Adaptive Fault Management of Parallel Applications for High-Performance Computing

Zhiling Lan, *Member*, *IEEE Computer Society*, and Yawei Li, *Student Member*, *IEEE*

Abstract—As the scale of high-performance computing (HPC) continues to grow, failure resilience of parallel applications becomes crucial. In this paper, we present FT-Pro, an adaptive fault management approach that combines proactive migration with reactive checkpointing. It aims to enable parallel applications to avoid anticipated failures via preventive migration and, in the case of unforeseeable failures, to minimize their impact through selective checkpointing. An adaptation manager is designed to make runtime decisions in response to failure prediction. Extensive experiments, by means of stochastic modeling and case studies with real applications, indicate that FT-Pro outperforms periodic checkpointing, in terms of reducing application completion times and improving resource utilization, by up to 43 percent.

Index Terms—Adaptive fault management, parallel applications, high-performance computing, large-scale systems.

1 Introduction

In the field of high-performance computing (HPC), the insatiable demand for more computational power in science and engineering has driven the development of ever-growing supercomputers. Production systems with hundreds of thousands of processors, ranging from tightly coupled proprietary clusters to loosely coupled commodity-based clusters, are being designed and deployed [1]. For systems of this scale, *reliability* becomes a critical concern as the systemwide mean time between failures (MTBF) decreases dramatically with the increasing count of components. Studies have shown that MTBFs for teraflops- and petaflops-scale systems are only on the order of 10-100 hours, even for systems based on ultrareliable components [2], [3]. Meanwhile, to accurately model realistic problems, parallel applications are designed to span across a substantial number of processors for days or weeks until completion. Unfortunately, the current state of parallel processing is such that the failure of a single process usually aborts the entire application. As a consequence, large applications find it difficult to make any forward progress because of failures. This situation is likely to deteriorate as systems get bigger while applications become larger.

Checkpointing is the conventional method for fault tolerance. It is reactive by periodically saving a snapshot of the application and using it for restarting the execution in case of failures [4], [5]. When one of the application processes experiences a failure, all the processes, including nonfaulty processes, have to roll back to the previously saved state prior to the failure. Thus, a significant performance loss can be incurred due to the work loss and failure recovery. Unlike checkpointing, the newly emerged *proactive approach* (e.g., process migration) takes preventive actions before failures, thereby preventing failure experience and avoiding rollbacks [6], [7]. Nevertheless, it requires accurate fault prediction, which is hardly achievable in practice. Hence, the proactive approach alone is unlikely to be sufficient to provide a reliable solution for fault management in HPC.

In this paper, we present *FT-Pro, an adaptive approach* for fault management of parallel applications by combining the merits of proactive migration and reactive checkpointing. Proactive actions enable applications to avoid anticipated faults if possible, and reactive actions intend to minimize the impact of unforeseeable failures. The goal is to reduce application completion time in the presence of failures. While checkpointing and process migration have been studied extensively, the key challenge facing the design of FT-Pro is *how to effectively select an appropriate action at runtime*. Toward this end, an adaptation manager is designed to choose a best fit action from opportunistic skip, reactive checkpointing, and preemptive migration by considering a number of factors.

We demonstrate that FT-Pro can enhance fault resilience of parallel applications and consequently improve their performance, by means of stochastic modeling and case studies with parallel applications. Our results indicate that FT-Pro outperforms periodic checkpointing, in terms of reducing application completion time and improving resource utilization, by up to 43 percent. A modest allocation of spare nodes (less than 5 percent) is usually sufficient for FT-Pro to achieve the above gain. Additionally, the overhead caused by FT-Pro is less than 3 percent.

FT-Pro is built on the belief that technological innovation combined with advanced data analysis makes it possible to predict failures with a certain degree of accuracy. Recent studies with actual failure traces have shown that with a proper system monitoring facility, critical events can be predicted with an accuracy of up to 80 percent [8], [9], [10], [11], [12]. A distinguishing feature of FT-Pro is that it does

• *The authors are with the Department of Computer Science, Illinois Institute of Technology, 10 W. 31st Street, Chicago, IL 60616. E-mail: {lan, liyawei}@iit.edu.*

Manuscript received 25 Apr. 2007; revised 14 Aug. 2007; accepted 26 Feb. 2008; published online 20 May 2008.
Recommended for acceptance by C.-L. Wang.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-04-0140.
Digital Object Identifier no. 10.1109/TC.2008.90.

not require perfect failure prediction to be effective. As we will show, FT-Pro outperforms periodic checkpointing as long as failure prediction is capable of capturing 30 percent of failures, which is feasible by using existing predictive technologies.

FT-Pro is intended to bridge the gap between failure prediction and fault handling techniques by effectively exploring failure prediction for better fault management. It complements the research on checkpointing and process migration by providing adaptive strategies for runtime coordination of these techniques. The proposed FT-Pro can be integrated with state-of-the-art failure predictors and existing fault-tolerant tools [9], [10], [13], [14], [15], [16], [17], [18], [19], [20] to provide an end-to-end system for adaptive fault management of parallel applications.

The remainder of this paper is organized as follows: Section 2 briefly discusses the related work. Section 3 gives an overview of FT-Pro, followed by a detailed description of its adaptation manager in Section 4. Section 5 describes our stochastic modeling and simulation results. Section 6 presents case studies with a number of parallel applications. Finally, Section 7 summarizes the paper and points out future directions.

2 Related Work

Checkpointing, in various forms, has been studied extensively over the past decades. A detailed description and comparison of different checkpointing techniques can be found in [4]. In the field of HPC, a number of checkpointing libraries and tools have been developed, and examples include libckpt [16], BLCR [17], open MPI [18], MPICH-V [13], and the Cornell Checkpoint (pre)Compiler (C^3) [15]. Checkpointing optimization is generally approached by selecting optimal intervals [21], [22], [23] or reducing the overhead per operation, such as copy-on-write [24], incremental checkpointing [25], diskless checkpointing [26], [27], double in-memory technique [28], etc. Generally speaking, checkpointing is a conservative method. It requires increasing the number of checkpoints to deal with higher failure rates as the computing scale increases.

