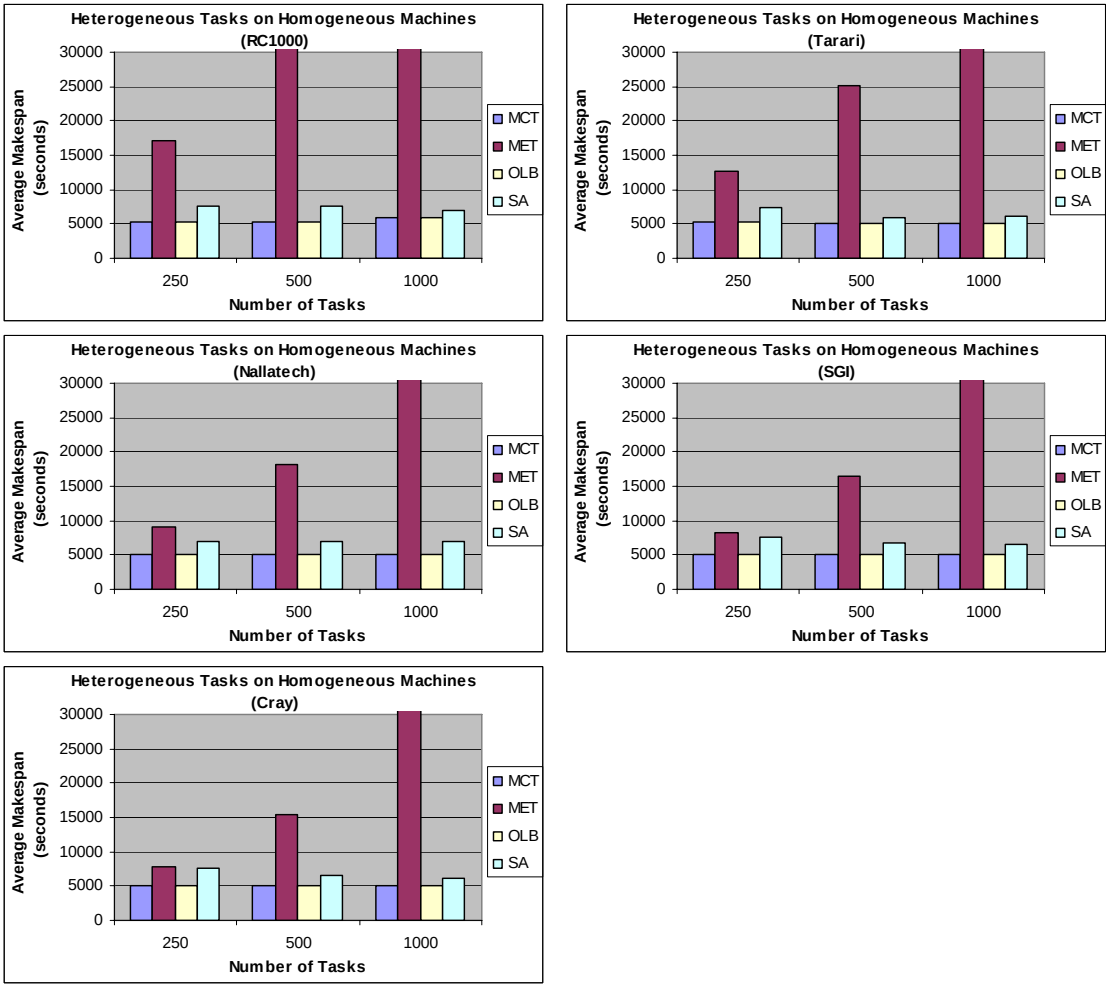


Figure 3-4. Scheduling of heterogeneous tasks on various RC systems

Figure 3-5 shows the performance of inline scheduling heuristics when homogeneous tasks are scheduled on a system with heterogeneous machines. There is no difference in performance between MCT and OLB for tasks that do not have long execution times. However, when the machines are highly loaded, as in the case of 1000 tasks of UMT and IRS, MCT performs slightly better than OLB. Thus we can say that for the difference in performance between MCT and OLB to be realized, the machine workload must be high. In our opinion, such heavy loading

of machines is difficult to realize in reality as it would require a large number of long-running tasks.

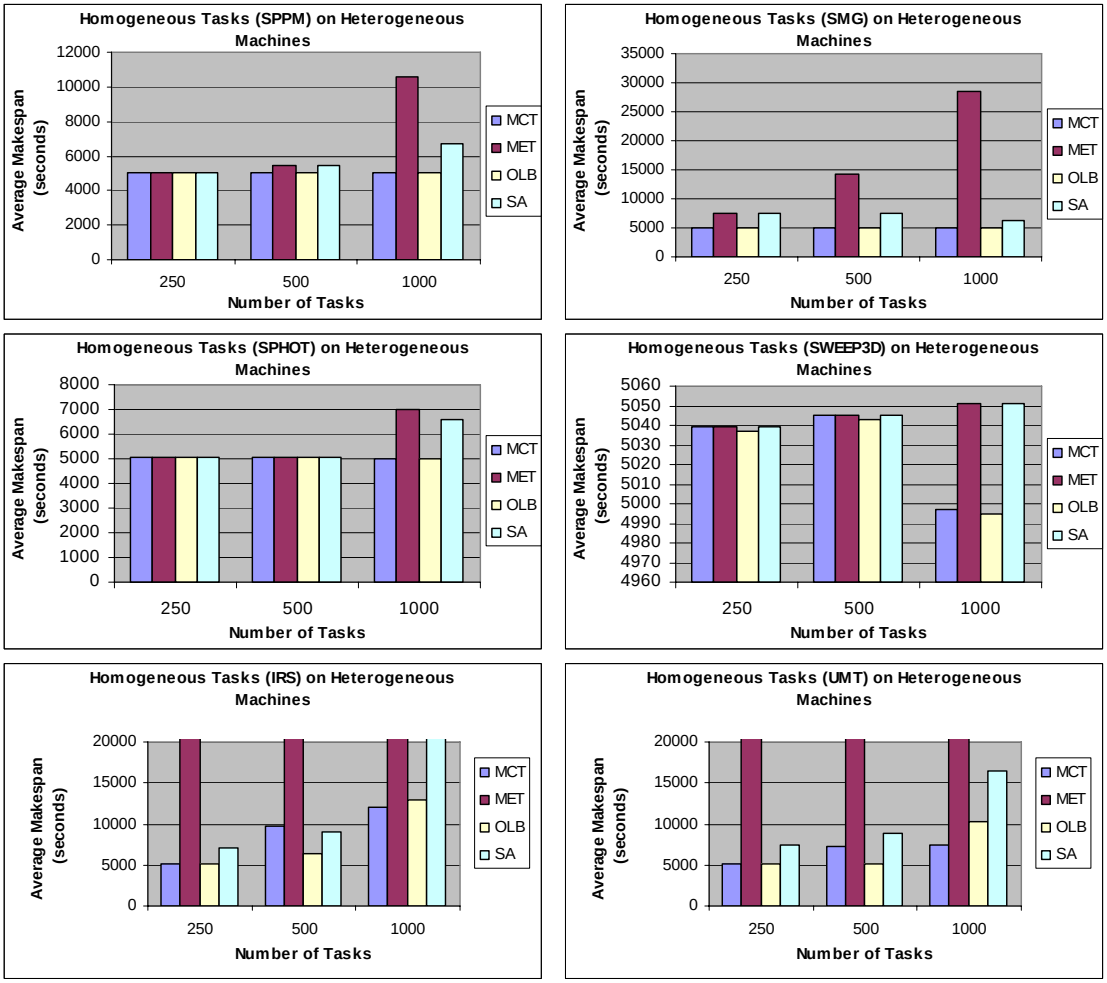


Figure 3-5. Scheduling of various homogeneous tasks on a system with heterogeneous RC machines

Figure 3-6 shows the performance of inline scheduling heuristics when heterogeneous tasks are scheduled on a system with heterogeneous machines. There is no difference in performance between MCT and OLB. In order to find a performance difference between MCT and OLB, we also ran simulations for a system with 6 machines. Even for such a setup (results not shown in figure), the system load was not high enough to widely differentiate the

performance between MCT and OLB (e.g. the makespan for 1000 tasks was 7186 sec for MCT and 7301 sec for OLB).

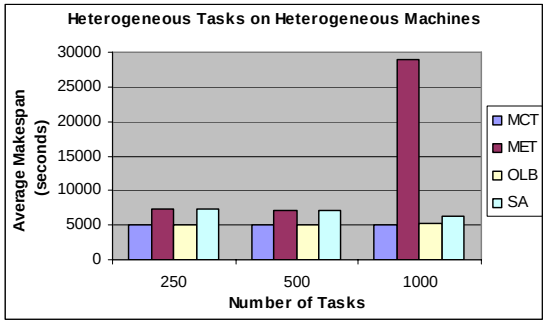


Figure 3-6. Scheduling of heterogeneous tasks on a system with heterogeneous RC machines

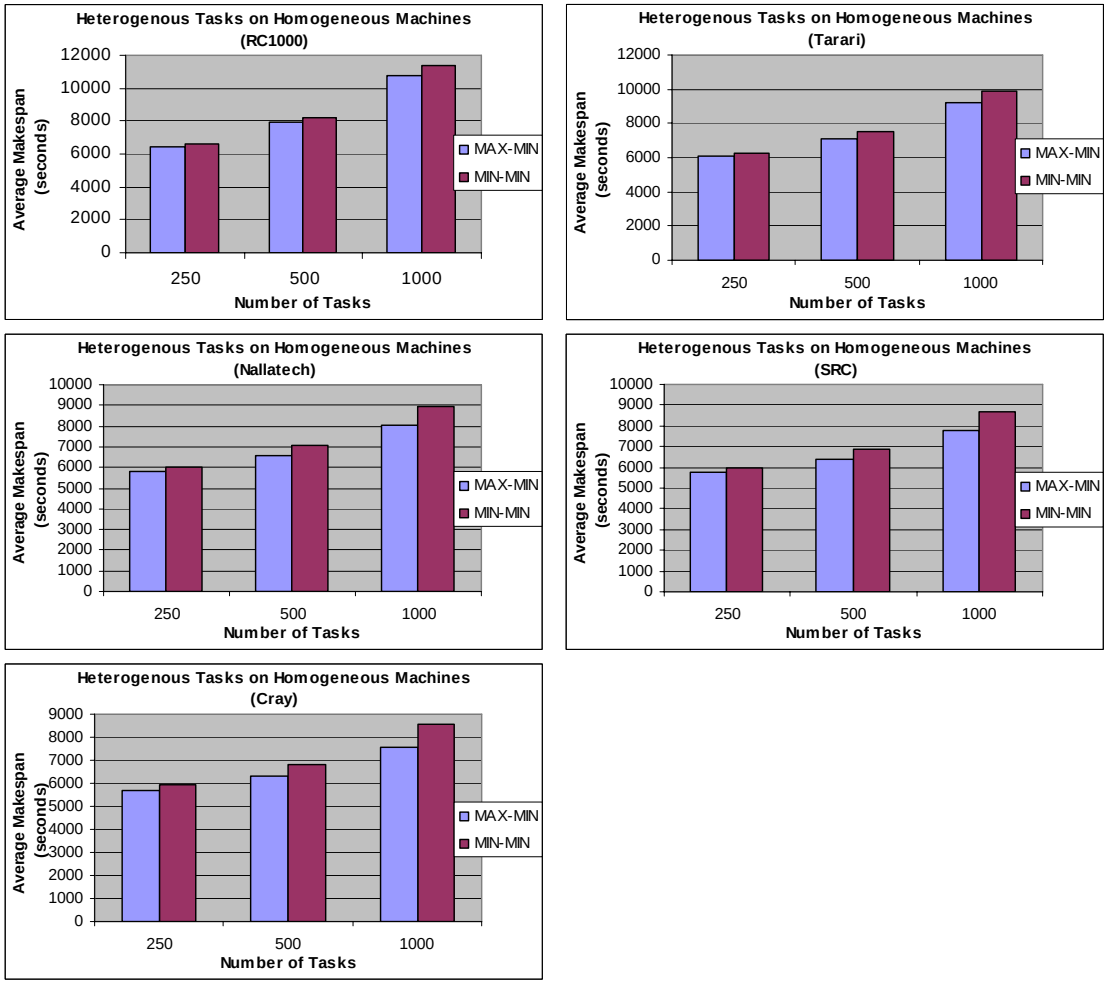


Figure 3-7. Batch scheduling of various heterogeneous tasks on a system with homogeneous RC machines

The results shown until now were for inline scheduling heuristics. Figure 3-7 shows the performance of batch scheduling heuristics for heterogeneous tasks on a system with homogeneous machines. The results for setups when homogeneous tasks are scheduled are not shown because MAX-MIN and MIN-MIN heuristics perform the same as long as the tasks are homogeneous. The results show that MAX-MIN performs better than MIN-MIN in terms of the average makespan. MAX-MIN also improves with the number of tasks being scheduled.

For heterogeneous tasks on a system with heterogeneous RC machines (shown in Figure 3-8), both batch scheduling heuristics show the same behavior seen in Figure 3-7. MAX-MIN performs better than MIN-MIN in all trials.

