Modeling Coordinated Checkpointing for Large-Scale Supercomputers

Long Wang, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801, USA
{longwang, pattabir, kalbar, iyer}@crhc.uiuc.edu

Lawrence Votta, Christopher Vick, Alan Wood
Sun Microsystems
Menlo Park, CA 94025, USA
{lawrence.votta, christopher.vick, alan.wood}@sun.com

Abstract. *Current supercomputing systems consisting of thousands of nodes cannot meet the demands of emerging high-performance scientific applications. As a result, a new generation of supercomputing systems consisting of hundreds of thousands of nodes is being proposed. However, these systems are likely to experience far more frequent failures than today's systems, and such failures must be tackled effectively. Coordinated checkpointing is a common technique to deal with failures in supercomputers. This paper presents a model of a coordinated checkpointing protocol for large-scale supercomputers, and studies its scalability by considering both the coordination overhead and the effect of failures. Unlike most of the existing checkpointing models, the proposed model takes into account failures during checkpointing and recovery, as well as correlated failures. Stochastic Activity Networks (SANs) are used to model the system, and the model is simulated to study the scalability, reliability, and performance of the system.*

1. Introduction

The computational demands of emerging applications, such as protein folding, is giving rise to a new generation of supercomputers (currently in the planning stage) consisting of several thousand processors. For example, the newly deployed IBM BlueGene/L [1] is expected to scale to 64K dual-processor nodes. Despite the huge computing power these systems provide, the large number of nodes makes them significantly more vulnerable to errors. The resulting larger number of failures due to errors can impair system performance and limit scalability.

Although a hierarchy of error detection and recovery techniques, such as ECC, CRC, and message retransmission, can correct some errors/failures, some transient errors/failures cannot be covered using these techniques, e.g. corrupted states due to propagation of undetected errors. For these errors/failures, checkpointing and rollback may be to recover the application before rebooting or reconfiguring the system. This paper focuses on errors/failures that need checkpointing and rollback to recover.

The most commonly used checkpointing scheme for supercomputing systems is coordinated checkpointing, due to its simplicity of implementation. In this approach, cooperative processors synchronize to ensure a global consistent state before taking a checkpoint [3]. The main problem with coordinated checkpointing is its lack of scalability, as it requires all processors to take a checkpoint simultaneously.

This paper makes two main contributions. First, it builds a model of a large-scale system that uses coordinated checkpointing for recovery from failures with complex semantics. Second, it studies the scalability and performance of the system for several hundred thousand processors by simulating the model with realistic parameter values.

An important issue considered in our model is the effect of scaling from several thousand processors to several hundred thousand processors, i.e., by two orders of magnitude. Issues such as failures during checkpointing and recovery, correlated failures within the system, and checkpointing overhead due to coordination are of primary importance for the new generation of supercomputers. This is because their larger number of nodes and higher failure rates invalidate some assumptions that existing models make about system behavior [7, 8, 9, 10, 11, 12] and exacerbate some effects previously considered negligible. These assumptions are:

- The computation interval and the checkpoint overhead are much smaller than the mean time between failures (MTBF). However, large-scale supercomputers experience much smaller MTBFs and much larger checkpoint overheads, and hence failures during checkpointing and recovery can occur and must be taken into account [5].
- Failures are independent of each other. This is not a valid assumption, as Tang and Iyer [6] showed that even a small number of correlated failures increase system unavailability considerably.
- The overhead of inter-processor coordination for checkpointing is negligible. However, as the number of nodes increases, the coordination overhead grows, and it cannot be ignored.

A measure called *useful work* similar to *accumulated reward* [17] is used to evaluate system performance. Useful work is defined as the computation that contributes to the ultimate completion of the job (see definition in Section 7). If a failure occurs before the computation can be checkpointed, the computation since the last checkpoint needs to be repeated after the recovery and is not counted as useful work. Accurate modeling of useful work requires knowledge on future behavior of the system and cannot be represented using simple Markov models. Instead, Stochastic Activity Networks (SANs) are used to model the system behavior. The modeling power of SANs allows us to concisely represent complex system phenomena such as checkpoint coordination, failures during checkpointing and recovery, and correlated failures. The SAN model is studied using simulation, and the impact of system parameters on system performance and scalability is evaluated.

2. Related Work

Checkpointing models. One of the earliest models for computing the optimal checkpointing interval is by Young

0-7695-2282-3/05 $20.00 © 2005 **IEEE**

[7]. This model assumes that the MTBF of the system is very large compared to the checkpoint and recovery time, and hence it does not consider failures during checkpointing and recovery. Daly [8] presents a modification of Young's model for large-scale systems. This model takes into account failures during checkpointing and recovery as well as multiple failures in a single computation interval. However, it does not model the coordination overhead of the checkpointing protocol itself or consider correlated failures.

Kavanagh and Sanders [9] evaluate two time-based coordinated checkpointing protocols based on analytical and simulation models, which take the overhead of coordination into account. However they do not consider failures during checkpointing and recovery, as they assume that the MTBF of the system is much greater than the checkpoint interval.

Plank and Thomason [10] investigate the use of spare nodes to provide redundancy in the system to handle permanent failures. We do not consider permanent failures in our model and assume that all nodes can be recovered by restarting the system from the last-saved checkpoint. Plank and Thomason do not consider the overhead of coordination in their model or the effect of scaling the model to a large number of nodes. A recent paper by Elnozahy et al. [11] extends the work of Plank and Thomason to systems consisting of thousands of nodes. It considers the effects of failures during checkpoint and recovery and multiple failures in a single computation interval. However, it does not consider the effects of coordination among the nodes in the checkpointing protocol, nor does it consider correlated failures.

Vaidya [12] derives an analytical expression for the optimal checkpointing frequency in a uniprocessor system. It distinguishes the checkpoint latency from the overhead of a checkpointing scheme. This model considers failures during checkpointing/recovery but does not take into account the scalability of the checkpointing protocol or the system.

Large-scale systems. Bronevetsky et al. [23] present a compiler-based technique for asynchronous, coordinated checkpointing. Agarwal et al. [24] consider an adaptive, incremental checkpointing technique for scientific applications on large-scale systems. Finally, Zhang et al. [18] do an extensive study of failure data analysis in large-scale supercomputing systems and show the existence of temporal and spatial correlation among failures in large-scale systems. We consider temporal correlations in our model (correlated failures), but not spatial correlations.

3. Target System

This study focuses on a typical abstract structure commonly shared by many supercomputers and a basic coordinated checkpointing protocol whose variants are applied in the supercomputing world.