Much progress has been made in failure analysis and prediction. On the hardware side, modern computer systems are designed with various features (e.g., hardware sensors) that can monitor the degradation of an attribute over time for early detection of hardware errors [29], [30], [31], [32]. On the software side, a variety of predictive techniques have been developed to infer implicit and useful fault patterns from historical data for failure prediction. They can be broadly classified as *model-based* or *data-driven*. A model-based approach derives an analytical or probabilistic model of the system and then triggers a warning when a deviation from the model is detected [33], [34], [35], [36], [37]. Data mining, in combination with intelligent systems, focuses on learning and classifying fault patterns without a priori model [8], [9], [10], [11]. In addition, leading HPC vendors have started to integrate hardware and software components into their systems for comprehensive fault analysis, such as the Cluster Monitoring and Control System (CMCS) service in IBM Blue Gene systems and the Cray RAS and Management System (CRMS) in Cray XT series systems [38], [39].

Leveraging the research on failure prediction, there are growing interests in utilizing failure prediction for proactive fault management. For example, the HA-OSCAR project provides high availability for head nodes in Beowulf clusters by using a failover strategy [40]. There are several research efforts on failure-aware scheduling [41], [42]. Process or object migration is a widely used proactive technique [7]. Most migration methods adopt the *stop-and-restart* model for the migration of parallel applications, in which the application takes a checkpoint and then restarts on a new set of resources—after swapping the failure-prone nodes with healthy ones [43]. There are several active research projects on providing *live migration* support for MPI applications [19], [20], [44]. While the proactive approach is cost efficient, it requires accurate failure prediction. In practice, prediction misses and false alarms are common. Prediction misses can lead to significant performance damage, whereas false alarms can introduce intolerable overhead. Hence, solely relying on the proactive approach is not sufficient for HPC.

Recognizing the limitations of reactive and proactive approaches, FT-Pro aims at getting the best of both worlds by intelligently coordinating process migration with checkpointing. Similar to cooperative checkpointing [45], FT-Pro ignores unnecessary fault-tolerant requests when failure impact is trivial. Further, it enables an application to avoid imminent failures through preventive migration. The adaptation between process migration and selective checkpointing is built upon a quantitative modeling of application performance.

The idea of using adaptation for fault management is not new. It has been used in the fields such as mission-critical spacecrafts and storage systems [46], [47]. Nevertheless, to the best of our knowledge, we are among the first to exploit adaptive fault management for HPC. Different from the above research that mainly focuses on efficiently utilizing duplicated components for high availability, this work centers upon reducing the application completion time by dynamically choosing between proactive and reactive actions.

3 Overview of FT-Pro

We define a *failure* as any event in hardware or software that results in an immediate termination of a running application. To be effective, FT-Pro requires the presence of a failure predictor. Predictive techniques mentioned in Section 2, as well as our own previous work [11], [48], [49], can be used to provide such an engine. Failure prediction can be either *categorical*, where the predictor forecasts whether a failure event will occur or not in the near future, or *numerical*, where the probability of failures is provided for a given time window. Numerical results can be easily translated into categorical results by applying threshold-based splitting; hence, in this paper, we uniformly describe *failure prediction* as a process that periodically estimates whether a node will experience failures in a given time window (e.g., a few minutes to an hour). Such a prediction mechanism is generally measured

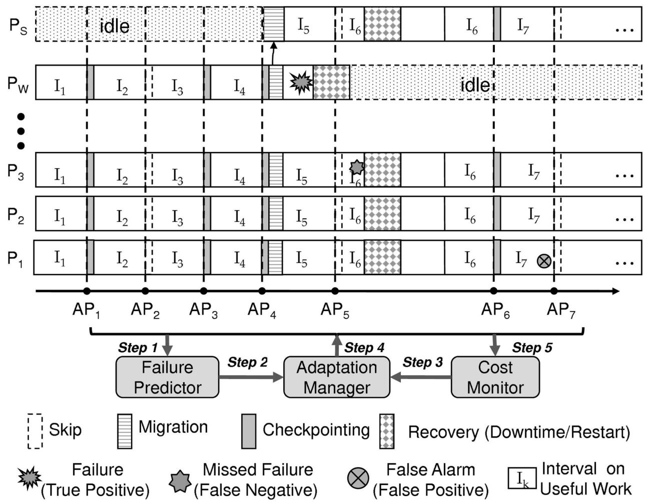


Fig. 1. The main idea and steps of FT-Pro.

by two accuracy metrics: *precision* and *recall*. Precision is defined as the proportion of correct predictions to all the predictions made (i.e., $\frac{T_p}{T_p+F_p}$), and recall is the proportion of correct predictions to the number of failures (i.e., $\frac{T_p}{T_p+F_n}$). Here, T_p is the number of correct predictions (i.e., *true positives*), F_p is the number of false alarms (i.e., *false positives*), and F_n is the number of missed failures (i.e., *false negatives*). Obviously, a good prediction engine should achieve a high value (closer to 1.0) for both metrics.

A user can issue (or set up) fault-tolerant requests, denoted as *adaptation points*, during the application execution, and FT-Pro makes a runtime decision upon these points to determine which action should be taken [50]. For example, a user may set adaptation points to where the application completes a segment of useful work, i.e., the computation that is not redone due to a failure [51]. Three actions are currently considered in FT-Pro:

- *SKIP.* The fault-tolerant request is ignored. This action is taken to remove unnecessary actions when the failure impact is trivial.
- *CHECKPOINT.* The application takes a checkpoint. This action is to reduce application work loss caused by unforeseeable failures.
- *MIGRATION.* The processes on suspicious nodes (i.e., the nodes predicted to be failure prone in the near future) are transferred to healthy nodes. This action is to avoid an imminent failure. Currently, we assume that process migration is conducted by taking a coordinated checkpoint followed by a stop-and-restart action [43].