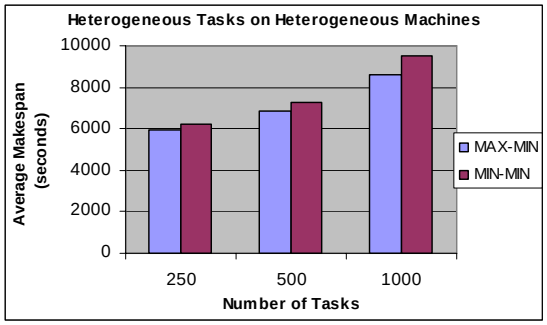


Figure 3-8. Batch scheduling of heterogeneous tasks on a system with heterogeneous RC machines

3.4.3 Simulation Results on Small-scale Systems

In the previous section, we studied the performance of the heuristics on relatively large systems executing in the order of several hundred tasks. However, the space systems might be smaller in terms of the size of the system and the number of tasks executing on them as well. Hence, performance of the heuristics is briefly studied for small-scale systems in this section to compare with the results presented for large-scale systems in the previous section.

The task arrival to the scheduler is again modeled as a Poisson distribution. The total time for which the tasks arrive is fixed at 200 seconds. The mean value of the Poisson distribution is

fixed at 5 sec, 10 sec, and 20 sec to simulate the arrival of 40 tasks, 20 tasks and 10 tasks, respectively. The number of machines in the system is fixed at 3 for all simulations including a RC1000, a Tarari and a SRC. The average makespan (i.e., time taken to complete all the tasks that arrive) is measured for the same simulation setups described in the previous section namely homogeneous tasks on homogeneous machines, heterogeneous tasks on homogeneous machines, homogeneous tasks on heterogeneous machines and heterogeneous tasks on heterogeneous machines.

Figure 3-9 shows the performance of inline scheduling heuristics when homogeneous tasks are scheduled on a homogeneous system. The performance of the heuristics in the small-scale systems was similar to that in large-scale systems and hence only two cases namely SPPM on RC1000 and UMT on SGI are shown here. There is no difference in the makespan of the tasks for MCT and OLB as long as the tasks and machines are homogeneous. Since the tasks execute very fast, the makespan is driven only by the arrival rate of the tasks for the case of SPPM. However, for UMT with longer execution times, makespan increases with the number of tasks and decreases with faster machines in the system.

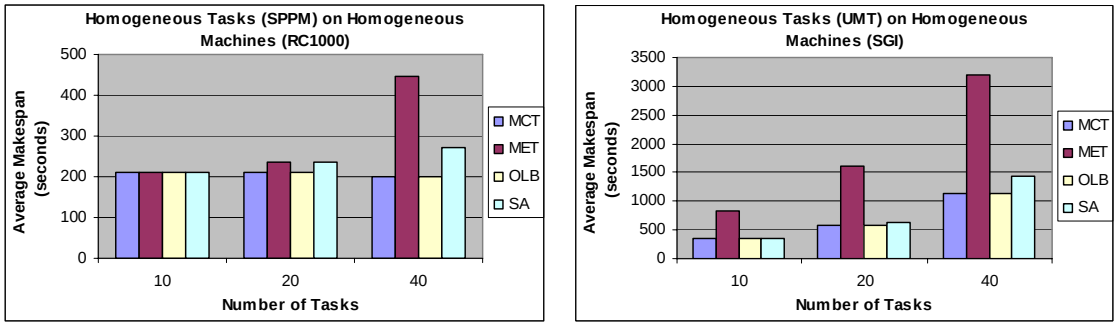


Figure 3-9. Scheduling of homogeneous tasks on homogeneous systems

Figure 3-10 shows the performance of inline scheduling heuristics when heterogeneous tasks are scheduled on a system with homogeneous machines. Since the machines are homogeneous, MCT and OLB perform the same. The makespan for MCT and OLB does not

differ much between machines for the same reason previously stated (i.e., not enough load on the machines).

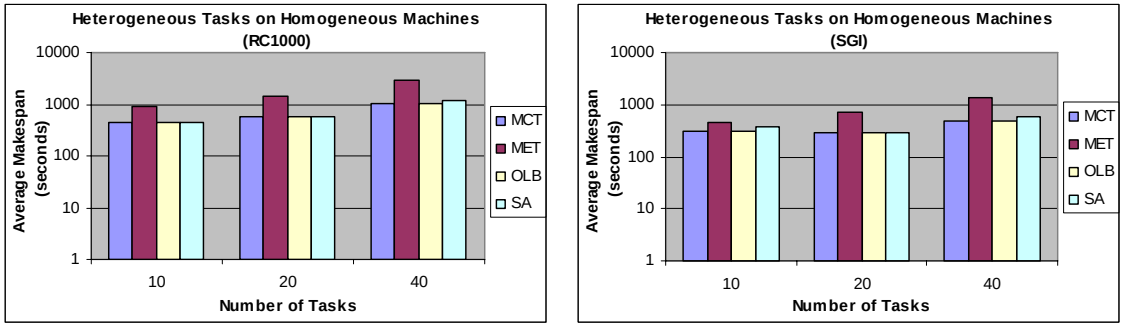


Figure 3-10. Scheduling of heterogeneous tasks on various RC systems

Figure 3-11 shows the performance of inline scheduling heuristics when homogeneous tasks namely SMG and IRS are scheduled on a system with heterogeneous machines. The trend is similar to that in large-scale systems. There is no difference in performance between MCT and OLB since the machines are not overloaded with tasks.

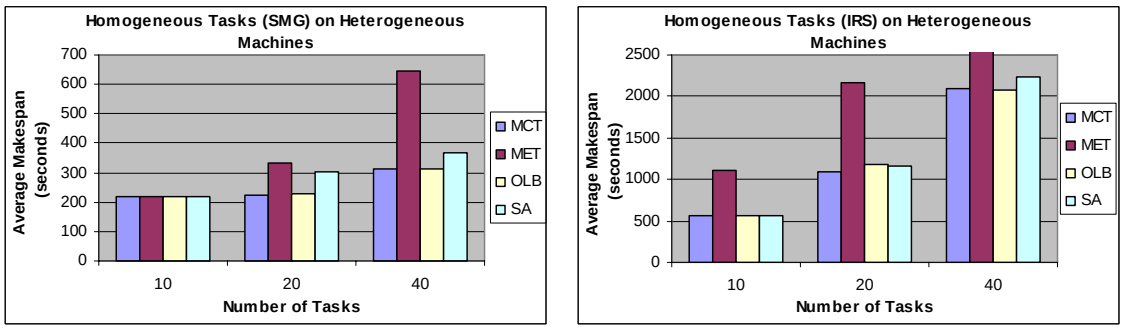


Figure 3-11. Scheduling of homogeneous tasks on heterogeneous systems

The performance of the heuristics was also studied in a heterogeneous system when heterogeneous tasks were scheduled and the trend was similar to that seen in large-scale systems and hence those results are not shown here. In summary, the heuristics perform similarly in both large-scale and small-scale systems.

3.5 Conclusions and Future Research

Recently, systems augmented with FPGAs offering a fusion of traditional parallel and distributed machines with customizable and dynamically reconfigurable hardware have emerged as a cost-effective alternative to traditional systems. However, providing a robust runtime environment for such systems to which HPC users have become accustomed has been fraught with numerous challenges. Dynamic scheduling of HPC applications in such parallel reconfigurable computing (RC) environments is one such challenge and has not been sufficiently studied to our knowledge.

In this phase of research, we to improve the overall execution time of applications by analyzing methodologies for effective task scheduling. Reducing the overall execution time of applications exposes the applications to lesser number of faults in the system. We analyzed the performance of several scheduling heuristics that can be used to schedule application tasks on HPC systems. Typical HPC applications and FPGA/RC resources were used as representative cases for our simulations. A performance model was also developed to predict the overall execution time of tasks on RC resources. The model was used by our scheduling heuristics to schedule the tasks. The performance model and the extensive simulative analysis of scheduling heuristics for FPGA-based RC applications and platforms is the first of its kind and is the primary research contribution of this phase.

Among the inline scheduling heuristics, 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. Among the batch scheduling heuristics, MAX-MIN outperforms MIN-MIN in terms of average makespan. Even

with a small 12-machine system, the load on the system was poor. With larger systems, the impact of poor load will further diminish the performance difference between the scheduling heuristics. In a smaller system with 3 machines, typical of what would be available in space systems, the performance trend was similar to that of the 12-machine system. Also, the space systems are not going to be continuously heavily loaded with tasks and hence a simple scheduling heuristic such as the OLB might suffice for scheduling tasks.

The intent of this phase of research is to analyze several scheduling heuristics and compare their performance to identify the suitable candidates for use in space-based HPC systems. We have shown the performance results for average makespan. Based on the performance metric that is critical for a given environment, the corresponding scheduling heuristics can be used to schedule tasks. In future, the knowledge gained so far to suggest suitable scheduling heuristics for an experimental scheduler can be applied in the job management service intended for the space-based HPC system being developed at the HCS Research Laboratory at University of Florida. Another interesting future research would be to use typical parallel RC applications to study the validity of the simulative analysis by comparing with the effectiveness of task scheduling (in terms of makespan) of the job management service.

CHAPTER 4
A FAULT-TOLERANT MESSAGE PASSING INTERFACE (PHASE III)

Fault tolerance is a critical factor for HPC systems in space to meet the emerging high-availability and reliability requirements. Recovery from failure needs to be faster and automatic while the impact of failures on the system as a whole should be minimal. In the previous phases of this dissertation, we addressed the issue of minimizing the impact of failures through indirect approaches, mechanisms that do not address direct recovery from faults. The indirect approaches certainly avoid computation loss but in order to enable applications to meet high-availability and high-reliability requirements we need to consider other options. Some of the options include: incorporating fault-tolerant features directly into the applications, developing specialized hardware that is fault-tolerant, making use of and enhancing the fault-tolerant features of the operating system, and developing application-independent middleware that would provide fault-tolerant capabilities. Among these options, developing application-independent middleware has the minimal intrusion in the system and can support any general application including legacy applications that fall into the umbrella of the corresponding middleware model.

In this phase of the dissertation, we investigate, design, develop and evaluate a fault-tolerant, application-independent middleware for embedded cluster computing known as FEMPI (Fault-tolerant Embedded Message Passing Interface). We also present performance results with FEMPI on a traditional PC-based cluster and on a COTS-based, embedded cluster system prototype for satellite payload processing in development at Honeywell Inc. and the University of Florida for the Space Technology 8 (ST-8) mission of NASA's New Millennium Program. We take a direct approach to provide fault-tolerance and improve the availability of the HPC system in space. FEMPI is a lightweight fault-tolerant variant of the Message Passing Interface (MPI) standard.

4.1 Introduction

Because of its widespread usage, MPI [45] has emerged as the de-facto standard for development and execution of high-performance parallel applications. By its nature as a communication library facilitating user-level communication among a group of processes, the MPI library needs to maintain global awareness of the processes that collectively constitute a parallel application. An MPI library consequently emerges as a logical and suitable place to incorporate selected fault-tolerant features in order to enable legacy and new applications to meet the emerging high-availability and reliability requirements of HPC systems in space. However, fault tolerance is absent in both the MPI-1 and MPI-2 standards. To the best of our knowledge, no satisfactory products or research results offer an effective path to providing scalable computing applications for cluster systems and specifically resource-constrained embedded systems such as in space with effective fault-tolerance. However, there have been a few efforts to develop lightweight MPI [46] and more specifically for embedded systems [47, 48] but without fault-tolerance. In this dissertation, we present the design and analyze the characteristics of FEMPI, a new lightweight, fault-tolerant message passing middleware for clusters of embedded systems. The scope of this paper is focused upon a presentation of the design of FEMPI and performance results with it on a COTS-based, embedded cluster system prototype for space. Considering the small scale of the embedded cluster, we also provide results from experiments on an Intel Xeon cluster to show scalability of FEMPI. The experiments on the Xeon cluster highlight the compatibility of FEMPI across platforms and also permit performance comparisons with conventional MPI middleware. Finally, we also study the performance of FEMPI in comparison to conventional MPI variants with a real application.

Performance and fault-tolerance are in general competing goals in HPC. Achieving a sufficient level of fault-tolerance for a large set of practical environments with minimal or no

impact on performance is a significant challenge. Providing fault-tolerant services in software inevitably adds extra processing overhead and increases system resource usage. In this phase of the research, we focus on developing a message-passing middleware architecture that can successfully address both fault tolerance and performance and if conflicting then fault tolerance is given priority. We address fault-tolerance issues in both the MPI-1 and MPI-2 standards, with attention to key application classes, fault-free overhead, and recovery strategies. When developed, the architecture would provide key new capabilities to parallel programs, programmers, and cluster systems, including the enhancement of existing commercial applications based on MPI. The optimizations targeted at the popular recovery mechanisms, for key classes of applications, can be applied to any middleware and hence would result in improving the performance of applications in general.

The rest of this chapter is organized as follows. Section 4.2 provides background on several existing fault-tolerant MPI implementations for traditional general-purpose HPC systems. Section 4.3 discusses the architecture and design of FEMPI. Failure-free performance results are presented in Section 4.4 while failure recovery performance is analyzed in Section 4.5. Performance of FEMPI when used in a real application is discussed in Section 4.6. Section 4.7 concludes the paper and summarizes insight and directions for future research. The final section provides the summary and research scope.

4.2 Background and Related Research

In this section, we provide an overview of MPI and its inherent limits relative to fault tolerance. Also included is a brief survey and summary of existing tools with features for bringing fault tolerance to MPI, albeit primarily targeting conventional, resource-plentiful HPC systems and their applications instead of embedded, mission-critical systems that are the emphasis of our work.