3.1 Architecture

Each node of the supercomputing system is a tightly integrated unit consisting of multiple processors. For example, Blue-Gene/L has 2 processors per node, and ASCI Q has 4 processors per node. Future systems could have 8, 16, or 32 processors per node.

Usually, large-scale supercomputing systems have dedicated nodes for job computation (compute nodes) and for I/O operations (I/O nodes). The compute nodes in a set share the connections to an I/O node, and all the I/O nodes are connected to a parallel file system through a separate connection network. For example, IBM BG/L has 64K compute nodes and 1024 I/O nodes. The network bandwidth from 64 compute nodes to one I/O node is 350MB/s, and the bandwidth from one I/O node to the file system is 1 Gb/s.

Data writes from compute nodes to the file system are performed in two steps: from compute nodes to I/O nodes and then from I/O nodes to the file system. The I/O nodes locally buffer the application data or checkpoint they receive from the compute nodes and then write it to the file system in the background while the compute nodes continue with the computation. The two steps are reversed for data reads with the exception that reads cannot be done in the background, as the application may have to wait for the data to be read before proceeding, depending on the nature of the read.[1]

3.2 Checkpoint Protocol

There are two checkpointing approaches used in supercomputing systems. One is application-based, where a global barrier is explicitly used in the application for saving a global consistent state. This places the burden of checkpointing on the application (e.g., in BlueGene/L [1]). The other approach is system-supported checkpointing (e.g., the algorithm used by Cray in the IRIX OS [19]). Our checkpointing protocol is a system-supported synchronous checkpointing and follows the basic principles of coordinated checkpointing, e.g., Koo and Toueg's protocol [4].

In our protocol, a single coordinator node, or master, periodically initiates the checkpointing as follows:

(1) *The master broadcasts a 'quiesce' request to all the compute nodes.*
(2) *On receiving 'quiesce' each node quiesces its operations, i.e., stops all its activities at a consistent and interruptible state and replies 'ready' to the master.*
(3) *After receiving 'ready' from all the compute nodes, the master broadcasts 'checkpoint' to all the compute nodes.*
(4) *On receiving 'checkpoint' each compute node dumps its state to an I/O node, and then sends a 'done' message to the master.*
(5) *When the master collects the 'done' messages from all the compute nodes, it broadcasts 'proceed' to all the compute nodes, and the I/O nodes begin to write the checkpoint to the file system in the background.*
(6) *On receiving 'proceed' each compute node continues its activity from the point at which it quiesced.*

When a node is quiesced, it means that it stops all the task-related activities in a consistent and interruptible state. Further, a timeout period is specified at the master to avoid waiting indefinitely for the 'ready' responses. This indefinite wait can occur as a result of an erroneous or failed node that does not respond to the quiesce request. If all the responses are not received within this time, the master times out and broadcasts an 'abort' message to all the compute nodes, causing them to abandon the checkpointing and proceed with their computations.

Note that the current checkpoint does not overwrite the previous checkpoint, unless the checkpointing successfully

[1] While current supercomputing systems may not have this capability, future systems might allow this two-step I/O.

0-7695-2282-3/05 $20.00 © 2005 IEEE

completes and the checkpoint is verified to be correct. So whenever the checkpointing is abandoned the previous checkpoint is still valid. Hence, the system can always recover to the last good checkpoint upon a compute node failure.

3.3 Application

The application is a parallel, scientific computing workload composed of multiple computation tasks. Each compute processor runs exactly one task of the parallel application and no other tasks.

Application tasks may be performing computation, communication, or I/O at any time. Since most parallel, scientific applications are written using the BSP (Bulk Synchronous Parallel) model [13], the multiple tasks more or less coordinate their actions and behave as one cohesive unit.

The application is instrumented with a number of checkpoint primitives at its safe points (e.g. a global barrier), where it can safely quiesce, as at the end of a loop. For example, in IRIX, the programmer inserts checkpoint functions in the source code, and the OS calls these whenever it wants to take a checkpoint.

A task that is doing an I/O write, cannot quiesce until it finishes the I/O operation, as this could leave the I/O in an inconsistent state and possibly corrupt the file system. While there are methods to address this, ensuring global coordination is complicated, and the simple approach of non-preemptive I/O is preferred in practice. I/O reads of a task can be stopped for checkpointing at any time, and hence, they are not specifically considered in our model.

3.4 Failure and Recovery

On the failure of a compute node, the entire application rolls back to the last saved checkpoint and recovers, i.e., we only consider failures that require recovery from a checkpoint. While permanent/persistent errors are not considered in the paper, checkpointing can still be used to recover from permanent hardware failures. This, however, would require system reconfiguration and remapping of the checkpointed states into a new set of nodes (assuming that spare nodes are available).

Failures of compute nodes and I/O nodes are always detected without any latency. The mechanism for failure detection is not modeled.

When an I/O node fails, all the I/O nodes need to be restarted. This assumption is reasonable, since in the BSP model, the application needs the I/O operations on all the I/O nodes to be completed before continuing the computation.

When the master node fails when checkpointing is not in progress, we assume that the error is detected and the master recovers independently of the other nodes. If the master fails during checkpointing, the checkpointing protocol is aborted and the master goes back to the initial state.

As nodes have multiple processors, the node failure rate is the product of the processor failure rate and the number of processors per node. The system parameter MTTF is used to refer to the per-node mean time to failure throughout this paper unless specified otherwise. Then per-processor MTTF is MTTF times the number of processors per node. It is assumed that advanced design and error handling techniques are applied to maintain low node failure rates, e.g., use of multiple cores on a chip.

As there is no consensus on MTTF in the literature, we assume an MTTF value from 1 year to 25 years due to both hardware and software errors based on the following: (i) ASCI-Q has a per-node MTTF of 1 year [11], (ii) IBM 380 X processor has an MTTF of 8 years [16], (iii) IBM mainframes have an MTTF of 25 years, and (iv) IBM G5 processor is advertised with an MTTF of 45 years [22] (hardware failures only).

3.5 Correlated Failure

This paper models two categories of correlated failures: (i) correlated failures due to error propagation only and (ii) generic correlated failures.

For correlated failures due to error propagation, we assume that recovery fully restores the application/system state and that propagated errors do not cross recovery boundaries. The error propagation is characterized by a short error burst, which typically impacts the recovery. The duration of the error burst is referred to as the *correlated failure window*. The system may need to recover several times before a successful recovery [20]. A typical value of the correlated failure rate is 600 times the normal failure rate [6] (see Section 6).

Correlated failures may be caused by factors other than error propagation, e.g., common causes such as increases in node temperature or some environmental phenomena. Usually, a hyper-exponential distribution is assumed for modeling generic correlated failures, i.e., the system experiences an independent failure rate and a correlated failure rate alternatively. Unlike correlated failures due to propagation, the semantics of generic correlated failures is not necessarily limited to a short duration, but rather forms a global view of the system for the entire system life.