The main idea of FT-Pro is illustrated in Fig. 1, where the useful work is segmented into intervals denoted by I_k. Suppose the application runs on nodes denoted as $\{P_1, P_2, \ldots, P_W\}$ and one spare node, denoted as P_S, is allocated for proactive actions. Spare nodes can be either reserved at the application submission time or allocated through the resource manager during execution. Upon each adaptation point AP_i, FT-Pro first consults *the failure predictor* to get the status of each computation node. It then triggers *the adaptation manager* (discussed in Section 4) to determine a best fit action in response to the failure prediction, followed by invoking the corresponding action on the application. Here, *the cost monitor component* keeps track of the runtime overhead of different fault-tolerant actions. If the application fails during checkpointing or migration, it rolls back to the most recent checkpoint. Let us take a look at a few examples:

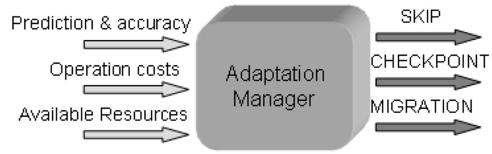


Fig. 2. Diagram of the adaptation algorithm.

- FT-Pro always grants the first fault-tolerant request at AP_1 by taking a checkpoint.
- At AP_2, where the failure predictor does not anticipate any failure in the near future, given that the failure impact during the next interval is trivial, FT-Pro ignores the request by taking a SKIP action. Similarly, FT-Pro takes a SKIP action at AP_7.
- At AP_3, considering that the work loss would be significant if an unforeseeable failure occurred in the next interval, FT-Pro decides to take a coordinated checkpoint although no failure warning is issued at this point.
- At AP_4, where the predictor forecasts a failure on P_W (which turns out to be a true positive), FT-Pro transfers the process from P_W to the spare node P_S. The application is first checkpointed, followed by a process migration. Once repaired, P_W becomes a spare node.
- At AP_5, where the predictor fails to warn the upcoming failure on P_3 (e.g., a false negative), FT-Pro takes a SKIP action. The application therefore loses the work done between AP_5 and the failure occurrence, suffers from failure recovery, rolls back to the last checkpoint completed at AP_4, recomputes the work due to the failure, and proceeds to the next adaptation point AP_6.
- In case of false alarms, such as at AP_6, where the predictor erroneously gives a warning (e.g., a false positive), FT-Pro takes a checkpoint.

4 Adaptation Manager

The adaptation manager is responsible for determining the most suitable action upon each adaptation point. Designing an efficient manager is challenging. First, it must consider a range of factors that may impact application performance. These include not only the available spare nodes but also costs and benefits of different fault-tolerant actions. Second, given that a failure predictor is subjected to false negatives and false positives, it must take account of both errors during its decision-making process. Last, it must make a timely decision without causing noticeable overhead on application performance.

By considering the above requirements, we develop an adaptation manager, which is illustrated in Fig. 2. It takes account of three sets of parameters for decision making,

TABLE 1
Nomenclature

Symbol	Description
T_{appl}	Application failure-free execution time, i.e. time spent on useful work
T_{ckp}, T_{ft-pro}	Application completion time by using checkpointing or FT-Pro
N_W	Number of computation nodes allocated to the application
N_S	Number of spare nodes allocated to the application
I	Fault tolerance interval
$l_{current}$	Index of the current adaptation point
l_{last}	Index of the last adaptation point where a checkpoint is taken
$precision, recall$	Prediction accuracy, defined as $\frac{T_p}{T_p+F_p}$ and $\frac{T_p}{T_p+F_n}$ respectively
f_{appl}	application failure probability
C_r	Mean recovery cost
C_{ckp}	Checkpointing overhead
C_{pm}	Migration overhead
E_{next}	Expected time for the application to reach the next adaptation point

namely, prediction accuracy, operation costs of different actions, and the number of available resources for proactive actions. Before presenting our detailed algorithm, we first list a set of nomenclatures that will be frequently used in the rest of the paper (see Table 1).

Upon each adaptation point AP_i, if the failure predictor anticipates any failure on a computation node, the manager takes account of prediction $precision$. Specifically, it estimates E_{next}—the expected time for the application to reach the next adaptation point AP_{i+1}—and *selects the action that minimizes* E_{next}. Suppose the current interval index is $l_{current}$. Due to the uncertainty of the exact failure time, a conservative policy is adopted by assuming that the failure will occur immediately before the next adaptation point. Here, "conservative" is with respect to the amount of work loss. The choices of actions of the manager are described as follows:

- SKIP. 1) If a failure occurs in the next interval, the application spends I time for the execution, C_r time for the recovery, and then $[I + (l_{current} - l_{last}) \times I]$ time to reach the next adaptation point from the most recent checkpoint. 2) If no failure occurs, the application smoothly proceeds to the next adaptation point. By using the total probability law, we have

$$\begin{aligned} E_{next} &= [C_r + (2 + l_{current} - l_{last}) \times I] \times f_{appl} \\ &\quad + I \times (1 - f_{appl}), \\ f_{appl} &= 1 - (1 - precision)^{N_W^f}. \end{aligned} \tag{1}$$

  Here, N_W^f denotes the number of computation nodes that are predicted to be failure prone in the next interval.
- CHECKPOINT. The application first spends C_{ckp} for performing checkpointing and then updates l_{last}. 1) If a failure occurs in the next interval, the application spends I time for the execution, C_r time for the recovery, and then I time to reach the next adaptation point from the current adaptation point. 2) If no failure occurs, the application smoothly proceeds to the next adaptation point. Thus, we have

$$\begin{aligned} E_{next} &= (C_{ckp} + C_r + 2I) \times f_{appl} + (I + C_{ckp}) \times (1 - f_{appl}), \\ f_{appl} &= 1 - (1 - precision)^{N_W^f}. \end{aligned} \tag{2}$$

- MIGRATION. The application first spends C_{pm} for process migration and updates l_{last}. Due to the possibility of multiple simultaneous failures, the number of spare nodes may not be enough to accommodate all the migration requests. FT-Pro uses a best effort strategy to migrate as many processes as possible. E_{next} is calculated as follows:

$$\begin{aligned} E_{next} &= (C_{pm} + C_r + 2I) \times f_{appl} + (I + C_{pm}) \times (1 - f_{appl}), \\ f_{appl} &= \begin{cases} 1 - (1 - precision)^{N_W^f - N_S^h}, & N_W^f > N_S^h, \\ 0, & N_W^f \le N_S^h. \end{cases} \end{aligned} \tag{3}$$

  Here, N_S^h denotes the number of spare nodes that will be failure free during the next interval.