4.2.1 Limitations of MPI Standard

The MPI forum released the first MPI standard, MPI-1, in 1995, with drafts provided over the previous 18 months. The main goals of the standard in this release were high performance and portability. Achieving reliability typically includes utilization of additional resources and methodologies. This additional utilization conflicts with the main goal of high performance and supports the MPI Forum's decision for limited reliability measures. Emphasis on high performance thus led to a static process model with limited error handling. The success of an MPI application is guaranteed only when all constituent processes finish successfully. "Failure" or "crash" of one or more processes leads to a default application termination procedure which is in fact required standard behavior. Subsequently, current designs/implementations of MPI suffer inadequacies in various aspects to providing reliability.

Fault Model: MPI assumes a reliable communication layer. The standard does not provide methods to deal with node failures, process failures, and lost messages. MPI also limits faults recognized in the system to incorrect parameters in function calls and resource errors. This coverage of faults is incomplete and insufficient for high-scale parallel systems and mission-critical systems that are affected by transient faults such as SEUs.

Fault Detection: Fault detection is not defined by MPI. The default mode of operation of MPI treats all errors as fatal and terminates the entire parallel job. MPI provides for limited fault notification in the form of return codes from the MPI functions. However, critical faults, such as process crashes, may preempt functions from returning these return codes to the caller. The ability to continue execution after the return of certain error codes is completely ambiguous, and left as implementation-specific properties and hence not portable.

Fault Recovery: MPI provides users with functions to register error-handling callback functions. These callback functions are invoked by the MPI implementation in the event of an error in MPI

functions. Callback functions are registered on a per communicator (communication context defined in MPI for groups of processes) basis and do not allow per-function based error handlers. Callback functions provide limited capability and flexibility and cannot be invoked in case of process crashes and hangs.

The MPI Forum released the MPI-2 standard in 1998, after a multi-year standards process. MPI-2 consists of extensions in the areas of process creation and management, one-sided communications, extended collective operations, and parallel I/O. A significant contribution of MPI-2 is Dynamic Process Management (DPM), which allows user programs to create and terminate additional groups of processes on demand. DPM may be used to compensate for the loss of a process while MPI I/O can be used for check pointing the state of the applications. However, the lack of failure detection precludes the potential for added reliability.

4.2.2 Fault-tolerant MPI Implementations for Traditional Clusters

Several research efforts have been undertaken to make MPI more reliable for traditional HPC systems. This section introduces some of these efforts and analyzes their approaches in providing a reliable MPI middleware.

CoCheck [49] from the University of Germany, Munich, is a checkpointing environment for parallel applications and is one of the earliest efforts to make MPI more reliable. CoCheck was primarily targeted for process migration, load balancing, and stalling long-running applications for later resumption. CoCheck extends the single process checkpoint mechanisms to a distributed message-passing application. Unlike most checkpointing middleware, CoCheck sits on top of the message passing system and provides checkpointing transparent to the application. CoCheck incurs a large overhead by checkpointing entire process state and requires a centralized coordinator. Recovery of a dead process is achieved by a recovery function run at

the user level. The status of inconsistent internal data structures in message-passing middleware is not addressed. Thus, CoCheck provides coarse reliability measures for the MPI.

MPICH-V [50] from University of South Paris, France, is an MPI environment that is based upon uncoordinated checkpointing/rollback and distributed message logging. The architecture as shown in Figure 4-1 relies on channel memories and checkpoint servers. It is assumed that on a failure, a node is no more reachable and the computations by the failed node will have no impact on the eventual results. Channel memories are special nodes doing the job of a middleman through which all the communications pass through and hence are logged. The dispatcher is a coordinating node that schedules tasks to computing nodes and coordinates resources as well. On a failure, the coordinator detects the failure and, with the help of the channel memories and checkpointing schedulers, the task is rescheduled on fault-free nodes. Although MPICH-V suffers from single points of failure it is suitable for Master/Worker types of MPI applications.

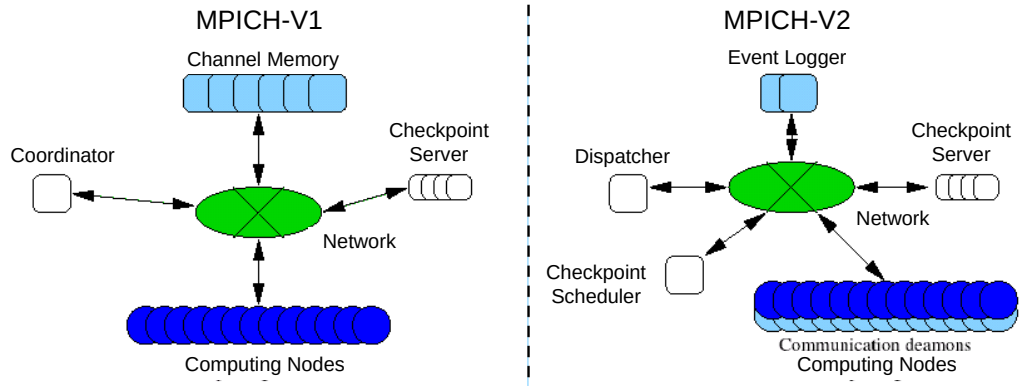


Figure 4-1. MPICH-V architecture (Courtesy: [50])

Starfish [51] from the Technion University, Israel, is an environment for executing dynamic MPI-2 programs. Figure 4-2 illustrates the architecture of Starfish. Every node executes a starfish daemon and many such daemons form a process group using the Ensemble group communication toolkit [52].

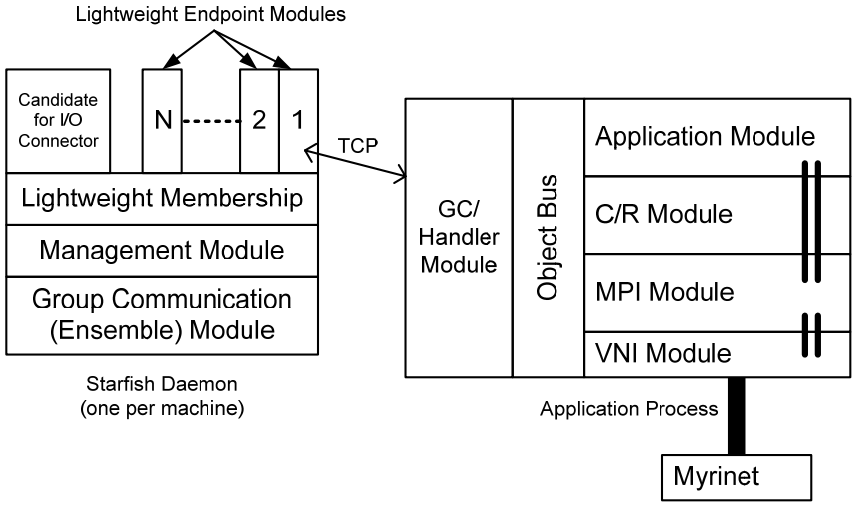


Figure 4-2. Starfish architecture (Courtesy: [51])

The daemons are responsible for interacting with clients, spawning MPI programs and tracking and recovering from failures along with group membership. Starfish uses an event model that requires the processes and components to register to listen on events. The event bus that provides a fast data path supplies messages to reflect cluster changes and process failures. Each application process includes a group communication handler module to interact with the daemon, an application module that includes the user supplied MPI code, a checkpoint/restart module, an MPI module and a virtual network interface (VNI). The architecture allows for any checkpoint/restart protocol implementation. Likewise, the architecture can be ported to any network by providing a thin layer inside the VNI pertaining to the particular network.

Egida [53] from the University of Texas, Austin, is an extensible toolkit to support transparent rollback recovery. Egida as shown in Figure 4-3 is built around a library of objects that implement a set of functionalities that are the core of all log-based rollback recovery. Any arbitrary rollback protocol can be specified and Egida can synthesize an implementation. Egida also allows for the coexistence of multiple implementations. Egida has been ported to MPICH [54], a widely used and free implementation of MPI. Egida shares some of its drawbacks with

CoCheck. Egida checkpoints the state of both processes and messages and may lead to large overheads in some cases.

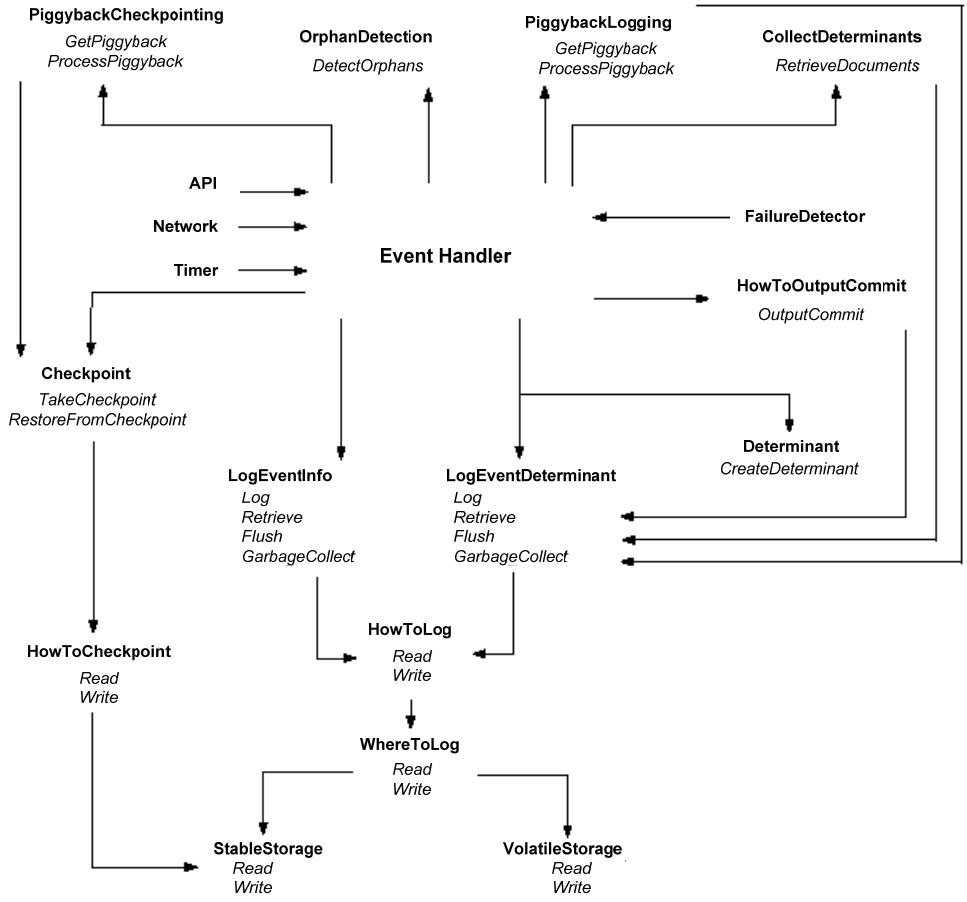


Figure 4-3. Egida architecture (Courtesy: [53])

Implicit FT-MPI [55] from the University of Cyprus takes an approach similar to CoCheck but is a more stripped-down version. Implicit FT-MPI targets only the master/slave. A separate observer process to which the master node sends all the messages is responsible for coordination of all the processes and also records the status of all the processes. The observer is also responsible for creation of new processes on failure of slave nodes and, if the master node fails, the observer takes up the role of the master itself. Implicit FT-MPI is very simple in terms of features available and suffers from single point of failures.

LAM/MPI [56], an implementation of MPI from Indiana University and Ohio Supercomputing Center also has some fault-tolerant features built into it. Special functions such as lamgrow and lamshrink are used for dynamic addition and deletion of hosts. A fail-stop model is assumed and, whenever a node fails, it is detected as dead and the resource manager removes the node from the host lists. All the surviving hosts are notified asynchronously and the MPI library invalidates all the communicators that include the dead node. Pending communication requests are marked as errors. Since attempts to use invalid communicators raise errors, applications can detect these errors and free the invalid communicators and new communicators can be created if necessary.

FT-MPI [57] from University of Tennessee, Knoxville, attempts to provide fault-tolerance in MPI by extending the MPI process states and communicator states from the simple {valid, invalid} as specified by the standard to a larger number of states. A communicator is an important scoping and addressing data structure defined in the MPI standard that defines a communication context, and a set of processes in the context. The range of communicator states specified by FT-MPI helps the application with the ability to decide how to alter the communicator, its state and the behavior of the communication between intermediate states on occurrence of a failure. In case of a failure, FT-MPI returns a handle back to the application. MPI communicator functions specified in the MPI-2 standard are used to shrink, expand or rebuild communicators. FT-MPI provides for graceful degradation of applications but has no support for transparent recovery from faults. The design concept of FEMPI is also largely based upon FT-MPI, although there are significant differences given that FEMPI is designed to target embedded, resource-limited, and mission-critical systems where faults are more commonplace, such as payload processing in space.

4.3 Design of FEMPI

As described in the previous section, several efforts have been made to develop fault-tolerant MPI implementation. However, all the designs described target conventional large-scale HPC systems with sufficient system resources to cover the additional overhead for fault tolerance. More importantly, very few among those designs have been successfully implemented and are mature enough for practical use. Also, the designs are quite heavyweight primarily because fault tolerance is based on extensive message logging and checkpointing. Many of the designs are also based on centralized coordinators that can incur severe overhead and become a bottleneck in the system. An HPC system in space requires a reliable and lightweight design of fault-tolerant MPI. Among those that exist, FT-MPI from University of Tennessee [57] was viewed to be the one with the least overhead and is the most mature design in terms of realization. However, FT-MPI is built atop a metacomputing system called Harness that can be too heavy for embedded systems to handle.