4. Overall Composition of the Model

The system is decomposed into several subsystems. Each subsystem is modeled as a separate Stochastic Activity Network (SAN) submodel, and the overall model is obtained by integrating these submodels. All the compute nodes are modeled as a single unit and all the I/O nodes are modeled as another unit. This allows the model to scale to a large number of nodes without requiring a large simulation time. Table 1 lists the SAN submodels of the entire system, and Figure 1 illustrates how these submodels (the ovals in Figure 1) are integrated into an overall model. The arrows in the figure illustrate the logical interactions between the submodels. These interactions are implemented by state sharing between the submodels. The dots in the submodels in Figure 1 indicate the initial position of the tokens in the corresponding SAN. It should be emphasized that Figure 1 is not a state diagram,in that the ovals do not represent the states of the system at any particular time. The submodels are organized into four modules: *computing & checkpointing, failure and recovery, correlated failure,* and *useful work computation*.

Computing and checkpointing module. The *compute_nodes* submodel depicts the computation and checkpointing behavior of the compute nodes in the failure-free mode. While the compute nodes are in execution, the applica-

tion may be performing either computation or I/O operations, and this is represented in the *app_workload* submodel. The *master* submodel represents the master node in the coordinated checkpointing protocol. It triggers and coordinates the checkpointing, as modeled in the *compute_nodes* submodel. The coordination among the compute nodes is modeled in the *coordination* submodel. The *io_nodes* submodel captures the I/O operations conducted by I/O nodes. It receives data from the *compute_nodes* submodel, writes/reads checkpoints to/from the file-system, and writes data on behalf of the application in the *app_workload* submodel. These five submodels form the *computing and checkpointing* module of the system model and are further described in Section 5.

Table 1: Submodel List

Module	Submodel	Comments
Computing & checkpointing	app_workload	Application state: performing computation or I/O operations
	compute_nodes	Compute processor state in the checkpoint cycle: executing (including both application's computation and I/O operations), quiescing, or checkpoint dumping
	coordination	Coordination procedure for checkpointing
	io_nodes	I/O processor state: idling (including data transmission between compute nodes), writing application data, writing checkpoint, or reading checkpoint; if checkpoint is locally buffered
	master	System checkpointing state: if checkpointing is started or not
Failure & recovery	comp_node_failure	Failure behavior of compute nodes
	comp_node_recovery	Recovery behavior of compute nodes
	io_node_failure	Failure behavior of I/O nodes
	io_node_recovery	Recovery behavior of I/O nodes
	system_reboot	System reboot operation
Correlated failure	correlated_failures	Correlated failure behavior
Useful work	useful_work	Useful work computation

Failure and recovery module. A compute node or I/O node may fail in any of its states. The occurrence of failures in compute nodes is modeled in the *comp_node_failure* submodel. Recovery is initiated following the detection of the failure and modeled in the *comp_node_recovery* submodel. As failures may also occur during recovery, compute nodes may experience multiple failures and subsequent recoveries in the *comp_node_recovery* submodel before the final successful recovery, after which the system resumes the normal execution and checkpointing cycle. Failures of compute nodes do not affect the I/O nodes if error propagation is not considered. The behavior of I/O nodes is similar, except that when an I/O node fails while writing application data to the file system, the application results are lost and the system rolls back to the last checkpoint. This is represented in Figure 1 by an arrow from the *io_node_failure* submodel to the *comp_node_failure* submodel.

The recovery process occurs in two stages. First, the I/O nodes read the checkpoint from the file system and buffer it in their local memories. Then the compute nodes read the checkpoint from the I/O nodes and complete the recovery. The compute nodes then go back to the execution state, the master process gets reset, and the system exits the correlated failure window if there was one. If the checkpoint is already locally buffered in the I/O nodes when a compute node fails, the first stage is skipped. If an I/O node fails while writing out a checkpoint, the checkpoint is aborted and the I/O nodes get restarted, but the compute nodes are not affected.

If the number of unsuccessful recoveries in the *comp_node_recovery* and/or *io_node_recovery* submodel(s) exceeds a predefined threshold, the whole system, including the compute nodes and I/O nodes, is rebooted in *system_reboot* ("severe failures" transitions from *comp_node_recovery* and *io_node_recovery* to *system_reboot* in Figure 1). When the reboot completes, I/O processors are ready for execution, but compute nodes still need to read the last checkpoint and recover. So the arrows of "reboot completes" from the *system_reboot* submodel point to the *io_nodes* and *comp_node_failure* submodels, instead of the *compute_nodes* submodel in Figure 1.

Figure 1: The overall composition of the model

Correlated failure module. The *correlated_failures* submodel models the semantics of correlated failures separately from the compute and I/O nodes' failure and recovery submodels. It controls the rates of all failures in the system. When a correlated failure occurs, the system enters a correlated failure window, in which it experiences failures with a higher rate than the independent failure rate. Note that independent failures can continue to occur when the system is within a correlated failure window.

Useful work module. The *useful_work* submodel calculates the useful work completed by the system. A positive reward is accumulated when the compute nodes perform job computation or I/O operations, and a negative reward equal to the amount of the lost work is applied when a compute node fails.

5. Modeling Computing and Coordinated Checkpointing

In this section, we describe the details of modeling the *computing and coordinated checkpointing* module using SANs. Due to space limitations, detailed SAN models of the

0-7695-2282-3/05 $20.00 © 2005 IEEE

other three modules are not described in this paper. The reader may refer to the technical report [25] for these.

Figure 2 shows the SAN submodels for the *computing and coordinated checkpointing* module. States are shared among the submodels with the same names. Selected shared states are numbered in Figure 2 to help identify them.

When the application is started in the system, the compute nodes start out in the *execution* state and the master is in the *master_sleep* state. We assume the application starts doing computation and the *app_workload* is in the *compute* state. The I/O nodes are in the *ionode_idle* state. Initially, each of these states has a token, indicated by block arrows in Figure 2. In our model, the non-random events are modeled as deterministic activities, and exponential distribution is assumed for random events. To simplify the model, message transmissions are not explicitly modeled in SAN, but the parameters of the corresponding events are appropriately set to include the message transmission latency. Also, the 'done' and 'proceed' message exchanges are not modeled in the interest of simplicity. The following steps detail the behavior of the model.