Upon an adaptation point where the failure predictor does not give any warning, the manager takes account of prediction $recall$. Given the possibility of unpredictable failures, the performance loss could be significant when a number of SKIP actions have been taken continuously before an unpredicted failure. Hence, when the number of consecutive SKIP actions reaches a threshold, rather than blindly relying on the prediction, the manager enforces a checkpoint. The rationale here is to enforce a checkpoint in case the failure prediction is wrong. Currently, the threshold is set to $\frac{MTBF}{I \cdot (1 - recall)}$. It is based on an intuitive estimation that the expected time between false negatives is $\frac{MTBF}{(1 - recall)}$. Clearly, if $recall$ is equal to 1.0, the threshold is ∞, meaning that there is no need to enforce preventive checkpoints as the predictor is able to capture every failure.

The special cases are when $precision$ or $recall$ is zero. If $precision$ is zero, it means that every alarm provided is a false alarm. According to (1)-(3), a SKIP action is selected upon these adaptation points. If $recall$ is zero, it means that every failure is missed by the predictor. In this case, periodic checkpointing is adopted.

In addition, the adaptation manager adopts an automatic mechanism to assess the application-specific parameters listed in (1)-(3), namely, the checkpointing overhead C_{ckp} and the migration overhead C_{pm}, for its decision making. Both parameters depend on many factors like the implementations of checkpointing and migration, system configurations, computation scale, and application characteristics. The manager automates the acquisition of these parameters, without user involvement, in the following ways:

- Upon the initiation of the application, it records the application starting cost. Further, it always grants the first checkpoint request (see Fig. 1). A recent study done by Oliner et al. has proved that any strategy that skips the first checkpoint is noncompetitive [52]. At the second adaptation point, the manager uses the recorded checkpointing overhead C_{ckp} and estimates C_{pm} as the summation of C_{ckp} and the application starting cost.
- During the subsequent execution, it always keeps track of these parameters via the cost monitor component and uses the last measured values for decision making at the next adaptation point.

The adaptation manager can be easily implemented on top of existing checkpointing tools. For instance, we implement FT-Pro with MPICH-VCL [13] by adding the adaptation manager as a new component (see Fig. 8).

5 Stochastic Modeling

We now proceed to comprehensively evaluate the performance of FT-Pro. In this section, we present a stochastic model of FT-Pro, and case studies with applications will be discussed in the next section.

5.1 Performance Metrics

Three performance metrics are used to compare FT-Pro with periodic checkpointing:

1. *Execution time.* Considering that the main objective of HPC is to reduce application execution time, we therefore use it as our primary metric:

$$T = \begin{cases} T_{ckp} & \text{using checkpointing,} \\ T_{ft-pro} & \text{using FT-Pro.} \end{cases} \tag{4}$$

2. *Time reduction.* For the convenience of comparison, we also measure the relative time reduction by using FT-Pro over periodic checkpointing. It is defined as

$$\frac{T_{ckp} - T_{ft-pro}}{T_{ckp}}. \tag{5}$$

3. *Service unit (SU) reduction.* In production HPC systems, users are generally charged based on SUs—the product of the number of processors and time—used by their applications. Thus, we measure the relative reduction on SUs, which represents the improvement of FT-Pro with respect to system utilization. It is defined as

$$\frac{N_W \cdot T_{ckp} - (N_W + N_S) \cdot T_{ft-pro}}{N_W \cdot T_{ckp}}. \tag{6}$$

5.2 Model Description

Application performance (e.g., application completion time) can be regarded as a continuous accumulated reward, which is affected by many factors, including failure arrival/recovery, fault-tolerant actions, and available spare nodes. Such behaviors are difficult to be modeled by the traditional stochastic Petri net (SPN); hence, we built a fluid SPN (FSPN) to analyze FT-Pro and to validate its adaptive strategy. Basically, FSPN is an extension of the classical SPN and is capable of modeling both discrete and continuous variables. Additional details about FSPN can be found in [53].

Fig. 3 presents our FSPN model of FT-Pro. It is generated by using the SPNP package developed at Duke University [53]. The model consists of three subnets. The first subnet—*subnet of failure behavior*—describes the failure behavior of the system, the second one—*subnet of adaptation manager*—models the behavior of the adaptation manager in FT-Pro, and the last one—*subnet of application performance*—uses the continuous fluid to model application completion time. The detailed explanation of the model is given in Appendix A.

A FSPN model is also built for periodic checkpointing. We then used these models to study FT-Pro as against periodic checkpointing.

5.3 Modeling Results

Four sets of simulations are conducted to examine the impact of computation scales, allocation of spare nodes, prediction accuracies, and operational costs, respectively. The baseline configuration is summarized in Table 2. These parameters and their corresponding ranges are selected based on the results reported in [5], [42], and [54]. Note that the interval I, calculated based on the well-known optimal frequency [21], is used as the adaptation interval for FT-Pro and the checkpoint interval for periodic checkpointing.

5.3.1 Impact of Computation Scales

In the first set of simulations, we tune the number of computation nodes from 16 to 192 (the maximum number of processing units allowed in SPNP is 200), with only one spare node being allocated. The purpose is to study the impact of computation scales on the performance of FT-Pro.