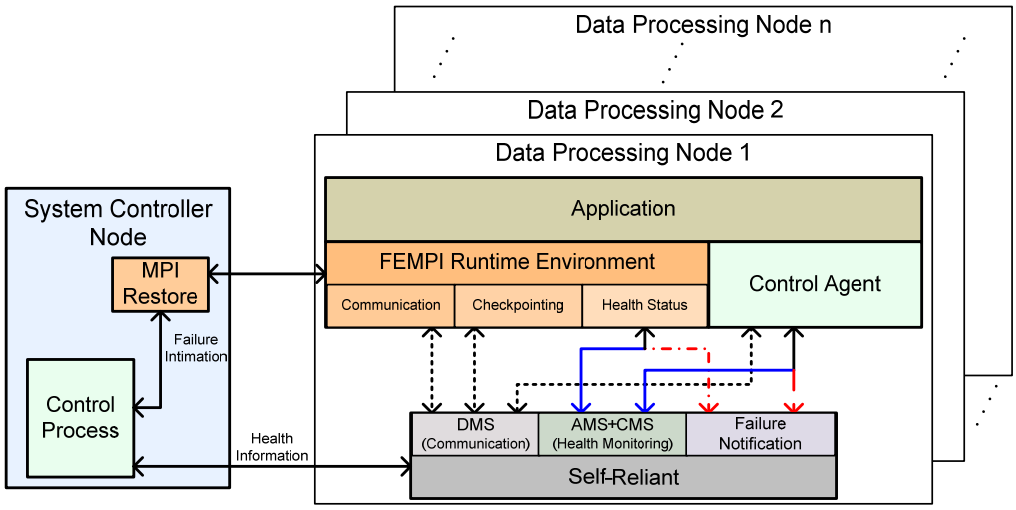
For the design of FEMPI, we try to avoid design and performance pitfalls of existing HPC tools for MPI but leverage useful ideas from these tools. FEMPI is a lightweight design in that it does not depend upon any message logging, specialized checkpointing, centralized coordination or other large middleware systems. Our design of FEMPI resembles FT-MPI. However, the recovery mechanism in FEMPI is different in that it is completely distributed and does not require any reconstruction of communicators. On the other hand, FT-MPI requires the reconstruction of communicators on a failure for which the system enters an election state and a voting-based leader selection is performed. The leader is responsible for distributing the new communicator to all the nodes in the system.

4.3.1 FEMPI Architecture

Fault tolerance is provided through three stages including detection of a fault, notification of the fault, and recovery from the fault. In order to reduce development time and to ensure reliability, FEMPI is built atop a commercial high-availability (HA) middleware called Self-Reliant (SR) from GoAhead Inc. whose services are used to provide detection and notification capabilities. The primary functions of the HA middleware are resource monitoring, fault detection, fault diagnosis, fault recovery, fault reporting, cluster configuration, event logging, and distributed messaging. SR is based on a small, reliable, cross-platform kernel that provides the foundation for all standard services, and its extensions. The kernel also provides a portability layer limiting user dependencies on the underlying operating system and hardware.

SR allows processes to heartbeat through certain fault handlers and hence has the potential to detect the failure of processes and nodes, enabled by the Availability and Cluster Management Services (AMS and CMS) in Figure 1. CMS manages the physical nodes or instances of SR, while AMS manages the logical representation of these and other resources in the availability system model. The fault notification service for application processes is developed as an extension to SR. Application heartbeats are managed locally within each node by the service and only health 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 a controller node. 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. SR also guarantees reliable communication via network interface failover capabilities and in-order delivery of messages between the nodes in the system through its Distributed Messaging Service

(DMS). It should be mentioned here that a lightweight version of SR is used for our prototype with just enough services for cluster management. Moreover, FEMPI only uses minimal services of SR such as failure detection and DMS. Hence, FEMPI with SR would be less stressful on the embedded cluster as opposed to FT-MPI with Harness which is intended for large clusters.



AMS: Availability Management Service CMS: Cluster Management Service

Figure 4-4 Architecture of FEMPI

Figure 4-4 shows the architecture of FEMPI. The application is required to register with the Control Agent in each node, which in turn is responsible for updating the health status of the application in that node to a central Control Process. The Control Process is similar to a system manager, scheduling applications to various data processing nodes. In the ST-8 mission, the Control Process will execute on a radiation-hardened, single-board computer that is also responsible for interacting with the main controller for the entire spacecraft. Although such 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.

With regard to MPI applications, failures can be broadly classified as process failures (individual processes of an MPI application crash) and network failures (communication failure

between two MPI processes). FEMPI ensures reliable communication (reducing the changes of network failures) with all the low-level communication through DMS. A process failure in conventional MPI implementations causes the entire application to crash, whereas FEMPI avoids application-wide crashes when individual processes fail. MPI Restore, a component of FEMPI, resides on the System Controller and communicates with the Control Process to update the status of nodes. On initialization by the application using the *MPI_Init* function call, the FEMPI runtime environment attached to the application process subscribes itself to a *Health Status* channel. The same procedure is performed for all the MPI processes corresponding to the application executing in all the nodes. It is through this channel that updates about the status of nodes are received from MPI Restore. On a failure, MPI Restore notifies all other MPI processes regarding the failure via DMS. The status of senders and receivers (of messages) are checked in FEMPI before communication to avoid trying 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.

With the initial design of FEMPI as described in this dissertation, we focus only on selected baseline functions of the MPI standard. The first version of FEMPI includes 19 baseline functions shown in Table 4-1, of which four are setup, three are point-to-point messaging, five are collective communication, and seven are custom data-definition calls. The function calls were selected based upon profiling of popular and commonly used space science kernels such as Fast Fourier Transform, LU Decomposition, etc. With the ability to develop these common application kernels, the baseline functions would be sufficient to support the desired applications for the ST-8 mission. Provision of just certain baseline functions (along with ability to develop the other functions defined in the MPI standard) in order to support the development of desired

applications is common during the design of a new MPI implementation[46-48, 58]. In our initial version of FEMPI, we also focus only on blocking and synchronous communications, which is the dominant mode in many MPI applications. Blocking and synchronous communications require the corresponding calls (e.g. send and receive) to be posted at both the sender and receiver processes for either process to continue further execution.

Table 4-1. Baseline MPI functions for FEMPI in this phase of research

Function	MPI Call	Type	Purpose
Initialization	*MPI_Init*	Setup	Prepares system for message-passing functions.
Communication Rank	*MPI_Comm_rank*	Setup	Provides a unique node number for each node.
Communication Size	*MPI_Comm_size*	Setup	Provides the number of nodes in the system.
Finalize	*MPI_Finalize*	Setup	Disable communication services.
Send	*MPI_Send*	P2P	Send data to a matching receive.
Receive	*MPI_Recv*	P2P	Receive data from a matching send.
Send-Receive	*MPI_Sendrecv*	P2P	Simultaneous send and receive between two nodes (i.e., send and receive using the same buffer).
Barrier Synchronization	*MPI_Barrier*	Collective	Synchronize all the nodes together.
Broadcast	*MPI_Bcast*	Collective	"Root" node sends same data to all other nodes.
Gather	*MPI_Gather*	Collective	Each node sends a separate block of data to "root " node to provide an all-to-one scheme
Scatter	*MPI_Scatter*	Collective	"Root" node sends a different set of data to each node providing a one-to-allscheme
All-to-All	*MPI_Allgather*	Collective	All nodes share their data with all other nodes in the system.
Datatype	*MPI_Type_**	Custom	Seven functions to define custom data types useful for complicated calculations

4.3.2 Point-to-Point Messaging (Unicast Communication)

The basic communication mechanism of MPI, referred to as 'point-to-point messaging' is the transmittal of data between a pair of processes, one sending and the other receiving. A set of send and receive functions allow the communication of data of a specific datatype with an associated tag. The tag allows selectivity of messages at the receiving end: one can receive on a particular tag, or one can wild-card this quantity, allowing reception of messages with any tag. Message selectivity on the source process of the message is also provided.

DMS, the messaging service of SR, operates by managing distributed virtual multicast groups with publish and subscribe mechanisms over primary and secondary networks. The publish and subscribe mechanisms are used by FEMPI for provisioning MPI point-to-point messaging. On initialization, each FEMPI (application) process is registered to a data channel wherein messages can be published or received. Each process registers with its unique identity (MPI process rank). The message to be transmitted from a process is published on the data channel with additional information to indicate the identity of the target receiver. The identity being unique to each process, DMS filters the message to be relayed only to the target process and is not broadcasted to the other processes.

4.3.3 Collective Communication

A collective operation is executed by having all processes in the group call the communication routine, with matching arguments. Collective communications transmit data among all processes in a group specified by an intracommunicator object. One exception however is the *MPI_Barrier* function that serves to synchronize processes without passing data.

In FEMPI, the publish and subscribe mechanism of DMS is used directly for collective routines such as broadcast or all-to-all communication that involves the transmission of a message from a single node to all other nodes. However other routines that involve more than one process transmitting messages are designed as variations of point-to-point messaging (sequence of multiple point-to-point operations).

4.3.4 Failure Recovery

Presently, two different recovery modes have been developed in FEMPI, namely IGNORE and RECOVER. In the IGNORE mode of recovery, when a FEMPI function is called to communicate with a failed process, the communication is not performed and *MPI_PROC_NULL* (meaning the process location is empty) is returned to the application. Basically, the failed

process is ignored and computation proceeds without any effect of the failure. The application can either re-spawn the process through the control process or proceed with one less process. The MPI communicators are left unchanged. IGNORE mode is useful for applications that can execute with reduced number of processes while the failed process is being recovered, especially if the recovery procedure is of a long duration. With the RECOVER mode of recovery, the failed process has to be re-spawned back to its healthy state either on the same or a different processing node. When a FEMPI function is called to communicate with a failed process, the function call is halted until the process is successfully re-spawned. When the process is restored, the call is completed and the control is returned back to the application. Here again, the MPI communicators are left changed. RECOVER mode is useful for applications that cannot execute with any less number of nodes than when they started execution.

In future, we plan to develop a third mode of recovery, namely REMOVE. With the REMOVE mode of recovery, when a process fails, it is removed from the process list. The MPI communicator is altered (shrunk) to reflect the change that the system has one less process. REMOVE mode would be useful for cases when any process or processes have irrecoverably failed while running applications that cannot proceed with any failed process in the system. However, this mode would require the coordination of all the MPI processes via a MPI control process to consistently update the communicator.

4.3.5 Covering All MPI Function Call Categories

MPI function calls can be broadly classified into four different categories based on the locality of impact of a failure in the system. The categories include point-to-point communication calls with communication between specific processes, collective communication calls with communication between a group of processes, process-specific calls that are local to a single process, and group-specific calls that are collective in nature but do not involve explicit

message passing. The function calls developed for the initial version of FEMPI in this dissertation, also listed in Table 4-1, have samples in all these categories (except for the non-blocking version of point-to-point communication) indicating the ability for FEMPI to be extended to the other function calls specified by the MPI standard.

For point-to-point communication calls, the impact of a process failure only extends to the partner (communication partner) process. In FEMPI, the failure is handled by waiting for the partner process to recover or by returning an error to the calling routine. FEMPI calls that fall in this category include *MPI_Send*, *MPI_Recv* and *MPI_Sendrecv*. The locality of impact of a failure during collective communication calls depend on whether the failure was in a root or non-root process. Several collective routines such as broadcast and gather have a single originating or receiving process. Such processes are called the root. On failure of a non-root process, the impact is only on the root which is the solitary process communicating with non-roots. Hence all other processes can complete successfully. The root deals with the failure similar to point-to-point communication calls. On the other hand, if the root fails, all non-roots are impacted. The non-root processes wait for the root to recover or return an error to the calling routing. FEMPI calls that fall in this category include *MPI_Bcast*, *MPI_Gather*, *MPI_Scatter*, *MPI_Alltoall* and *MPI_Barrier*.

In the case of process-specific calls, failures do not impact any other process and hence do not mandate any fault handling in the other non-faulty processes. FEMPI calls in this category include *MPI_Type_**, *MPI_Comm_rank* and *MPI_Comm_size*. However, recovery of the faulty process is still required. Group-specific calls are similar to collective communication calls. Although there is no explicit communication of application messages, these calls involve the exchange of several control messages. The functions in this category can be considered as a

collective communication calls with all the processes involved being roots. FEMPI function calls in this category include *MPI_Init* and *MPI_Finalize*.

4.4 Performance Analysis

In this section, we discuss the performance of FEMPI based on experiments conducted on both a prototype system and a ground-based cluster system. Beyond providing results for scalability analysis, experiments on the ground-based cluster system demonstrate FEMPI's flexibility and the ease with which space scientists may develop their applications on an equivalent platform. The next section describes the setup for these systems, followed by a discussion of the performance of FEMPI on failure-free systems. We discuss the performance of FEMPI in systems with failures in Section 4-5.

4.4.1 Experimental Setup

For the current research phase of the NASA's New Millennium Project, a prototype system has been designed to mirror when possible and emulate when necessary the features of a typical satellite system. The system to be launched in 2009 has been developed at Honeywell Inc. and the University of Florida. The prototype hardware shown in Figure 4-5 consists of a collection of single-board computers, some augmented with FPGA coprocessors, a power supply and reset controller for performing power-off resets on a per-node basis, redundant Ethernet switches, and a development workstation acting as satellite controller.

Six Orion Technologies COTS Single-Board Computers (SBCs) are used to mirror the specified data processor boards to be featured in the flight experiment (four) and also to emulate the functionality of the radiation-hardened components (two) currently under development. Each board is comprised of a 650MHz IBM 750fx PowerPC, 128MB of high-speed DDR SDRAM, dual Fast Ethernet interfaces, dual PCI mezzanine card slots, and 256MB of flash memory, and runs MontaVista Linux. 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 from which to leverage. 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 system over bus-based approaches currently in use by space systems. Of the six boards, one is used as the primary system controller node. Another SBC is used to emulate both the backup controller (that takes over the control functions when the primary controller encounters a failure) and a central data store (used for storing input and output data of applications along with application checkpoints). The data store is currently implemented as a 40 GB PMC hard drive, while the flight system will likely include a radiation-hardened solid-state storage device. The remaining four SBCs are used as data processing nodes which we will exercise to show the performance of FEMPI.