- First, assume that the checkpoint interval expires and the *checkpoint* activity is enabled. The master moves from the *master_sleep* state to the *master_checkpointing* state and starts a timer as shown by the *start_timer* gate. (Figure 2d)
- The compute nodes are initially in the state *execution.* When the master moves to *master_checkpointing,* the compute nodes move to the *quiescing* state after a latency of *recv_quiesce_bcast _time* (broadcast overhead). (Figure 2a)
- Henceforth, the behavior depends on whether the application workload is performing computation or I/O. If the *app_workload* is in the *compute* state, the coordination for checkpointing is started, as shown in the *to_coordination* activity. If the *app_workload* is in the *IO* state, the compute nodes wait till the I/O completes before starting the coordination activity. (Figure 2c)
- After the coordination activity (*coord)* completes, a token is placed in the *complete_coordination* state, enabling the activity *coordinate* in *compute_nodes,* and the compute nodes move from *quiescing* to *checkpointing*. (Figure 2e, 2a)
- If the timer expires before the coordination is complete, it places a token in the *timedout* state. This activates the *skip_chkpt2* activity in *compute_nodes,* causing the compute nodes to abort the checkpointing and move to the *back_to_execution* state. (Figure 2d, 2a, 2e)
- When the compute node is in the state *checkpointing* and the I/O node is in the state *ionode_idle*, the *dump_chkpt* activity is enabled. The checkpoint dump time depends on the checkpoint size and the bandwidth between the compute nodes and the I/O nodes. (Figure 2a)
- After storing the checkpoint, the compute nodes go back to the *execution* state. The completion of this activity also places tokens in the *enable_chkpt* state. (Figure 2a)
- When the I/O node is in *ionode_idle,* it sees the token in the *enable_chkpt* state and goes to the *writing_chkpt* state. This enables the *write_chkpt* activity, which models the writing of the checkpoint to the file system. The latency of the write depends on the checkpoint size and the bandwidth between the I/O node and the file system. (Figure 2b)

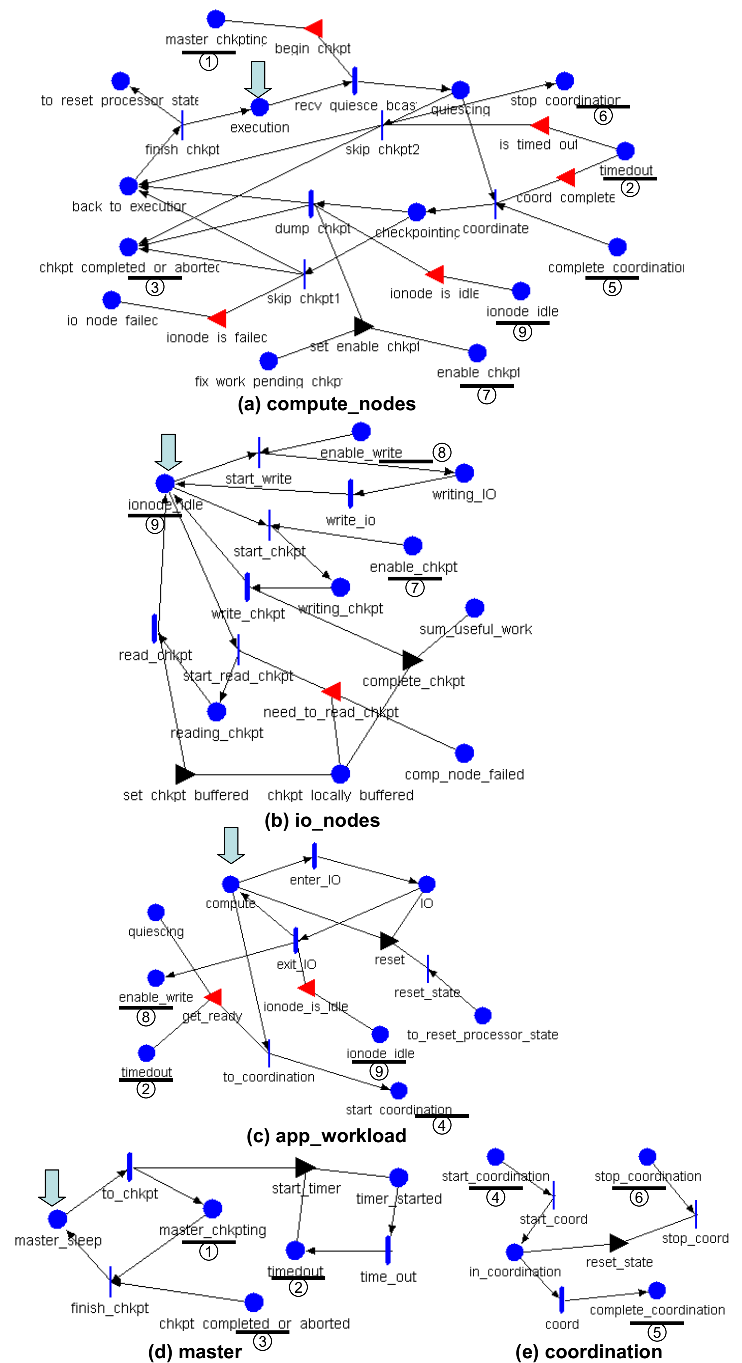

Figure 2: Submodels for computing and checkpointing

- If the I/O node is not in *ionode_idle*, the compute node has to wait for the I/O node to come to the *ionode_idle* state before sending the checkpoint to it. This prerequisite is enforced by the *ionode_is_idle* input gate. (Figure 2a)
- When the checkpointing is completed or aborted, tokens are placed in the two states *chkpt_completed_or_aborted* and *to_reset_processor_state*. The tokens cause the master to move back to the *master_sleep* state and the *app_workload* to reset at the *compute* state. (Figure 2c)

Since the model considers all the compute nodes as a single unit, it does not reflect the discrepancy in the quiesce times among the compute nodes and does not show how the variation in the quiesce time among the nodes can cause the

master to timeout. This behavior is modeled separately in the *coordination* submodel (Figure 2e). It is assumed that each node has an identical, exponentially distributed quiesce time with the mean of MTTQ. We use a random variable Y, representing the maximum of all the quiesce times, to model the coordination time as follows:

Let n and X_i denote the number of compute nodes and the ith node's quiesce time, respectively, and $Y = max\{X_i\}$ $(1 \le i \le n)$. Then, the CDF of Y is $F_Y(y)=(F_X(y))^n=(1-e^{-\lambda y})^n$ where λ is the quiesce rate of a single compute node. Y can be generated from a uniform random variable U between 0 and 1 by $Y = -1/\lambda \cdot log(1-U^{1/n})$. The value of Y is used as the latency in the *coord* activity in the *coordination* submodel to represent the coordination process.

6. Modeling Correlated Failures

Two categories of correlated failures are modeled in the paper: (i) correlated failures due to error propagation and (ii) generic correlated failures. Both are modeled by appropriately increasing the node/processor failure rates. This section describes how these increased rates are derived.