To reflect the fact that the checkpointing overhead and the migration overhead generally grow with the application size, we make corresponding changes on the values of C_{ckp} and C_{pm}. How to accurately set these parameters is difficult, as they are application dependent. Considering the principle of coordinated checkpointing, we use a simple model of $(O_{IO} + O_{msg})$, where O_{IO} is a fixed I/O overhead, and O_{msg} is the message passing overhead, which is linearly increased with the growing scale of computation. According to this formula, the checkpointing overhead C_{ckp} is set to 0.625,

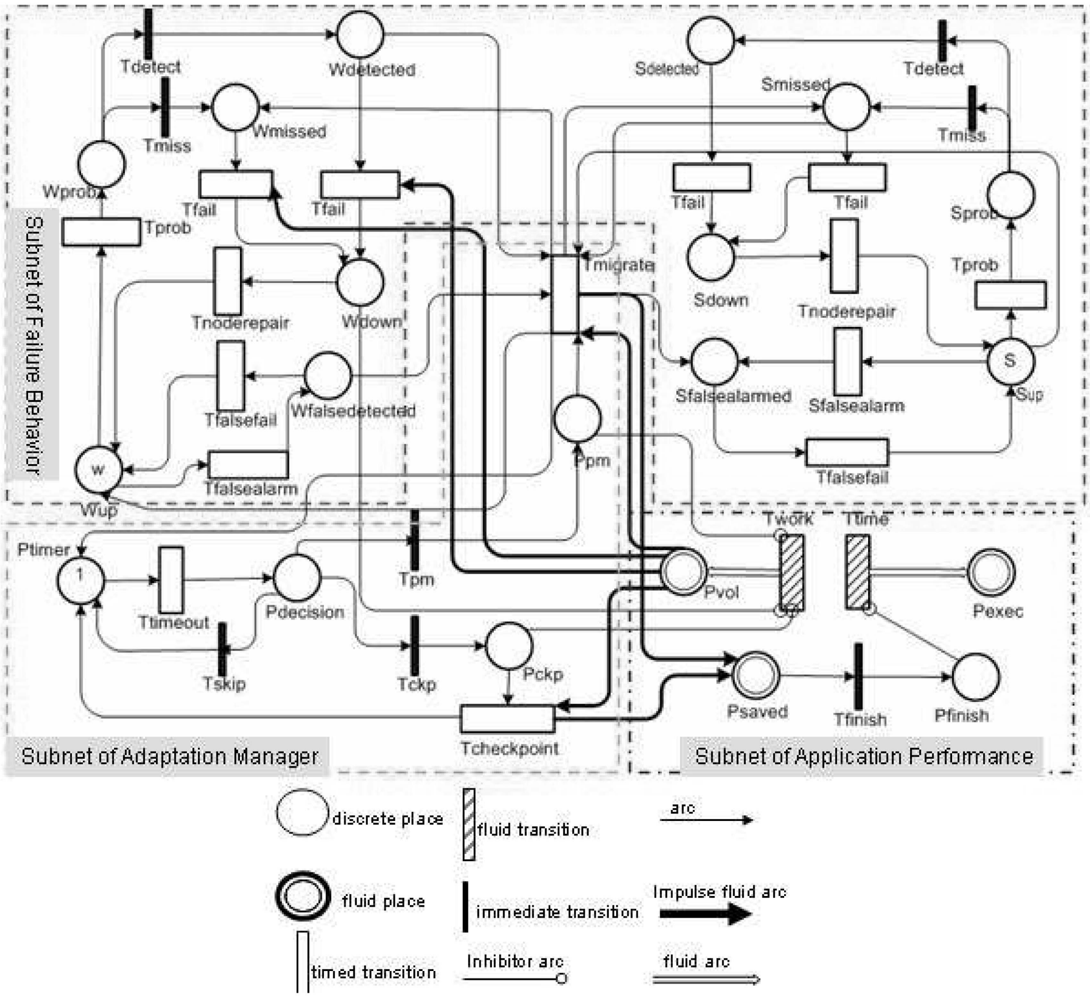


Fig. 3. FSPN modeling of FT-Pro. It consists of three subnets: 1) subnet of failure behavior, 2) subnet of adaptation manager, and 3) subnet of application performance. Together, they model the execution of parallel applications running on clustering systems in the presence of failures.

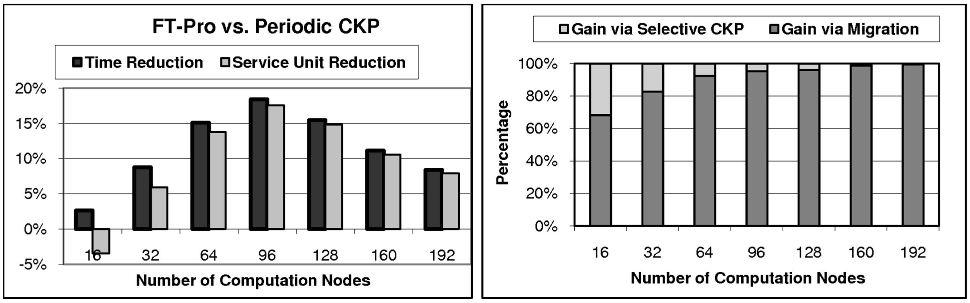
TABLE 2
Baseline Parameters

N_W	N_S	T_{appl}	I	MTBF (node)	C_r	C_{ckp}	C_{pm}	$precision$	$recall$
128	1	1000 hrs	48 min.	500 hrs	2 hrs	5 min.	10 min.	0.7	0.7

0.917, 1.5, 2.667, 3.833, 5.0, 6.167, and 7.33 minutes as the number of computation nodes N_W changed from 8 to 192. Depending on the migration implementation (e.g., stop-and-restart model [43] or live migration [19]), the overhead caused by migration may be different too. Here, we set the migration overhead C_{pm} to be twice the value of the corresponding C_{ckp}.