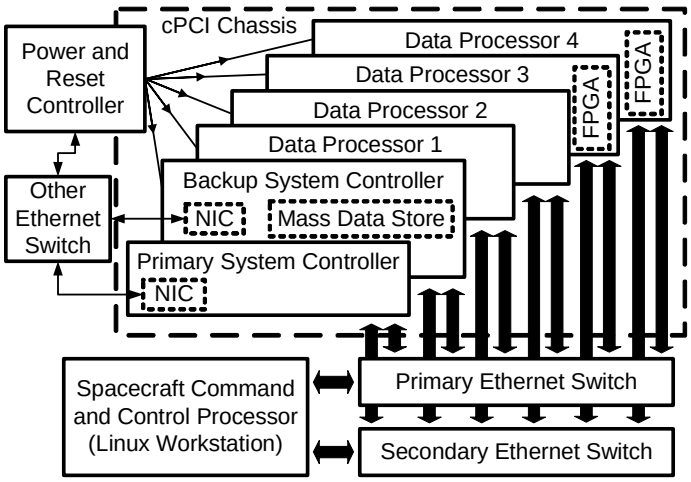


Figure 4-5. System configuration of the prototype testbed

In order to test the scalability of FEMPI and see how it will perform on future systems beyond the six-node testbed developed to emulate the exact system to be flown in 2009, experiments were conducted on a 16-node cluster. The cluster consists of traditional server machines each with 2.4GHz Intel Xeon processors running Redhat Linux 9.0, 1GB of DDR RAM and connected via Gigabit Ethernet network. As previously described, the fact that the

underlying PowerPC processors in the testbed and the Xeons in the ground-based 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 using FEMPI from a ground-based cluster to the embedded space system is as easy as recompiling on the different platform.

4.4.2 Results and Analysis

In this section, we present the performance results of FEMPI on failure-free systems where we report the general performance of FEMPI. The relative performance of FEMPI in comparison with conventional and commonly used MPI variants such as MPICH[54] and LAM/MPI[56] are also discussed in this section.

4.4.2.1 Point-to-point communication

Figure 4-6 shows performance of FEMPI's point-to-point communication on the Xeon platform. The performance of the message passing middleware is reported in terms of the application-level throughput. A specified amount of data is sent from one node (sender) to another (receiver) and time taken to finish the communication completely is measured. Throughput is measured as the ratio of the size of data transferred and the time taken to transfer the data. In order to avoid any transient errors, the results reported are averages of 100 trials.

For one-sided communications using *MPI_Send* and *MPI_Recv*, the maximum throughput using FEMPI reaches about 590 Mbps on the Xeon cluster with 1 Gbps links. This maximum throughput value is comparable to the throughput provided by the conventional MPI implementations (i.e., MPICH and LAM/MPI). It can be seen that the throughput with MPICH saturates at approximately 430 Mbps while that for LAM/MPI saturates at approximately 700

Mbps. However, both MPICH and LAM/MPI perform better for smaller data sizes because of data buffering. Presently, FEMPI does not implement buffering of data. It is true that LAM/MPI performs better than FEMPI for all data sizes; however, it must be noted that we get the additional benefit of fault tolerance with FEMPI.

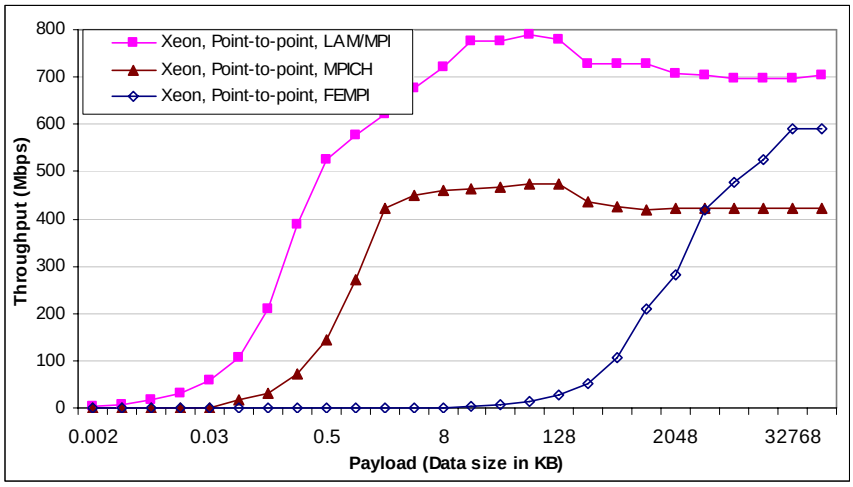


Figure 4-6. Performance of point-to-point communication on a traditional cluster

Figure 4-7 shows the performance of FEMPI's point-to-point communication calls, both one-sided and two-sided, on the prototype testbed. Due to limited resource availability in the embedded system testbed it was not possible to use the conventional MPI variants that require more resources. Hence, we do not show the performance of these variants on the testbed. On the embedded cluster with 100 Mbps links and slower processors compared to the Xeon cluster, the maximum throughput is approximately 31 Mbps, which is again comparable to what can be achieved using popular MPI implementations on Fast Ethernet networks. For the *MPI_Sendrecv* function call (two-sided communication), the throughput is slightly lower than the one-sided communication using *MPI_Send* and *MPI_Recv*. In a system with two nodes, data is exchanged between the two processing nodes simultaneously in a sendrecv call leading to possible network congestion and hence lowering the throughput. For systems with more than two nodes, the

throughput can increase if the send and receive (that are part of a node) are not with a same node thereby enabling simultaneous communication with different nodes (i.e., data can be sent to one node while receiving data from a different node).

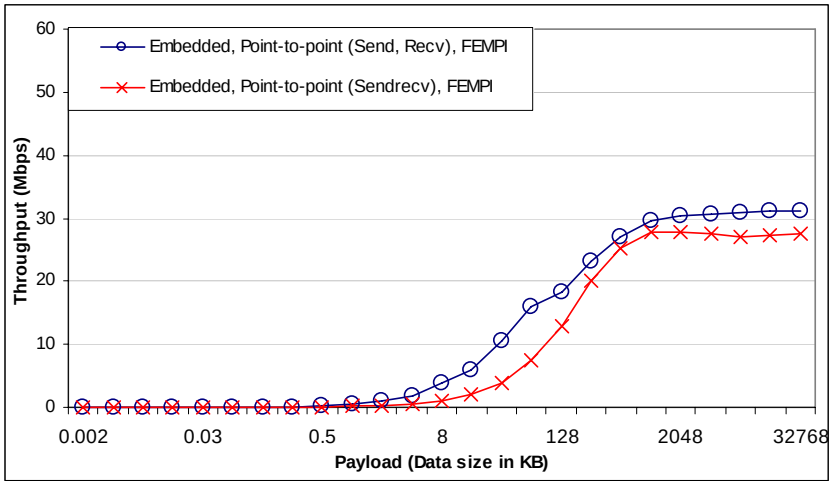


Figure 4-7. Performance of point-to-point communication on prototype testbed

4.4.2.2 Collective communication

Collective communication involves the transfer of messages between multiple processes in the system. Figure 4-8 shows performance of FEMPI's broadcast communication on the Xeon cluster in comparison to that of the conventional MPI variants on three nodes. The throughput trend for the broadcast communication is similar to that for point-to-point communication. The throughput for FEMPI saturates at approximately 590 Mbps, while that for LAM/MPI at 710 Mbps and 490 Mbps for MPICH. For larger data sizes, FEMPI performs better than MPICH although it is poorer than LAM/MPI. Since broadcast is a collective communication call that generally involves more than two nodes, we studied the performance in systems with up to 16 nodes. However, there was not much variation in the throughput results. This behavior is attributed to that fact that the broadcast call is primarily dominated by the root node and the non-roots do not have a major impact on the performance. Similar to the Xeon cluster, the

performance of FEMPI's broadcast communication on the prototype testbed (not shown here) resembled the trend of point-to-point communication.

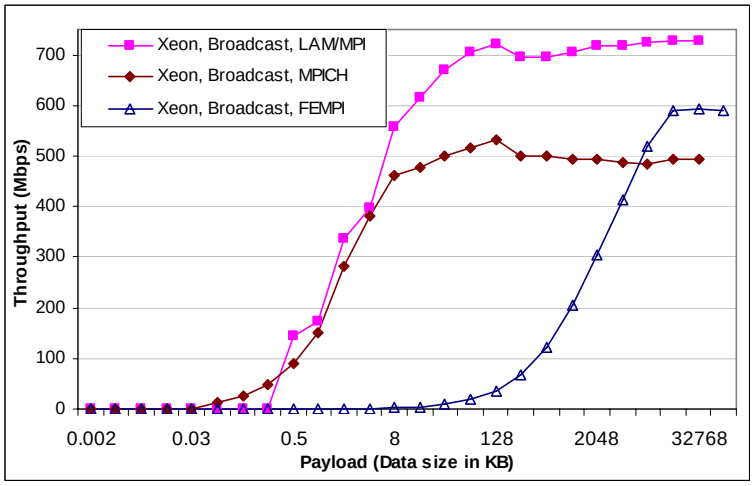


Figure 4-8. Performance of broadcast communication on a traditional cluster

Figure 4-9 gives the timing for synchronization of nodes with FEMPI, MPICH and LAM/MPI using the barrier synchronization function *MPI_Barrier* defined by the MPI standard. It can be seen from the figure that in most cases, the synchronization time in FEMPI is lower than that in LAM/MPI and MPICH on the Xeon cluster. The synchronization time is on the order of tens of milliseconds for all the three MPI variants in smaller systems. However, as the system size goes beyond 8 nodes, the synchronization time for MPICH increases beyond 100 ms. This abrupt increase can be attributed to the design of barrier using tree structures in MPICH. Such an increase is also seen for LAM/MPI when system size is increased beyond 4 nodes. However, the synchronization time for LAM/MPI is almost independent of system size beyond this point. For both MPICH and FEMPI, the synchronization time increases with system size but the rate is lesser for FEMPI. Synchronization in FEMPI does not depend on any tree structure rather it is distributed and only depends on explicit synchronization messages broadcast using DMS.

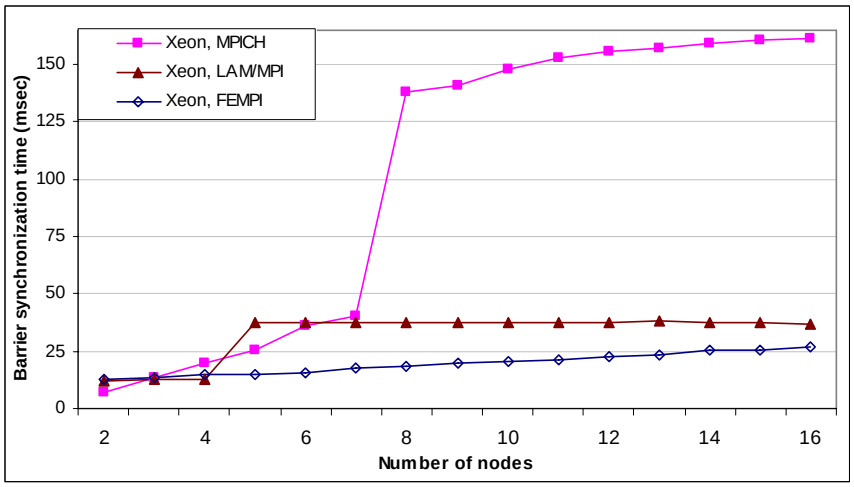


Figure 4-9. Performance of barrier synchronization on a traditional cluster

Table 4-2 gives the timing for synchronization of nodes with FEMPI on the prototype testbed. As expected, the synchronization time is higher on the embedded system, which can be attributed to the slow processing power and network.

Table 4-2. Barrier synchronization using FEMPI on prototype testbed

No. of nodes	Synchronization Time (ms)
2	23.5
3	30.0
4	36.5

Figures 4-10 and 4-11 show the performance of other collective communication calls including *MPI_Gather* and *MPI_Scatter* on the Xeon cluster and on the prototype testbed respectively. We only report the performance of FEMPI here and do not compare the performance with LAM/MPI and MPICH. As mentioned earlier, both these MPI variants have data buffering capabilities which are presently absent in FEMPI (provisions for buffering increases the complexity of providing fault tolerance). Ability to buffer data boosts performance of these communication calls, as transactions with a node can be processed while simultaneously sending/receiving data to/from another node. Hence, it would be unfair to compare the performance of FEMPI's gather and scatter calls with that of LAM/MPI and MPICH.