Correlated failures due to error propagation. When an independent failure occurs in the system, with some probability p_e there is a conditional probability of a second failure due to the first. This results in an increased failure rate. We compute this failure rate increase for all nodes by multiplying the independent failure rate with a constant parameter called *frate_correlated_factor*.

Figure 3 shows the birth-death Markov process of correlated failures due to error propagation. λ_i and λ_c denote the rates of the system-wide independent failures and successive correlated failures, respectively. λ is the independent failure rate of a single node. μ denotes the recovery rate of the system. F_i is the system state in which i failures have occurred before a successful recovery (three states, F_0, F_1 and F_2, are shown as examples in Figure 3). As we assume that any successful recovery wipes off all latent errors, all the F_i states transit directly to F_0 with the recovery rate. It is also assumed that the failure rates at all the F_i states (i > 0) are the same. So, the conditional probability of another failure occurrence provided that a failure occurs is,

$$p= \lambda_c/(\lambda_c +\mu) \quad => \quad \lambda_c=p\mu/(1-p).$$

Let n denote the number of nodes, and r denote the multiple *frate_correlated_factor*. Then according to the model,

$$\lambda_c=\lambda_i+rn\lambda=n\lambda(1+r) \quad => \quad r=p\mu/((1-p)n\lambda)-1.$$

For a given set of n, λ and μ, r, i.e. *frate_correlated_factor,* actually represents the conditional probability p. As long as $\lambda_c>\lambda_i$, r can be chosen independently to study a range of correlated failure effects. For example, when $n=1024$, $p=0.3$, $MTTR=10min$, and $MTTF=25yrs$, r is about 600.

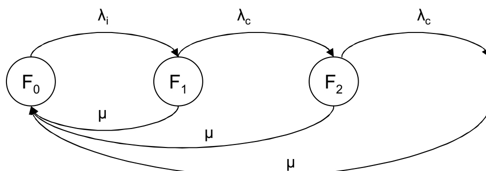

Figure 3: Birth-death Markov process of correlated failures

Generic correlated failures. The system may suffer from generic correlated failures at any instant of the system life. A correlated failure coefficient ρ is assumed to model generic correlated failures, which is the unconditional probability of a correlated failure occurring at any time. Table 2 lists the parameters used for modeling generic correlated failures. Then, the failure rate of generic correlated failures is given by,

$$\lambda_s = \lambda_{si} + \rho\lambda_{sc} = n\lambda + \rho rn\lambda = n\lambda(1 + \rho r).$$

Note that λ_{si}, λ_{sc}, and ρ are not the same as the λ_i, λ_c, and ρ_e in the discussion of correlated failures due to error propagation, because they model different probabilities. The symbols n, λ and r have the same meanings in both models.

Table 2: Parameters for modeling generic correlated failures

λ_s	Failure rate of the entire system
λ_{si}	Rate of independent failures in the system
λ_{sc}	Rate of correlated failures in the system
λ	Independent failure per node
r	Increased failure rate due to correlated failures
ρ	Correlated failure coefficient
n	Number of nodes

7. Experimental Setup and Results

We use the Mobius modeling environment [21] to create and simulate the SANs. Steady-state simulation is used with an initial transient period of 1000 hours to allow the system to enter the steady state. The confidence level is 95%. Unless otherwise specified, the parameter values are as in Table 3. These parameters are based on field data or projections of future systems.

As the modeled system is complicated and multiple mechanisms/parameters are present, we study the system by analyzing the effect of one feature at a time. Hence, the base model without coordination or correlated failures (but with failures during checkpointing and recovery) is first studied to understand the basic system behavior. Then we study the effects of coordination and correlated failures. The following two metrics are used to evaluate system performance.

- *Useful work fraction:* Fraction of time the system makes forward progress towards the completion of the job. It does not include work that is repeated due to failures.
- *Total useful work:* The product of the useful work fraction and the number of compute processors. It indicates how many processors of the same kind are required to achieve the same performance, assuming failure-free computation.

7.1 Study of Base Model

For the base model, we assume independent failures and consider the coordination time to be a fixed quiesce time. The system performance is analyzed for a range of parameters, including the number of processors, checkpoint interval, MTTF per node, and MTTR of the system, as follows:

- Number of processors per node: 8
- MTTF per node: 1 year
- MTTR of the system: 10 minutes[2]
- Number of processors: 64K
- Checkpoint interval: varied from 15 minutes to 4 hours

[2] If permanent failures are considered, the overhead of the system reconfiguration will result in a larger MTTR.

Table 3: Model Parameters

Parameter	Value /Range	Comments
Checkpoint interval	15 min to 4 hr	Derived from other studies [1], and private communication with vendors
MTTF (Mean Time To Failure per node)	1 – 25 yr	Including software and hardware failures recovered from checkpoint. 1 year for ASCI Q and 25 years for IBM mainframes
MTTR (system-wide Mean Time To Recovery of compute nodes)	10 min	Average time for all compute nodes to read checkpoint and reinitialize themselves
MTTR of IO nodes	1 min	Time to restart the I/O nodes
Number of compute processors	8K to 256K	Projection of current and future supercomputers
MTTQ (per-node Mean Time to Quiesce)	0.5-10 s	Time to close I/O and network file handles, clean up states, and perform computation until reaching a safe point
Broadcast overhead	1 ms	E.g. data for hardware broadcast trees in Blue-Gene/L [1]
Software overhead for transmission	1 ms	Measurement of message latency in TCP/IP and UDP
Period of I/O – compute cycle in application	3 min	Experimental data on I/O characteristics of parallel applications [15]
Fraction of computation	0.88 – 1.0	Experimental data on I/O characteristics of parallel applications [1]
Timeout value	20 sec to 2 min	The period for the master to timeout and cancel the checkpointing
Probability of correlated failure	0 to 0.2	Experimental data on correlated failures, e.g., [6]
Correlated failure rate	1/MTTF* (100~1600)	Projections on error propagation within a locally-federated cluster of nodes in the supercomputer
Correlated failure window	3 min	Experimental data for persistence of correlated failures in the system due to error propagation
System reboot time	1 hr	Anecdotal evidence for startup time of a large cluster
Aggregate bandwidth between compute nodes and one I/O node	350 MBps	E.g., Blue-Gene/L field data [1]
Number of compute nodes per I/O node	64	
File system bandwidth per I/O node	1 Gbps	
Checkpoint size per node	256 MB	
Average size of I/O data per node	10 MB	Experimental data on typical characteristics of parallel applications [15]

We report results for the number of processors, not the number of nodes, so that they can be easily scaled to a different number of processors per node. The major results are:

✧ For a given checkpoint interval (30 min), MTTR (10 min), and MTTF (1 yr per node), there is an optimum number of processors (128 K) for which total useful work done by the system is maximized. Adding more processors than this optimum value will hurt system performance due to failure effects[3]. The range of processors considered in this analysis is from 8K to 256K, and the optimum value of the number of processors varies from 128K to 32K as the MTTR varies from 10 minutes to 80 minutes.