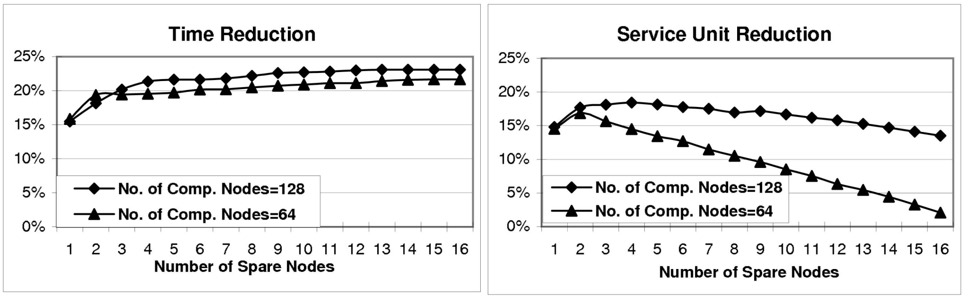
Fig. 4a shows the $Time\_Reduction$ and $SU\_Reduction$ achieved by FT-Pro with different computation scales. It shows three interesting patterns. First, although FT-Pro yieldss a positive value on $Time\_Reduction$ when the computation scale is set to 16, the $SU\_Reduction$ value is negative. This indicates that when the computing scale is relatively small (e.g., 16), the time reduction brought by FT-Pro may be overshadowed by the use of additional computing resources, thereby resulting in negative resource utilization. Second is the increasing gain achieved by FT-Pro as the number of computation nodes is increased from 16 to 96. When more nodes are used, application failure probability is getting higher, thereby implying more opportunities for FT-Pro to reduce the performance overhead by avoiding failures. The third feature is the decreasing benefit when the number of computation nodes is increased beyond 96. As shown in Table 2, in this set of simulations, only one spare node is allocated even when the number of computation nodes is set to 192. As a result, FT-Pro cannot avoid imminent failures due to the lack of available spare nodes. Note that even when the scale increases beyond 128 with only one spare node, FT-Pro still outperforms periodic checkpointing by more than 8.4 percent in terms of $Time\_Reduction$ and 7.9 percent in terms of $SU\_Reduction$.

Fig. 4b shows the breakdown of the gain achieved by FT-Pro. The benefit of FT-Pro comes from two parts: one is



(a) (b)

Fig. 4. Impact of computation scales, where the number of spare nodes is set to one. (a) $Time\_Reduction$ and $SU\_Reduction$ achieved by FT-Pro. (b) The breakdown of the gain achieved by FT-Pro. Generally, FT-Pro does better than periodic checkpointing. The decreasing performance when the size of computation is increased beyond 96 is due to the scarce number of spare nodes. The majority gain of FT-Pro comes from proactive migration, suggesting that avoiding failures in response to prediction is essential for reducing the performance loss caused by potential failures.



(a) (b)

Fig. 5. Impact of spare nodes, where the number of spare nodes ranges from 1 to 16. (a) $Time\_Reduction$ achieved by FT-Pro as against periodic checkpointing. (b) $SU\_Reduction$ achieved by FT-Pro as against periodic checkpointing. Obviously, the more number of spare nodes is allocated, the better $Time\_Reduction$ is. However, allocating more spare nodes does not always increase the overall resource utilization.

to take preventive migration to avoid imminent failures, and the other is to skip unnecessary checkpoints when the failure impact is low. The figure indicates that the benefit achieved by selective checkpointing is relatively low. This is caused by the fact that we use an optimal frequency for checkpointing, thereby resulting in few chances for FT-Pro to skip unnecessary checkpoints. Obviously, failure avoidance through preventive migration dominates the gain, especially when the computing scale is large. For instance, when the number of computation nodes is set to 16, 68 percent of the gain comes from proactive migrations. The percentage increases to nearly 100 percent when the computation scale is increased beyond 160. We observe a similar pattern in our case studies (see Section 6). This suggests that to fully utilize failure prediction, taking proactive actions, in addition to skipping unnecessary checkpoints, is essential for reducing the performance loss caused by potential failures.

5.3.2 Impact of Spare Nodes

In this set of simulations, we investigate the sensitivity of FT-Pro to the allocation of spare nodes.

Fig. 5 presents the $Time\_Reduction$ and $SU\_Reduction$ achieved by FT-Pro over periodic checkpointing, where the number of spare nodes N_S is ranging between 1 and 16. There are two curves in each plot, showing the result with the number of computation nodes set to 64 and 128, respectively.

As shown in Fig. 5a, although the improvement on $Time\_Reduction$ becomes less obvious as the number of spare nodes increases, it grows monotonically with the increasing number of spare nodes. With more spare nodes allocated, FT-Pro can more effectively avoid simultaneous failures. In other words, if time is the only concern, then allocating more spare nodes definitely helps. A better performance is achieved with the 128-node setting, as compared to the 64-node setting. The main reason is that the larger a computation is, the higher chance the application has to experience failures and the more amount of work loss can be introduced in case of failures. As a result, FT-Pro has more opportunities to provide improvement.

Fig. 5b presents $SU\_Reduction$ with varying numbers of spare nodes. While the gain is always positive, it also indicates that allocating more spare nodes does not always increase the overall resource utilization, as $SU\_Reduction$ decreases beyond a certain point. According to the figure, when the number of computation nodes is set to 64 and 128, the optimal allocation is 2 and 4, respectively. The figure also shows that in general, by allocating less than 5 percent of nodes for accommodating preventive actions (e.g., one to three when N_W is 64, and one to six when N_W is 128), the adaptive fault management approach outperforms periodic checkpointing by 14 percent-24 percent in terms of both $Time\_Reduction$ and $SU\_Reduction$. The optimal allocation of spare nodes depends on many factors, including failure behaviors (e.g., how often are simultaneous failures) and application size (e.g., how many nodes are requested for computation). A theoretic proof of the optimal allocation of spare nodes is the subject of our ongoing research.