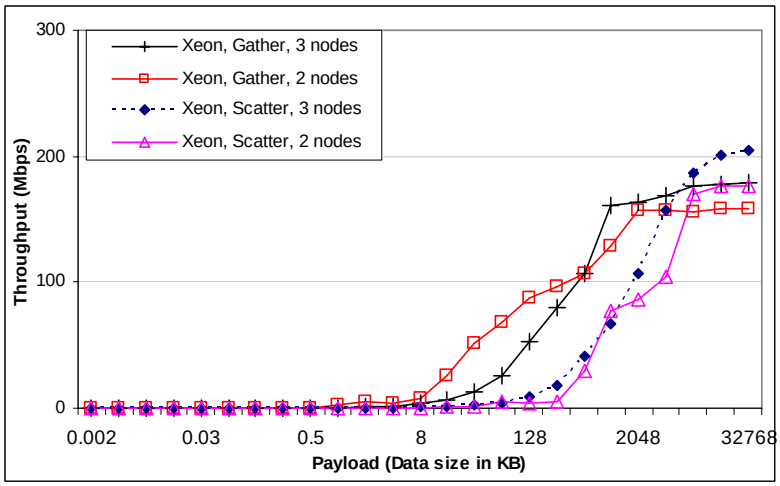


Figure 4-10. Performance of gather and scatter communication on a traditional cluster

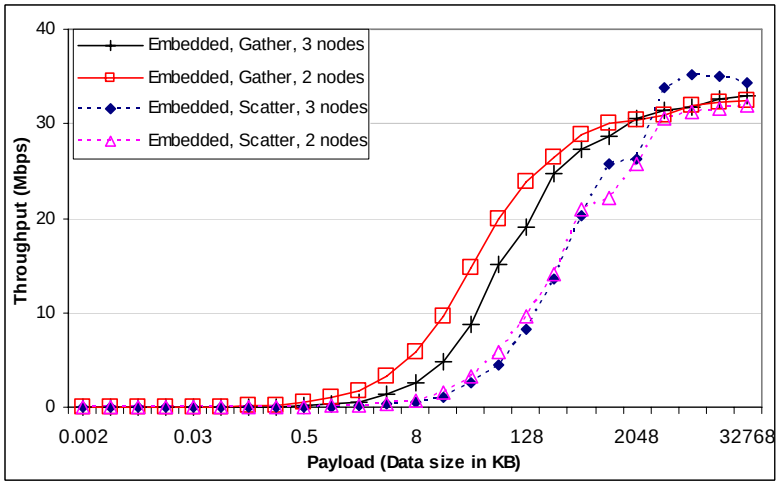


Figure 4-11. Performance of gather and scatter communication on prototype testbed

It can be seen from Figures 4-10 and 4-11 that the performance of *MPI_Gather* and *MPI_Scatter* is poor compared to broadcast communication. The root node individually communicates with each client to transfer data (data being unique to each client) in scatter and gather communication while with the broadcast the root publishes the same data to all the clients using a single transfer. The *MPI_Gather* function is used to gather data at a single root node from all the other nodes in the system. Local data at the root node is copied (*memcpy*) from the send buffer to the receive buffer while data received from the non-root nodes are placed directly

in the receive buffer. MPI_Scatter on the other hand is used to transfer data from the root node to all the non-root nodes. The throughput is low because the root node serializes the 'receives' (in case of gather) and 'sends' (in case of scatter) with the client nodes that are ready to send/receive their data. Once communication has been initiated by the root with a client, all transactions with that client are finished before moving to another client.

So far, we have reported the general performance of FEMPI on failure-free systems. From the results presented, it can be seen that FEMPI performs reasonably well for both point-to-point and collective communication in comparison to conventional non-fault-tolerant MPI variants. In addition, and more importantly, FEMPI provides fault tolerance and helps applications to successfully complete execution despite faults in the system. In the next section, we study the performance benefits provided by FEMPI when used in systems with faults.

4.5 Performance Analysis of Failure Recovery

In the previous section, we discussed the performance of FEMPI on a failure-free system. In this section, we qualitatively study the performance of FEMPI on systems with failures. We analytically derive the time required to recover from failures in FEMPI and in traditional non-fault-tolerant MPI variants. We then quantitatively analyze the recovery times using an actual application kernel in Section 6.

It is assumed that the system provides checkpointing functionalities that the application can use and hence would not have to restart from the beginning every time a failure is encountered. Checkpointing refers to the process of saving program/application state, usually to a stable storage (i.e., taking the snapshot of a running application for later use). Checkpointing forms the crux of rollback recovery and hence fault tolerance, debugging, data replication and process migration for high-performance applications. As in our prototype testbed, the stable storage is generally centralized and hence could be a bottleneck. However, centralized designs

are preferred because they are much simpler to realize and it is easier to coordinate the multiple processes of a parallel application trying to access the storage simultaneously.

4.5.1 Non-fault-tolerant MPI variants

We discuss the various timing elements involved in the recovery of an application executing in a system with a MPI variant that does not inherently support fault tolerance. Following are steps involved in recovering an application from a failure:

i. Detect the occurrence of a failure on a node
ii. Forcefully stop the application processes on all other nodes executing the application
iii. Redeploy the application processes (assuming that the failure has been rectified)
iv. All processes read checkpoint data from the stable storage and resume execution from the previously checkpointed stable state
v. Re-perform computation performed after previous checkpoint and before the failure occurred (to reach the same state before the failure)

The total time to recover from a failure (i.e., the total effective computation time lost due to the failure and subsequent recovery) can be calculated as:

$$T_{FR} = T_{FD} + T_{SA} + T_{RA} + T_{RC} + T_{CL} \quad (4.1)$$

where,

T_{FR} : Total time to recover from a failure
T_{FD} : Time to detect the failure
T_{SA} : Time to forcefully stop application processes on all nodes
T_{RA} : Time to redeploy application processes on all nodes
T_{RC} : Time for all application processes to read the checkpoint from stable storage
T_{CL} : Computation time lost since previous checkpoint (maximum value out of all processes)

4.5.2 Failure Recovery Timing in FEMPI

We discuss the various timing elements involved in the recovery of an application executing in a system with FEMPI in this section. Following are steps involved in recovering an application from a failure:

i. Detect the occurrence of a failure on a node

ii. Inform the other nodes in the system about the failure

iii. Redeploy only the application process that failed on a healthy node

iv. The recovered process alone reads the checkpoint data from the stable storage and resumes execution from the previously checkpointed stable state

v. The recovered process alone re-performs computation performed after previous checkpoint and before the failure occurred (to reach the same state before the failure)

The total time to recover from a failure (i.e., the total effective computation time lost due to the failure and subsequent recovery) can be calculated as:

$$T_{FFR} = T_{FFD} + T_{FFI} + T_{FRA} + T_{FRC} + T_{FCL} \quad (4.2)$$

where,

T_{FFR} : Total time to recover from a failure

T_{FFD} : Time to detect the failure

T_{FFI} : Time to inform application processes on all nodes about the failure

T_{FRA} : Time to redeploy application processes on one node

T_{FRC} : Time for restored process to read the checkpoint from stable storage

T_{FCL} : Computation time lost by the recovered process since previous checkpoint

4.5.3 Failure Recovery Timing: Comparative Analysis

In the previous sections, we identified the timing elements involved in the recovery of an application in the event of a failure. In this section, we compare the recovery timing for FEMPI with that for non-fault-tolerant MPI variants. The comparison shows the improvement in application execution time enabled by FEMPI justifying the development of a fault-tolerant MPI.

Comparing the corresponding timing elements in (4.1) and (4.2), the failure recovery of FEMPI will be worse than that of non-fault-tolerant MPI implementations if at least one of the following conditions are true:

- T_{FFI} is greater than T_{SA}
    - This condition is not possible; T_{FFI} being the time to just broadcast one small message is negligible while T_{SA} is much more complex than that. Sending a message to all processes and gracefully stopping them requires more time than just sending a message.
- T_{FRA} is greater than T_{RA}
    - This condition is not possible; Even in worst case, T_{FRA} can only be equal to $T_{RA,}$ implying that the time taken to deploy application processes does not depend on number of processes.
- T_{FRC} is greater than T_{RC}
    - This condition is not possible; Even in worst case, T_{FRC} can only be equal to $T_{RC,}$ implying that the time taken to read checkpoint data does not depend on number of processes accessing the stable storage simultaneously.
- T_{FCL} is greater than T_{CL}
    - This condition is not possible; Even in worst case, T_{FCL} can only be equal to $T_{CL,}$ implying maximum computation time is lost at the failed process.

All of the above timings elements can be worst-case scenarios for FEMPI. Nevertheless, FEMPI's recovery time will at least be equal to that of non-fault-tolerant MPI implementations, if not better. In general, a timing element in FEMPI is better than the corresponding timing in non-fault-tolerant MPI implementations in all of the above cases. Hence, total recovery time is much shorter for FEMPI. Additionally, FEMPI's recovery method also provides additional computation hiding. For example, if computation time on a healthy process until the next checkpoint is longer than the time required for the failed process to recover and reach its subsequent checkpoint state, then the failure recovery process has no impact on the healthy process. The computation hiding can be explained as follows:

T_{FCH} : Time for computation that can be performed in the healthy processes while the failed process is recovering (maximum value among all processes)

T_{FCN} : Time until next checkpoint in the healthy processes since the time the failure occurred in another process (maximum value among all processes)

T_{FCNR} : Time until next checkpoint in the recovered process from its restarted state

If, $(T_{FCN}-T_{FCH}) > T_{FCNR}$, then effectively the recovery time can be given by $(T_{FFR}-T_{FCH})$. In general, the computation might be stalled at healthy processes while the failed process is recovering, increasing the total application execution time by a value equal to the recovery time. Such is the case in non-fault-tolerant MPI implementations. However, in the case of FEMPI, if healthy processes can perform computation while the failed process is recovering, then the recovery minimally impacts the overall execution time of the application and hence in effect the recovery time can be perceived to be reduced. In summary, FEMPI-based recovery is faster and more effective when compared to non-fault-tolerant MPIs. We studied the performance improvement analytically in this section while in the next we study the improvement quantitatively using an application kernel as a case study.

4.6. Parallel Application Experiments and Results

So far, we have discussed the raw performance of FEMPI in comparison with other MPI implementations. However, the final target for such a communication service is for use by applications. Several space science applications have been proposed for architectures similar to ours. LU decomposition (LUD) is a typical kernel that is part of several space applications, such as Space-Time Adaptive Processing, and would benefit from a service such as FEMPI. In this section, we report the performance of FEMPI when used in a real application, LUD, in terms of execution time and failure recovery time. As mentioned earlier, it is not possible to execute applications using MPICH on the prototype testbed due to resource constraints. Hence, in order

to comparatively analyze the performance of FEMPI with MPICH, we restrict our experiments to the Xeon cluster. We study the scalability as well by executing the application on systems with up to 16 nodes.

4.6.1 LU Decomposition

In linear algebra, the LUD is a matrix decomposition which writes a matrix as the product of two triangular matrices. This decomposition is used in numerical analysis to solve systems of linear equations or find the inverse of a matrix. Let A be a square matrix. An LUD is a decomposition of the form, $A = LU$, where L and U are lower and upper triangular matrices of the same size, respectively. The decomposition implies that L has only zeros above the diagonal and U has only zeros below the diagonal. We can solve the equation $Ax = b$ for x, where A is an n x n matrix, x an n-dimensional column vector of unknowns, and b an n-dimensional column vector, by factoring A into the product of a unit lower triangular matrix L and an upper triangular matrix U, provided such factors exist.. Then we can write $Ax = b$ as $LUx = b$. To obtain the solution, we first solve $Ly = b$ for y and then solve $Ux = y$ for x.

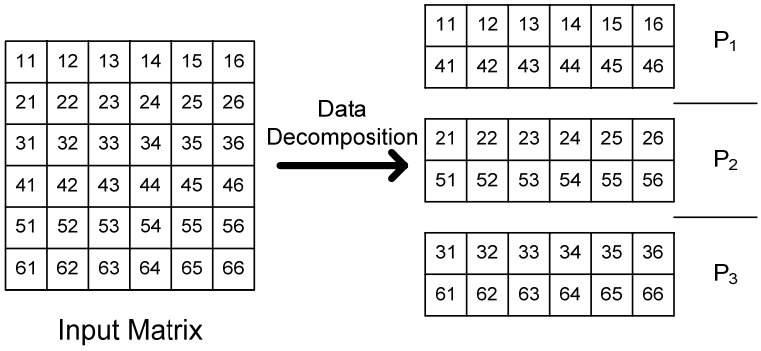


Figure 4-12. Data decomposition in parallel LUD

We have parallelized the LUD algorithm with row-wise cyclic striping and partial pivoting similar to ScaLAPACK, a standard high-performance system benchmark[59]. The input matrix is divided row-wise among the processing nodes, with each node operating on equal number of

rows. Figure 4-12 shows the decomposition of a 6 x 6 matrix on to 3 processors. Processor P_1 gets rows 1 and 4, P_2 gets 2 and 5, while P_3 gets rows 3 and 6.

During each iteration, the matrix can be logically divided in to three sub-matrices *X*, *Y*, and *Z* as shown in Figure 4-13. Each iteration consists of three steps: (1) The column of nodes that holds *X* factorizes their part and distributes the pivot information in its corresponding rows. (2) Each column of nodes exchanges the remaining part of the pivot rows. The decomposition is performed on the actual values of *Y* and the result is broadcasted. (3) All nodes update *Z* in parallel. Then the next iteration continues to complete the factorization of *Z*.

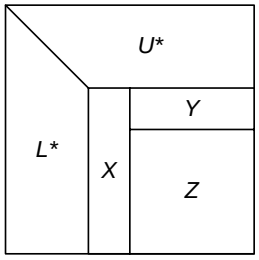


Figure 4-13. Parallel LUD algorithm

4.6.2 Failure-free Performance

The execution times of the parallel LUD application on failure-free systems with node counts ranging from 2 to 16 are shown in Figure 4-14. It should be mentioned here that the execution times reported do not include the time to start up the processes on the participating nodes. FEMPI uses job management daemons (part of the job management system developed for the prototype testbed) that reside permanently on each node to start the processes and hence is very fast. On the other hand, MPICH has a process in one central node remotely logging in to every node to start up the process, which takes a much longer time. Hence, for a fair comparison, we do not include the start up time in our execution time results.