[3] Elnozahy et al. [11] also make a conjecture that increasing the number of nodes beyond a certain extent hurts performance, but they do not quantify the extent of the performance loss.

✧ For the system to be scalable, checkpoints should be taken on the granularity of minutes (15-30 min), rather than hours, as is the current practice [26]. While in theory there is an optimal checkpoint interval, for any practical range there is no optimal checkpoint interval for which the useful work is maximized, contrary to what several other studies have shown [7, 8]. This is because the overhead of checkpointing is relatively low in our system, as the checkpoint writing is done in the background, and the effect of failures dominates the effect of taking checkpoints frequently.

✧ Even when the useful work is maximized, the overall useful work fraction is no more than 50% for an MTTF per node of 1 year. Hence, more than 50% of system resources are spent in checkpointing and recovering from failure.

✧ If the number of processors per node is increased to 32 from 8 and the per-node MTTF is maintained the same as 1 year, it is possible to increase the total useful work for the same number of nodes. This is because more compute power is provided per node, while maintaining the same failure rate. The optimum number of processors is in the range of 500K. However, the useful work fraction is unaltered, as the system failure rate, which depends only on the number of nodes and the per-node failure rate, is the same.

Variation of total useful work with number of processors. Figure 4a, c, and e show the variation of total useful work with different number of processors. In all the three figures, there is an optimum value of the number of processors for which total useful work is maximized. The rationale behind this is as follows: On one hand, more processors provide higher computing power for the job; on the other hand, more processors incur more frequent failures and hence more computation is wasted due to failures. For small numbers of processors, the former factor dominates, while for sufficiently large numbers of processors, the latter outweighs the former. Consider how the optimum number of processors varies with the MTTF, MTTR, and checkpoint interval.

- The optimum value decreases with smaller MTTFs, as shown in Figure 4a (from 128K processors for an MTTF of 1 year per node to 64K processors for an MTTF of 0.5 years per node).
- The optimum value decreases with larger MTTRs (from 128K processors for an MTTR of 20 minutes to 64K processors for an MTTR of 40 minutes), as shown in Figure 4c.
- The optimum value decreases with larger checkpoint intervals, as shown in Figure 4e (from 128K processors for a checkpoint interval of 30 minutes to 64K processors for a checkpoint interval of 60 minutes).

This is because, smaller MTTFs increase the failure rate, larger MTTRs increase the penalty of a failure, and the larger checkpoint intervals cause more work to be lost upon a failure. All the three aggravate the effects of failures, thus lowering the equilibrium point between the computing power and the failure effect.

Variation of total useful work with checkpoint intervals. Figure 4b, d, and f show the variation of total useful work for different checkpoint intervals. The results indicate that for a large-scale supercomputing system there is no optimum value

of the checkpoint interval within the range of values considered (15 minutes to 4 hours). This contradicts previous studies [7, 8], which have shown the existence of an optimum value of the checkpoint interval. This is because the loss of job computation due to failures in large-scale systems outweighs the overhead of frequent checkpointing, as our checkpoint overhead is low. The theoretical optimum value of the checkpointing interval is less than 15 minutes. However, checkpoint intervals less than 15 minutes are not considered because checkpoints as frequent as that may overwhelm the I/O subsystem and network and hence are not practical.

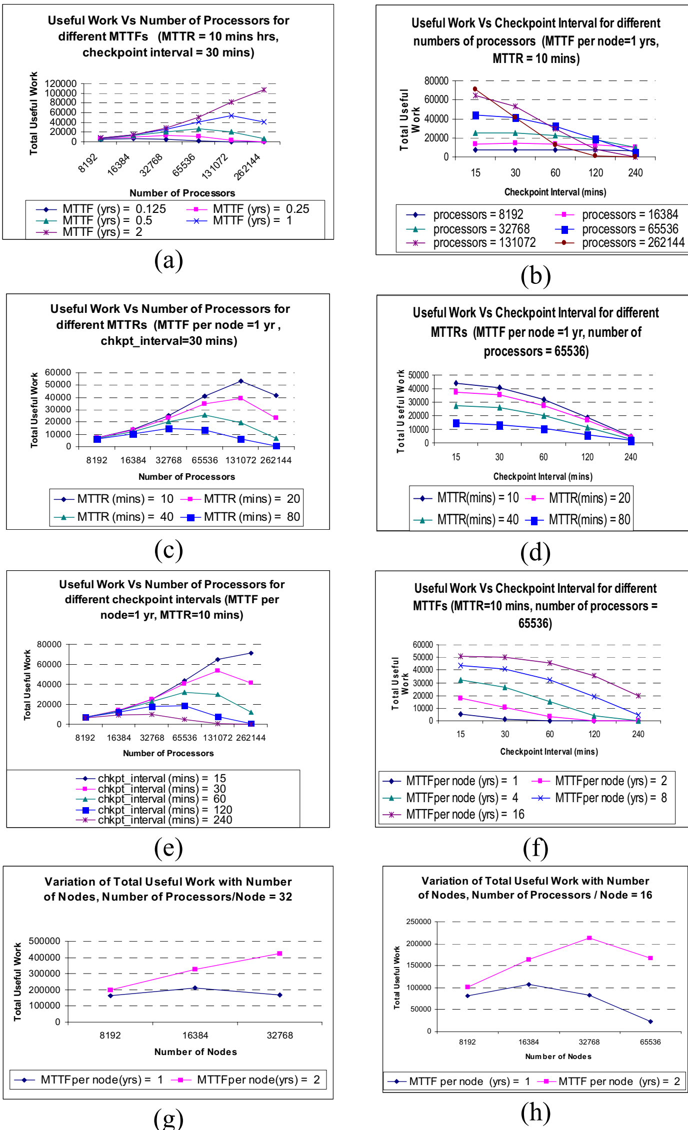

Figure 4: Sensitivity Study of the Base Model

Further, the total useful work is approximately constant for checkpoint intervals between 15 and 30 minutes but decreases sharply as the checkpoint interval is increased beyond 30 minutes. (For an MTTF of 8 years, the total useful work only decreases from 43000 job units[4] to 40000 job units when the checkpoint interval is increased from 15 minutes to 30 minutes, but it drops to 30000 job units when the checkpoint interval is increased to 60 minutes). This suggests that current checkpoint intervals in the granularity of hours are not appropriate for large-scale systems because of the high system failure rate. Rather, the checkpoint intervals should be between 15 and 30 minutes.