TABLE 3
Application Completion Times by Using FT-Pro (in Hours), Where the Application Completion Time by Using Periodic Checkpointing Is 6,500 Hours

		Precision									
		1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
Recall	1.0	4763	4853	4939	4944	4947	4980	5037	5322	5549	5915
	0.9	4907	4963	5000	5089	5135	5150	5176	5427	5712	6131
	0.8	5215	5334	5363	5367	5388	5404	5461	5474	5831	6198
	0.7	5425	5468	5475	5494	5521	5554	5665	5756	5905	6231
	0.6	5646	5689	5700	5745	5790	5826	5847	5897	5963	6230
	0.5	5775	5846	5919	5936	5957	6005	6067	6145	6182	6435
	0.4	5974	6150	6163	6170	6182	6231	6246	6345	6359	6579
	0.3	6218	6338	6357	6394	6458	6462	6466	6493	6507	6639
	0.2	6420	6554	6598	6618	6458	6666	6719	6732	6751	7055
	0.1	6710	6834	6852	6858	6884	6886	6955	6993	7099	7134

The number of computation nodes N_W is 128, and the number of spare nodes N_S is set to one.

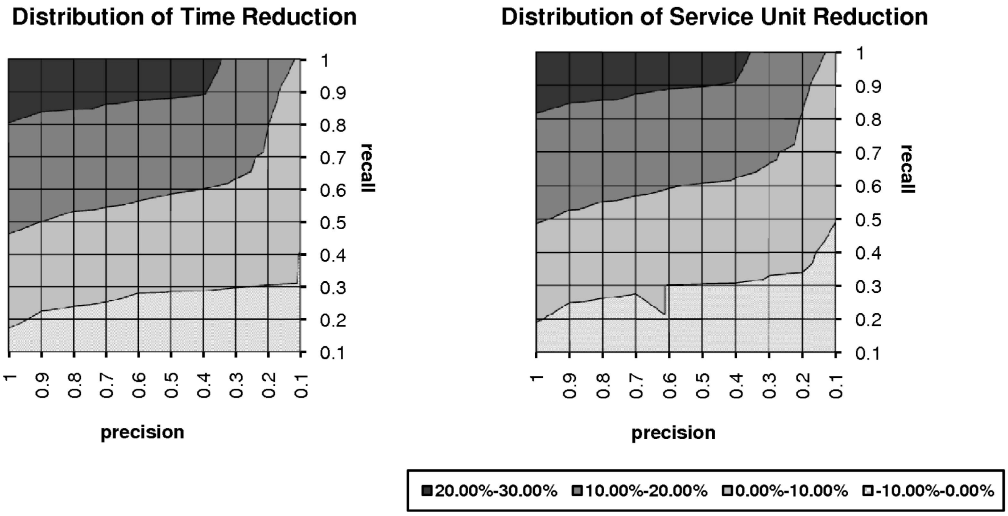


Fig. 6. Impact of prediction accuracies, corresponding to Table 3. Under perfect prediction, FT-Pro outperforms periodic checkpointing by about 26 percent. For FT-Pro to be effective, the prediction engine should be able to capture 30 percent of failures.

5.3.3 Impact of Prediction Accuracies

The performance of FT-Pro is influenced by prediction accuracy. Obviously, the more accurate a prediction mechanism is, the better performance FT-Pro can achieve. In this set of simulations, we simulate different levels of prediction accuracies and quantify the amount of gain achieved by FT-Pro under different $precision$ and $recall$ values.

Table 3 lists the application completion times obtained by using FT-Pro, where $precision$ and $recall$ range from 1.0 to 0.1. Here, the computation scale is set to 128, and only one additional spare node is allocated. In Fig. 6, we pictorially show the distributions of $Time\_Reduction$ and $SU\_Reduction$ with regard to different $precision$ and $recall$ values.

The results clearly show that the more accurate a prediction mechanism is, the higher the gain that FT-Pro can provide. For example, the best performance is achieved when $precision = recall = 1.0$ (perfect prediction), and the worst case occurs when $precision = recall = 0.1$ (meaning that 90 percent of the predicted failures are false alarms and 90 percent of the failures are not captured by the failure predictor). Under perfect prediction, the optimal gain achieved by FT-Pro is 26.72 percent on $Time\_Reduction$ and 26.15 percent on $SU\_Reduction$. When both $precision$ and $recall$ are in the range of [0.6, 1.0], FT-Pro outperforms periodic checkpointing by over 10 percent. Our prediction studies have shown that with a proper error-checking mechanism, it is feasible to predict failures with both rates above 0.6 [11]. Similar results have also been reported in [9] and [10].

Additionally, as long as both $precision$ and $recall$ are in the range of [0.3, 1.0], FT-Pro always surpasses periodic checkpointing with a positive gain in terms of $Time\_Reduction$. In other words, *to be effective, the failure predictor should be able to capture at least 30 percent of failures*. The figure also suggests that FT-Pro could be further improved by turning off adaptation when either of $precision$ and $recall$ is lower than 0.3.

The figure also indicates that FT-Pro is more robust to $precision$ than to $recall$. For instance, under the extreme case where $precision$ is 0.1 (meaning that there is only one true failure for every 10 predicted failures), FT-Pro is still capable of producing a positive $Time\_Reduction$ and $SU\_Reduction$ as long as $recall$ is controlled above 0.50. Note that FT-Pro adopts a cooperative mechanism for adaptive management such that the user sets his/her fault-tolerant requests, and FT-Pro makes runtime decisions on the invocation of different actions upon these points. If the

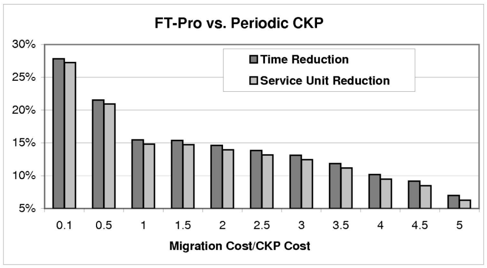


Fig. 7. Impact of operation costs, where the ratio between the migration overhead and the checkpointing overhead is tuned between 0.1 and 5.0. The number of computation nodes is set to 128, and only one spare node is allocated. The performance gain achieved by adaptive fault tolerance is apparent. A more efficient migration support such as live migration can make FT-Pro more promising.

warning is a false alarm, rather than blindly triggering a MIGRATION action, FT-Pro may take a different action based on its evaluation, thereby making it robust to false positives.