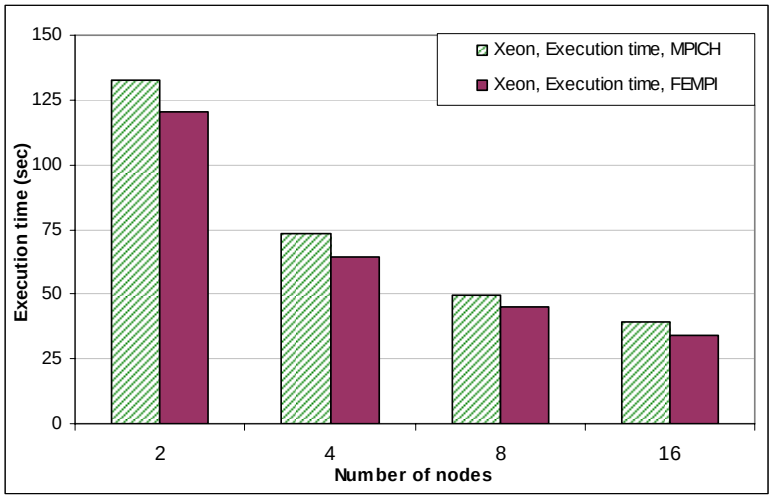


Figure 4-14. Failure-free execution time of parallel LUD application kernel with increasing system size

The input matrix is of size 4160 x 4160. Each element in the matrix is of type 'double' (8 bytes) and hence the size of the input matrix is 138.44 MB. As the number of nodes in the system increases, execution time decreases for both FEMPI and MPICH. With the matrix and hence the communication messages being large, MPICH cannot use its buffering capability as the data size exceeds the size of the buffer. With FEMPI and MPICH both without buffering, it can be seen from the figure that the application executes faster with FEMPI. Lower execution time highlights the better performance of FEMPI. However, as the system size increases, the size of the sub-matrix in each node and hence the communication messages decreases explaining the decrease in the performance difference between FEMPI and MPICH.

To compare the performance of FEMPI versus MPICH for shorter running applications with smaller communication messages, we executed the application for an input matrix size of 12 x 12. Although the execution time using FEMPI was larger than that with MPICH having buffering capability, the difference was minimal. For example, in a 2-node system, the execution time using FEMPI was 8 ms while that using MPICH was 4 ms. Likewise, in a 3-node system, the execution time using FEMPI was 6 ms while that using MPICH was 3 ms. In summary, in

failure-free systems, the performance of FEMPI is comparable to MPICH for short applications and better than MPICH for long running applications.

4.6.3 Failure Recovery Performance

In the previous section, we studied the performance of FEMPI when used in an actual application in a failure-free system. In this section, we study the performance in systems with failures in terms of the failure recovery time. The parallel LUD kernel uses the RECOVER mode of FEMPI for recovery from failures, requiring the failed process to be re-spawned back to its healthy state either on the same or a different processing node. When a FEMPI function is called to communicate with a failed process, the function call is halted until the process is successfully re-spawned. When the process is restored, the communication call is completed and the control is returned back to the application. On the other hand, with MPICH, when a process fails all the other processes are stopped and the entire application is restarted. The restarting process which is the failed process in case of FEMPI and all the processes in case of MPICH restart from the previous stable checkpoint state stored in the stable storage. As part of the computation, all processes checkpoint their states regularly.

Figures 4-15 and 4-16 compare the recovery time (time from when the failure occurred to when the application is back to the same state before the failure) for the parallel LUD application with FEMPI and MPICH for varying system sizes. While Figure 4-15 shows the recovery time for a small matrix (12 x 12), Figure 4-16 shows the recovery time for a large matrix (4160 x 4160). We show the results for two different versions of MPICH, with rsh and without rsh. While the MPICH with rsh includes the time to startup the processes (via remote login), MPICH without rsh does not. For the MPICH without rsh case, we assume that MPICH uses job management daemons (similar to FEMPI) and is presented for fair comparison with FEMPI.

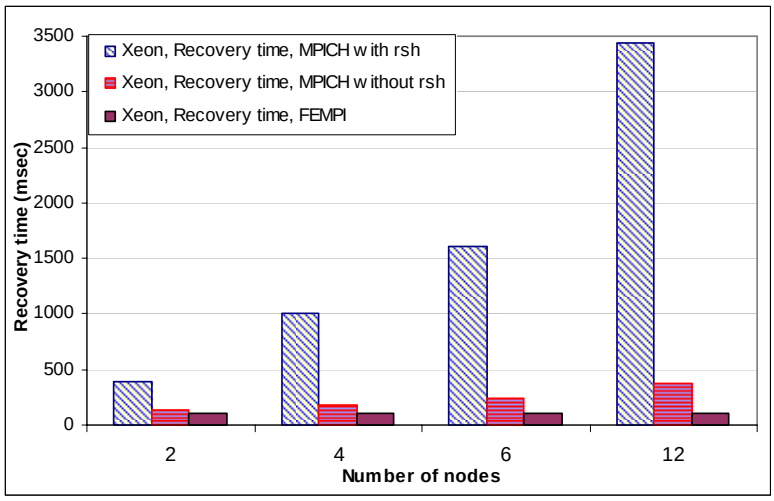


Figure 4-15. Recovery time from a failure with increasing system size for applications with small datasets

For both large and small applications, the recovery time of FEMPI is better than that of MPICH thereby improving the availability of the system. MPICH with rsh performs the worst due to the remote logging overhead. MPICH without rsh is comparable to FEMPI for smaller systems, but as the system size grows the performance worsens. For MPICH, the impact of all processes having to be restarted and all of them accessing checkpointing simultaneously becomes pronounced as the number of processes increases. The impact is even more pronounced for larger applications where the amount that has to be read from the stable storage is large. For a 12-node system running the smaller application in Figure 4-15, the recovery time for FEMPI is 108 ms while it is 383 ms for MPICH without rsh and 3,443 ms for MPICH with rsh. For a 16-node system running the larger application in Figure 13, the recovery time for FEMPI is 128 ms, while that for MPICH without rsh is 803 ms, and 4983 ms is needed for MPICH with rsh.

It should be mentioned here that the recovery times reported here are for recovery from one failure. If there are multiple failures during the course of the application execution, which is expected in failure-prone space systems, the overall time spent recovering from failures would be

equal to the time to recover from one failure multiplied by the number of failures. In summary, it can be seen from the results that FEMPI performs better than MPICH in terms of the execution time in failure-free systems and in terms of recovery time in systems with failures.

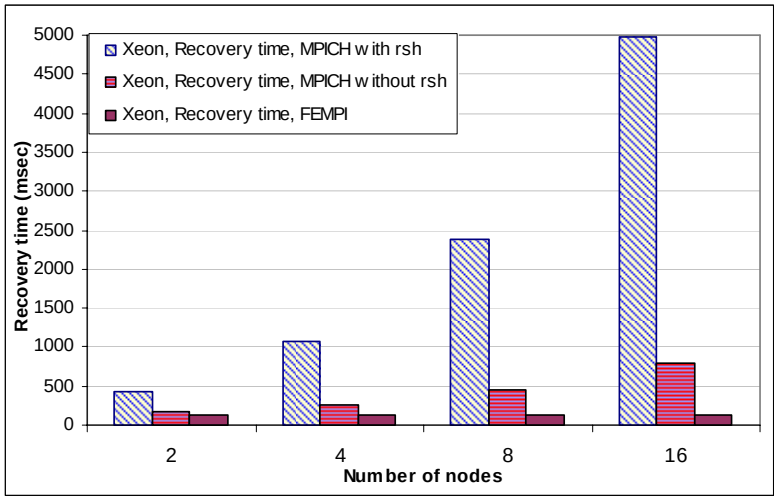


Figure 4-16. Recovery time from a failure with increasing system size for applications with large datasets

4.7 Conclusions and Future Research

In this phase of the research, we have taken a direct approach to provide fault-tolerance and improve the availability of the HPC system in space by investigating and proposing a new lightweight fault-tolerant message-passing middleware. We presented the design and analyzed basic characteristics of a new fault-tolerant, lightweight, message passing middleware for embedded systems called FEMPI. This fault-tolerant middleware reduces the impact of failures on the system and is particularly important for support of applications in harsh environments with mission-critical requirements. The lightweight fault-tolerant MPI for low resource environments is the first of its kind to our knowledge and its design is the primary research contribution of this phase of research. The recovery time of the system is improved allowing unimpeded execution of applications as much as possible. A key focus of the message-passing

middleware architecture has been toward tradeoffs between performance and fault tolerance, with emphasis on the latter but forethought of the former.

Results from experiments on performance of FEMPI on an embedded cluster system with PowerPCs and on a traditional Xeon cluster were presented. Both point-to-point and collective communication calls performed well, providing a maximum throughput of around 590 Mbps on a Xeon cluster with a 1 Gbps network and 31 Mbps on a cluster of PowerPC-based embedded systems with 100 Mbps network, both roughly comparable to conventional, non-fault-tolerant MPI middleware. FEMPI also achieves relatively low synchronization times. We also presented the results of using FEMPI with an application kernel on the Xeon cluster in terms of execution times, failure recovery time and scalability. FEMPI performs better than MPICH in terms of all the above-mentioned performance metrics for large (long running) applications while performance is comparable for small (short running) applications.

As future work, FEMPI will be tested with several other case studies including prominent space applications, with and without injected faults, to provide additional insight on the benefits and costs of fault tolerance in this environment. Based on the requirements of these applications, more function calls will also be developed. Analysis of the impact of the addition of buffered and asynchronous communication calls on fault tolerance, along with the REMOVE mode of recovery is also a promising future work.

CHAPTER 5
CONCLUSIONS

With the fast advancement of space research in the past few decades, the demand of space missions for data returns from their resources in space has highly increased. As the traditional approach of data collection and transmission is no longer viable, there have been several research efforts to make HPC systems available in space in order to have enough "on board" processing power. Such efforts have led to the idea of using COTS components to provide HPC in space. The susceptibility of COTS components to SEUs deems the development of fault-tolerant system functions to manage the resources available and improve availability of the system in space. Several techniques exist in traditional HPC to provide fault tolerance and improve overall computation rate but adapting these techniques for HPC in space is a challenge due to the resource constraints.

In this dissertation, we address this challenge by investigating approaches to improve and complement HPC in space. Three techniques are investigated in three different phases of this dissertation to improve the effective utilization and availability of HPC in space. In Phase 1, we improve the useful computation time of the system by optimizing checkpointing-related defensive I/O. In Phase 2, we improve the overall execution time of the application by simulatively analyzing scheduling heuristics for application task scheduling. Phase 3 improves the availability of the system by designing a new lightweight, fault-tolerant, message-passing middleware.

As part of the first phase of this research, the useful computation time of the system is increased by optimizing checkpointing-related defensive I/O. We studied the growth in performance of the various technologies involved in high-performance computing, highlighting the poor performance growth of disk I/O compared to other technologies. Presently defensive

I/O that is based on checkpointing is the primary driver of I/O bandwidth rather than productive I/O that is based on processor performance. There are several research efforts to improve the process of checkpointing itself but are generally application-specific. To the best of the author's knowledge, there have been no successful attempts to optimize the rate of checkpointing to reduce the overhead of fault-tolerant mechanisms. Such an approach is not application- or system-specific and is applicable to both HPC systems in space and on ground. The primary research contribution of this phase is the development of a new model for checkpointing process and in so doing identify the optimum rate of checkpointing for a given system. Checkpointing at an optimal rate reduces the risk of loosing much computation on a failure while increasing the amount of useful computation time available.

We developed a mathematical model to represent the checkpointing process in large systems and analytically modeled the optimum computation time within a checkpoint cycle in terms of the time spent on checkpointing, the MTBF of the system, amount of memory checkpointed and sustainable I/O bandwidth of the system. Optimal checkpointing maximizes useful computation time without jeopardizing the applications due to failures while minimizing the usage of resources. The optimum computation time leads to the minimal wastage of useful time by reducing the time spent on overhead tasks involving checkpointing and recovering from failures. In order to see the impact of checkpointing overheads and the overhead to recover from failures on the I/O bandwidth required in the systems, we analyzed the overall execution time of applications including these overhead tasks relative to the ideal execution time in a system without any checkpointing or failures.

In order to verify our analytical model, we developed discrete-event simulation models to simulate the checkpointing process in HPC systems. In the simulation models, we used

performance numbers that represent systems ranging from small embedded cluster systems (space-based) to large supercomputers (ground-based). The simulation results matched closely with those from our analytical model, verifying the correctness of our analytical model. As future work, the model can be experimentally verified with real systems running scientific applications. When successfully verified, the model can be used to find the optimal checkpointing frequency for various HPC and HPEC systems based on their resource capabilities.

As part of the second phase of this research, the overall execution time of applications is decreased by analyzing methodologies for effective task scheduling. Reducing the overall execution time of applications exposes the applications to lesser number of faults in the system. Recently, systems augmented with FPGAs offering a fusion of traditional parallel and distributed machines with customizable and dynamically reconfigurable hardware have emerged as a cost-effective alternative to traditional systems. Dynamic scheduling of large-scale HPC applications in such parallel reconfigurable computing (RC) environments is one such challenge and has not been sufficiently studied to our knowledge.