Useful work fraction. The discussion above only uses total useful work as the performance metric. The useful work fraction steadily decreases as the number of processors increases. This is because the greater number of processors does not contribute to the useful work fraction, and the failure effect degrades the useful work fraction. So, even when the maximum total useful work is achieved at the optimum number of processors, the useful work fraction is still small. For example, for an MTTF of 1 year per node in Figure 4a, the peak of total useful work is obtained with 128K processors, for which the useful work fraction is only about 56000/131072=42.7%, i.e., over 50% of system time is spent in handing failures. Thus, the overall failure rate of the system must substantially decrease for the useful work fraction to improve significantly.

Effect of increasing the number of processors per node. So far, we have assumed that each node has 8 processors and that the MTTF of a node is 1 year. In the future, advances in semiconductor and processor technology may allow 16 or 32 processor cores to be integrated on a single node while maintaining the same MTTF per node of 1 year. We studied the variation of total useful work with the number of nodes when each node has 32 and 16 processors, respectively for a per-node MTTF of 1 and 2 years. For a fair comparison, the number of processors is fixed at 1000K. The results are shown in Figure 4g and 4h and are summarized as follows:

- The optimum number of processors is obtained by multiplying the number of nodes by the number of processors per node. The optimum number of processors is now in the range of 500K to 1000K.
- For a given MTTF, the optimum number of nodes increases with the number of processors per node, as more compute power is provided at the same failure rate.
- For a given number of processors per node, the optimum number of nodes increases as the MTTF increases because the failure effect is less dominant

This reinforces the earlier observation that integrating more processors per node and maintaining the same node failure rate increases total useful work. However, the useful work fraction remains the same (still less than 50%), as it depends only on the system failure rate, which in turn depends only on the number of nodes and the MTTF per node.

Effect of failures during checkpointing/recovery. We also studied the effects of failures during checkpointing/recovery on system performance. We observed that they do not exert as significant an effect on the useful work fraction as do failures during computation. This is because the duration of checkpointing/recovery is much smaller than that of computation and hence incurs less loss of useful work. Detailed analysis of failures during checkpointing/recovery is not presented.

[4] One job unit is the amount of work done by a failure-free processor without checkpointing in unit time.

For the remainder of Section 7, we assume an increased per-node MTTF of 3 years, as otherwise the failure effects dominate the system performance for large numbers of nodes. An MTTF of 3 years corresponds to a per-processor MTTF of 24 years for our system consisting of 8-processors per node, which is close to the 25-year MTTF of IBM mainframes reported in the literature [22].

7.2 Effect of Coordination

The coordinated checkpointing protocol requires that all the compute processors arrive at a safe point to take the checkpoint, and a timeout is used to avoid waiting indefinitely. This is not considered in the base model. This section first investigates the pure coordination effect without the timeout mechanism or failures and then combines them into the study. Three main points are observed from the results:

- ✧ Coordination does not affect system performance significantly, as the coordination effect is logarithmic in the number of compute processors (Figure 5) because we assume the processors have identical exponentially-distributed quiesce times. So coordination scales well for practical systems.
- ✧ Combination of timeout and coordination behaves like a probabilistic checkpoint-abort. Small timeouts (80s or less in Figure 6) hurt the useful work fraction, whereas large timeouts (100s or larger) do not significantly degrade the useful work fraction.
- ✧ As long as the coordination timeout is equal to or larger than a threshold value, the system performance is insensitive to the timeout value. The threshold value is fairly small for practical systems (100s in our experiment).

Coordination only. We assume that all the processors have identical, exponentially distributed quiesce times with a mean of MTTQ (Mean Time To Quiesce per processor). Figure 5 illustrates the pure coordination effect on the useful work fraction for different MTTQs. Failures and timeouts are not considered. According to the figure, the coordination effects are logarithmic in the number of compute processors. This is because an identical exponential distribution is assumed for each processor. Moreover, the rate of increase of coordination time (or overall quiesce time) is proportional to the MTTQ, and the coordination effect is also proportional to the checkpoint frequency (figures not shown here).

Effects of failures and timeouts. Figure 6 shows the system performance in the presence of failures with an MTTF of 3 years per node, checkpoint interval of 30 minutes, and MTTQ of 10 seconds. We use "no coordination" to indicate the case when no variation in the quiesce times among the compute processors is assumed and the quiesce time of the system as a whole is exponentially distributed with a mean of 10 seconds.

Figure 6 shows that the coordination without a timeout mechanism does not significantly degrade system performance, because the only additional overhead is the small coordination time. If a timeout is applied, the master may time out before the coordination is completed and abort the checkpointing. Then, if a failure occurs in the next computation interval, it causes the computation completed in the last interval to be lost. So the combination of coordination and timeout actually behaves like a probabilistic checkpoint-abort mechanism. The probability depends on the coordination time (MTTQ and number of compute processors) and the timeout. Small timeouts incur large probabilities of checkpoint abortion, and the benefit of limiting the processors' waiting time is offset by the loss of work due to frequent checkpoint abortions. The drastic curve drops for timeouts of 20-100 seconds in Figure 6 clearly show the performance degradation.

Figure 6 also shows that the system is insensitive to timeouts, provided they are large enough, because the overall coordination time increases slowly with the number of processors. For example, the 8192-processor system's performance with a timeout of 100s is only slightly better than a timeout of 120s and no timeout.

7.3 Effect of Correlated Failures

We recall that there are two categories of correlated failures considered in the paper.

Correlated failures due to error propagation only. Correlated failures due to error propagation are modeled with three parameters: *probability of correlated failure* (p_e), *frate_correlated_factor* (r,) and *correlated failure window*. As shown in Section 6, a typical value of r in real systems is on the order of a few hundred. In our experiments, r values of 400, 800, and 1600 are used for various p_e values, with a correlated failure window of 3 minutes.

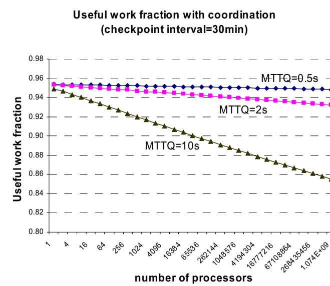

Figure 5 : Effects of coordination on system performance and scalability (no timeouts or failures)

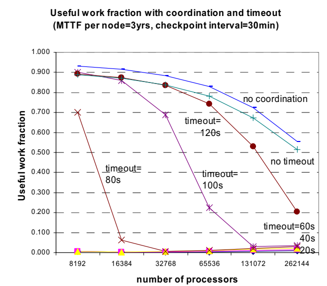

Figure 6 : Effects of coordination timeout on system performance and scalability (with failures)

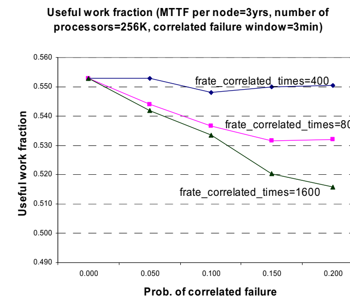

Figure 7 : Impact of correlated failures due to error propagation

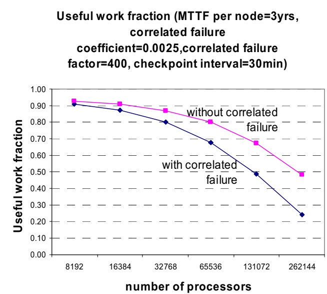

Figure 8 : Impact of generic correlated failures

The results of correlated failures in Figure 7 show that the useful work fraction is not susceptible to correlated failures due to error propagation (ranging between 0.51 and 0.56 in the figure). This is because we assume these failures only occur during recovery, and we observed that failures during recovery do not exert a significant effect on the useful work fraction.

Generic correlated failures. Generic correlated failures are modeled with two parameters: *correlated failure factor (r)* and *correlated failure coefficient (ρ).* An *r* value of 400 and *ρ* value of 0.0025 are used in our experiment. Therefore, the entire system failure rate gets doubled because of generic correlated failures. The results illustrated in Figure 8 show that, unlike correlated failures due to error propagation, there is a large performance degradation when generic correlated failures are present, and the performance degradation prevents the system from scaling well. For a system consisting of 256K processors with an MTTF of 3 years per node, the useful work fraction is reduced by 0.24 (51%).

8. Conclusions

This paper models a large-scale supercomputing system with coordinated checkpointing and rollback recovery. Unlike existing models in the literature, failures during checkpointing/recovery, coordination for checkpointing, and correlated failures are included in the model. The impact of these factors on system performance (measured as the *useful work fraction* and *total useful work*) as well as the scalability of systems with several hundred thousand processors is studied by simulating the model. The major conclusions from this study include:

- For a given checkpoint interval, MTTR, and MTTF, there is an optimum number of processors for which total useful work done by the system is maximized, e.g., for an MTTF per node of 1 year and an MTTR of 10 minutes, it is around 128K.
- The overall the useful work fraction is relatively low due to the effect of failures in large-scale systems.
- Correlated failures must be taken into account, as they degrade the performance and limit system scalability.

Acknowledgments

This work was supported in part by NSF grant ACI-0121658 ITR/AP and NSF grant ACI-CNS-0406351 (Next Generation Software). We thank Prof. W. Sanders for his suggestions on SAN modeling. We also thank Dr. B. Murphy for his advice and suggestions on the paper revision.

References

[1] N.R. Adiga et al., "An Overview of the Blue Gene/L," *Proc. of IEEE Int'l Conference on Supercomputing*, 2002.

[2] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," Technical Report of University of Tennessee, UT-CS, 1997.

[3] M. Chandy, L. Lamport. "Distributed Snapshots: Determinining Global States of Distributed Systems," *ACM Trans. on Computing Systems*, 3(1), 1985.

[4] R. Koo, S. Toueg, "Checkpointing and Recovery Rollback for Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 1, 1987.

[5] F. Petrini, K. Davis, J. C. Sancho, "System Level Fault Tolerance in Large-Scale Parallel Machines," *Proc. of IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'04),* 2004.

[6] D. Tang, R. K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Trans. on Computers,* Vol. 41, Num. 5, 1992.

[7] J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, Vol. 17, Num. 9, 1974.

[8] J. Daly, "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps," *Proc. of Int'l Conference on Computational Science*, 2003.

[9] G. P. Kavanaugh, W. H. Sanders, "Performance Analysis of Two Time-based Coordinated Checkpointing Protocols," *Proc. of IEEE Pacific Rim Int'l Symp. on Fault Tolerant Systems,* 1997.

[10] J. S. Plank, M. G. Thomason, "The Average Availability of Parallel Checkpointing Systems and Its Importance in Selecting Runtime Parameters," *IEEE Proc. Int'l Symp. on Fault-Tolerant Computing*, 1999.

[11] E. N. Elnozahy, J. S. Plank, W. K. Fuchs, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Trans. on Dependable and Secure Computing*, Vol. 1, Num. 2, 2004.

[12] N. H. Vaidya, "On Checkpoint Latency," *Proc. of IEEE Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, 1995.

[13] L. G. Valiant, "A Bridging Model for Parallel Computation" *Communications of the ACM*, Vol. 33, 1990

[14] E. Smirni, D. A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications," *Proc. of Int'l Conference on Computer Performance Evaluation: Modeling Techniques and Tools*, 1997.

[15] E. Rosti, et al., "Models of Parallel Applications with Large Computation and I/O Requirements," *IEEE Trans. on Software Engineering*, Vol.28, Num. 3, 2002.

[16] D. P. Siewiorek, R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2nd ed., Digital Press, 1992.

[17] G. Kulkarni, V. F. Nicola, K. S. Trivedi, "The Completion Time of a Job on Multimode Systems," *Advances in Applied Probability*, Vol. 19, 1987.

[18] Y. Zhang, et al., "Performance Implications of Failures in Large-Scale Cluster Scheduling," *10th Workshop on Job Scheduling Strategies for Parallel Processing,* 2004.

[19] B. Tuthill, et al. "IRIX Checkpoint and Restart Operation Guide," Document of Silicon Graphics, Inc., 1999.

[20] R. Iyer, D. Rossetti, "A Measurement-based Model for Workload Dependence of CPU Errors," *IEEE Trans. on Computers,* Vol. C-35, 1986.

[21]T. Courtney et al., "The Möbius Modeling Environment," *Tools of the 2003 Illinois Int'l Multiconference on Measurement, Modelling, and Evaluation of Computer Communication Systems*, Universität Dortmund Fachbereich Informatik research report no. 781, 2003.

[22] L. Spainhower, T. A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM Journal of Research and Development,* Vol. 43, Num. 5/6, 1999.

[23] G. Bronevetsky et al., "Automated Application-level Checkpointing of MPI Programs," *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[24] S. Agarwal et al., "Adaptive Incremental Checkpointing for Massively Parallel Systems," *Proc. of IEEE Int'l Conference on Supercomputing,* 2004.

[25] L. Wang et al., "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," Technical Report of University of Illinois, 2005.