We analyzed the performance of several scheduling heuristics that can be used to schedule application tasks on parallel RC systems that can potentially be part of HPC systems in space. Typical HPC applications and FPGA/RC resources were used as representative cases for our simulations. A performance model was also developed to predict the overall execution time of tasks on RC resources. The model was used by our scheduling heuristics to schedule the tasks. Based on the performance metric that is critical for a given environment, the corresponding scheduling heuristics can be used to schedule tasks. The performance model and the extensive

simulative analysis of scheduling heuristics for FPGA-based RC applications and platforms is the first of its kind and is the primary research contribution of this phase.

The intent of this phase of research is to analyze several scheduling heuristics and compare their performance to identify the suitable candidates for use in space-based HPC systems. We have shown the performance results for average makespan. But, based on the performance metric critical for a given environment, the corresponding scheduling heuristics can be used to schedule tasks. In future, the knowledge gained so far to suggest suitable scheduling heuristics for an experimental scheduler can be applied in the job management service intended for the space-based HPC system being developed at the HCS Research Laboratory at the University of Florida.

As part of the third phase of this research, we take a direct approach to provide fault-tolerance and improve the availability of the HPC system in space by designing a new lightweight fault-tolerant message-passing middleware. We designed an application-independent fault-tolerant message passing middleware called FEMPI, a lightweight fault-tolerant design (implementation) of the MPI standard. The lightweight fault-tolerant MPI for low resource environments is the first of its kind to our knowledge and its design is the primary research contribution of this phase of research. The fault-tolerant middleware reduces the impact of failures on the system and is particularly important for support of applications in harsh environments with mission-critical requirements. The recovery time of the system is improved allowing unimpeded execution of applications as much as possible. A key focus of the message-passing middleware architecture has been toward tradeoffs between performance and fault-tolerance, with emphasis on the latter but forethought of the former.

In this phase, we focused on designing a message-passing middleware architecture that can successfully address the trade-offs between performance and fault-tolerance. Several fault-tolerant MPI research efforts were surveyed and their qualitative performances were comparatively analyzed. With the wide knowledge base that we gained out of the study and based on previous MPI development experience we proposed a new framework for fault-tolerant MPI as a comprehensive solution to address fault tolerance in MPI and thereby improve productivity and throughput of HPC systems in space. The architecture is unique, flexible and independent of any particular MPI implementation which would enable the use of the design and architecture by any message-passing middleware.

The primary goal of this phase of research is to design a lightweight fault-tolerant middleware and provide for graceful degradation of applications while minimizing recovery times. We designed FEMPI and results from experiments on performance of FEMPI on an embedded cluster system with PowerPCs and on a traditional Xeon cluster were presented. We also presented the results of using FEMPI with an application kernel on the Xeon cluster in terms of execution times, failure recovery time and scalability. As future work, FEMPI will be tested with several other case studies including prominent space applications, with and without injected faults, to provide additional insight on the benefits and costs of fault tolerance in this environment. Based on the requirements of these applications, more function calls can also be developed.

Space environment is harsh and prone to failures while being limited in resources. Low overhead fault-tolerance methodologies that do not tax the space-based HPC systems is a key requirement. In this dissertation, we have addressed solutions to adapt fault-tolerant techniques from traditional ground-based HPC systems to improve effective utilization and availability of

HPC in space. This research provides novel techniques to improve the reliability of HPC in space with minimal effort to transition from HPC on the ground. The solutions are geared to be lightweight without straining the system while maintaining the quality and reliability of the fault-tolerant schemes.

LIST OF REFERENCES

[1] 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. IEEE/AIAA Aerospace Conference*, Big Sky, MT, March 4-11, 2006.

[2] W. J. Larson, J. R. Wertz, "Space Mission Analysis and Design," 3rd Edition, Microcosm Press/Kluwer Academic Publishers, 1999.

[3] "Scientific and Technical Information," http://www.sti.nasa.gov/products.html#pubtools (Accessed October 2, 2006)

[4] J. Samson, L. Torre, P.Wiley, T. Stottlar, and J. Ring, "A Comparison of Algorithm-Based Fault Tolerance and Traditional Redundant Self-Checking for SEU Mitigation," *Digital Avionics Systems Conference*, Daytona Beach, FL, October 14-18, 2001.

[5] "ASCI Purple Statement of Work, Lawrence Livermore National Laboratory", http://www.llnl.gov/asci/purple/Attachment_02_PurpleSOWV09.pdf (Accessed: October 2, 2006)

[6] G. Schocht, I. Troxel, K. Farhangian, P. Unger, D. Zinn, C. Mick, A. George, and H. Salzwedel, "System-Level Simulation Modeling with MLDesigner," *11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Orlando, FL, October 2003.

[7] "Cramming More Components onto Integrated Circuits," *Electronics*, Vol. 37, No. 8; April 19, 1965.

[8] "LINPACK," http://www.netlib.org/linpack/ (Accessed: October 2, 2006)

[9] J. Dongarra, P. Luszczek, A. Petitet "The LINPACK Benchmark: Past, Present, and Future," *Concurrency and Computation*, *Practice and Experience 15*, pp. 1-18, 2003.

[10] "Top 500 Supercomputer Sites," http://www.top500.org/ (Accessed: October 2, 2006)

[11] "HITACHI eyes 1 TB Desktop Drives, " http://www.pcworld.com/news/article/0,aid,120279,00.asp (Accessed: October 2, 2006)

[12] S. Heinze, M. Bode, A. Kubetzka, O. Pietzsch, X. Nie, and S. Blugel, "Real-Space Imaging of Two-Dimensional Antiferromagnetism on the Atomic Scale," Science Magazine, Vol. 288, Issue 5472, pp. 1805-1808, 9 June 2000.

[13] "MRAM-Info: Magnetic RAM news, forum, articles and more," http://www.mram-info.com/ (Accessed: October 2, 2006)

[14] "What is MEMS Technology," http://www.memsnet.org/mems/what-is.html (Accessed: October 2, 2006)

[15] "73.4GB 3.6MS/15000 (ULTRA 320 80PIN) 8192K 3.5"/HH," http://www.spartantech.com/product.asp?PID=ST373453LC&m1=pg (Accessed: October 2, 2006)

[16] "Seagate Barracuda 7200.8 400GB 3.5" IDE Ultra ATA100 Hard Drive – OEM," http://www.newegg.com/Product/Product.asp?Item=N82E16822148060 (Accessed: October 2, 2006)

[17] "Cheetah 15K.3 - ST336753LC," http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,552,00.html (Accessed: October 2, 2006)

[18] James S. Plank, Youngbae Kim, and Jack Dongarra, "Fault Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing," *Journal of Parallel and Distributed Computing*, Vol. 43, No. 2, pp. 125-138, September 1997.

[19] "Los Alamos/Liv 3D Simulations," Publication of Los Alamos National Laboratory, Vol. 3, No. 6, April 4, 2002.

[20] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, Vol. 17, No. 2-4, pp. 323-356, February-April, 2005.

[21] J. Plank, M. Beck, G. Kingsley, and K.Li, "Libckpt: Transparent Checkpointing under Unix," *Usenix Winter 1995 Technical Conference*, New Orleans, LA, January, 1995.

[22] G. P. Kavanaugh and W. H. Sanders, "Performance analysis of two time-based coordinated checkpointing protocols," *Pacific Rim International Symposium on Fault-Tolerant Systems*, Taipei, Taiwan, December 15-16, 1997.

[23] N. H. Vaidya, "A case for two-level distributed recovery schemes," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Ottawa, May 1995.

[24] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, Vol. 46, No. 8, pp. 942–947, August 1997.

[25] K. F. Wong and M. Franklin, "Checkpointing in distributed systems," *Journal of Parallel & Distributed Systems*, Vol. 35, No. 1, pp. 67–75, May 1996.

[26] "Fixed Point Iteration," http://pathfinder.scar.utoronto.ca/~dyer/csca57/book_P/node34.html (Accessed October 2, 2006).

[27] D. F. Stanat and S. F. Weiss, "Systematic Programming," *Online book resources*, http://www.cs.unc.edu/~weiss/COMP114/BOOK/BookChapters.html (Accessed: October 2, 2006)

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

[29] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, Vol. 34, No.2, pp. 171-210, 2000.

[30] R. Enzler, C. Plessl and M. Platzner, "System-level Performance Evaluation of Reconfigurable Processors," *Journal of Microprocessors and Microsystems*, Vol.29, pp. 63–73, 2005.

[31] Nallatech, "BenNUEY Reference Guide," Hardware Reference Guide, Glasgow, United Kingdom, 2004.

[32] Celoxica, Ltd., "RC1000 Hardware Reference Manual," Hardware Reference Manual, Oxfordshire, United Kingdom, 2001.

[33] Tarari, Inc., "High-Performance Computing Hardware Reference," Hardware Reference Manual, San Diego, CA, 2004.

[34] Silicon Graphics, Inc., "Reconfigurable Application-Specific Computing User's Guide," User's Guide, Mountain View, CA, 2004.

[35] J. Vetter and A. Yoo, "An Empirical Performance Evaluation of Scalable Scientific Applications," *Proc. of Supercomputing 2002*, Baltimore, MD, Nov. 16-22, 2002.

[36] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R. Woodward, L.J. Shieh, and S.W. White, "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," *Proc. of ACM/IEE conference on Supercomputing, High Performance Networking and Computing Conference*, Portland, OR, Nov. 13-19, 1999.

[37] P.N. Brown, R.D. Falgout, and J.E. Jones, "Semicoarsening Multigrid on Distributed Memory Machines," *SIAM Journal on Scientific Computing*, Vol. 21, No. 5, pp. 1823-1834, 2000.

[38] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," *Proc. of International Conference on Parallel Processing*, Toronto, Canada, August 21-24, 2000.

[39] W.K. Anderson, W.D. Gropp, D. K. Kaushik, D.E. Keyes, and B.F. Smith, "Achieving High Sustained Performance in an Unstructured Mesh CFD Application," *Proc. of ACM/IEE conference on Supercomputing, High Performance Networking and Computing Conference*, Portland, OR, Nov. 13-19, 1999.

[40] R.S. Tuminaro, S.A. Hutchinson, and J.N. Shadid, "The Aztec Iterative Package," *International Linear Algebra Iterative Workshop*, Toulouse, France, June 10-13, 1996.

[41] M. Smith and G. Peterson, "Analytical Modeling for High Performance Reconfigurable Computers," *Proc. SCS International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS)*, San Diego, CA, July 14-19, 2002.

[42] 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. of Eight Heterogeneous Computing Workshop*, San Juan, Puerto Rico, April 12, 1999.

[43] S. Vetter, S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo, "RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide, " IBM, 1998.

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

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

[46] M. Dantas, "Evaluation of a Lightweight MPI Approach for Parallel Heterogeneous Workstation Cluster Environments," *Proceedings of the Euro-Par conference*, Southampton, United Kingdom, September 1-4, 1998, pp. 397-400.

[47] A. Agbaria, D. Kang and K. Singh, "LMPI: MPI for Heterogeneous Embedded Distributed Systems," *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS '06)*, Minneapolis, Minnesota, July 12-15, 2006, pp. 79-86.

[48] J. Kohout and A. George, "A High-Performance Communication Service for Parallel Computing on Distributed DSP systems," *Parallel Computing*, Vol. 29, No. 7, July 2003, pp. 851-878.

[49] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *In Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawai, pp. 526-531, 1996.

[50] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," *SuperComputing 2002*, Baltimore, USA, November 2002.

[51] Agbaria and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations," *In the 8th IEEE International Symposium on High Performance Distributed Computing, IEEE CS Press*, Los Alamitos, California, pp. 167-176, 1999.

[52] "Ensemble project web site", http://dsl.cs.technion.ac.il/projects/Ensemble/ (Accessed: October 2, 2006).

[53] S. Rao, L. Alvisi and H. Vin, "Egida: An Extensible Toolkit for Low-overhead Fault-Tolerance," *In Proceedings of IEEE International Conference on Fault-Tolerant Computing (FTCS)*, pp. 48-55, 1999.

[54] "MPICH project web site", http://www-unix.mcs.anl.gov/mpi/mpich/ (Accessed: October 2, 2006).

[55] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "Mpi-ft: Portable fault tolerance scheme for mpi," *In ParallelProcessing Letters, World Scientific Publishing Company*, Vol. 10, No. 4, pp. 371-382, 2000.

[56] "LAM/MPI project web site", http://www.lam-mpi.org/ (Accessed: October 2, 2006).

[57] G. Fagg and J. Dongarra, “FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World,” *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, Hungary, Vol. 1908, pp. 346-353, 2000.

[58] T. Kaiser, L. Brieger, and S. Healy, “MYMPI – MPI Programming in Python, ” *Proceedings of the International Conference on Parallel and Distributed Processing and Applications*, Las Vegas, Nevada, June 26-29, 2006.

[59] “Super LU”, http://crd.lbl.gov/~xiaoye/SuperLU/#superlu_dist (Accessed: October 2, 2006).

BIOGRAPHICAL SKETCH

Rajagopal Subramaniyan was born in Tanjore, India. He completed his schooling at Carmel Garden matriculation higher secondary school, Coimbatore, India. He went on for his bachelor's degree in electronics and communication engineering with a scholarship at Anna University, Chennai, which is one of the most prestigious engineering institutions in India. He successfully completed his bachelor's degree with distinction and joined University of Florida to pursue his master's in electrical and computer engineering. In Spring 2001, he joined the High-performance Computation and Simulation (HCS) Research Laboratory as a graduate research assistant under the guidance of Dr. Alan George. He completed his master's degree in December 2002 and stayed back at University of Florida to pursue his Ph.D. in computer engineering. His research at the HCS Research Lab forms the basis of this work. He will complete his Ph.D. in December 2006 and plans to take up a career in the computer engineering field.