5.3.4 Impact of Operation Costs

Finally, we investigate the impact of operation costs on the performance of FT-Pro. More specifically, we change the ratio between the migration overhead and the checkpointing overhead by fixing C_{ckp} and varying C_{pm}. The results achieved by FT-Pro as against periodic checkpointing are plotted in Fig. 7. Here, the number of computation nodes is set to 128, and only one spare node is allocated. Obviously, a more efficient migration support can yield better performance. Even when the migration overhead is four times the checkpointing overhead, FT-Pro still maintains $Time\_Reduction$ at 11 percent.

In our current design, we use a stop-and-restart migration, meaning that the application is stopped and restarted on a new set of computation nodes after the suspicious nodes are replaced by spare nodes. Our case studies with real applications (discussed in the next section) show that with such an expensive migration support, the migration overhead C_{pm} is generally less than $3C_{ckp}$. We believe that the development of live migration such as the tool listed in [19] and [20] can significantly reduce the migration overhead, thereby making FT-Pro more promising.

5.4 Modeling Summary

In summary, the above stochastic study has indicated the following:

- Compared to the conventional checkpointing, FT-Pro can effectively reduce application completion time by avoiding anticipated failures through proactive migration and skipping unnecessary fault-tolerant requests through selective checkpointing.
- When both $precision$ and $recall$ are in the range of [0.6, 1.0], FT-Pro outperforms periodic checkpointing by over 10 percent; as long as both metrics are above 0.3, FT-Pro does better than periodic checkpointing.
- In general, a modest allocation of spare nodes—less than 5 percent—is sufficient for FT-Pro to achieve the above performance gain.

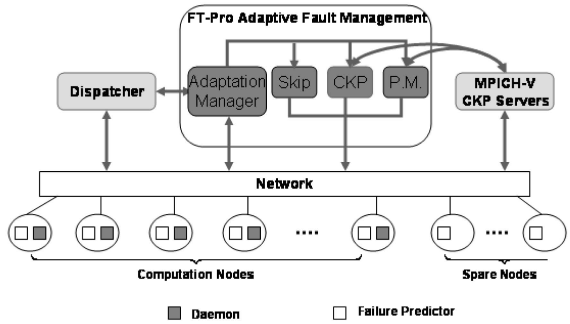


Fig. 8. Integrating FT-Pro with MPICH-VCL.

- To fully utilize failure prediction, the combination of failure avoidance and removing unnecessary fault-tolerant actions is of great importance for improving application performance.
- A more efficient migration support such as a live migration support can further improve the performance of FT-Pro.

6 Case Studies

In this section, we evaluate FT-Pro by using trace-based simulations. Application traces and a failure trace collected from production systems are used to investigate the potential benefit of using FT-Pro in realistic HPC environments.

We implement FT-Pro in the open source checkpointing package MPICH-VCL 0.76 [13]. Note that FT-Pro is independent of the underlying checkpointing tool and can be easily implemented with other tools such as LAM/MPI [14].

Fig. 8 illustrates our implementation. There are four major components:

1. *FT-Pro daemons*, which are colocated with application processes on computation nodes,
2. *the dispatcher*, which are responsible for managing computation resources,
3. *the adaptation manager*, which is in charge of decision making, as described in Section 4, and
4. *the CKP server*, which is used to perform coordinated checkpointing.

The migration support is based on the stop-and-restart model.

6.1 Methodology

The simulator is provided with a failure trace, an application trace, a computation scale N_W, and an interval I. Here, an application trace includes the application-failure-free execution time T_{appl} and fault-tolerant overheads such as C_{ckp} and C_{pm}. The details about the applications and the failure trace will be described in the following sections.

In the case of using periodic checkpointing, the application takes a coordinated checkpoint at a constant interval of I. In the case of using FT-Pro, a runtime decision is made at a constant time of I and the application takes an action from SKIP, CHECKPOINT, or MIGRATION according to the decision made by the adaptation manager. The outputs provided by the simulator are application completion times,

TABLE 4
Description of Parallel Applications

Application	Description
NPB [55] (class C)	BT, dominating with point-to-point communications CG, dominating with unstructured long-distance communications LU, involving the computation of implicit CFD with message transferring SP, solving non-diagonally dominant and scalar penta-diagonal equations
ENZO [56]	A parallel cosmology simulation code using SAMR algorithm
GROMACS [57]	A molecular dynamics code to study the evolution of interacting atoms

TABLE 5
Measured Operation Costs and Application Execution Times Using CKP

Appl.	N_W	CKP Image (MB)	C_{ckp} $(Sec.)$	C_{pm} $(Sec.)$	Appl.	N_W	CKP Image (MB)	C_{ckp} $(Sec.)$	C_{pm} $(Sec.)$
BT	9	171	111	168	CG	4	170	30	76
	16	100	81	146		8	90	33	51
	25	66	82	156		16	46	35	40
	36	48	88	195		32	25	37	56
	64	28	91	198		64	13	88	107
LU	4	168	61	84	ENZO	4	38	17	29
	8	87	36	64		8	32	15	28
	16	45	37	70		16	19	10	28
	32	24	33	79		32	14	22	49
	64	12	36	116		64	12	32	81
SP	9	125	76	122	GROMACS	4	10	6	11
	16	74	65	125		8	8	6	11
	25	48	62	126		16	7	12	32
	36	34	60	132		32	6	11	28
	64	21	61	145		64	6	25	70

i.e., T_{ckp} by using periodic checkpointing and T_{ft-pro} by using FT-Pro.

6.2 Parallel Applications

Six parallel applications, including parallel benchmarks and scientific applications, are tested in the study. They are the benchmark CG and three pseudoapplications (BT, LU, and SP) from NPB [55], the cosmology application ENZO [56], [58], and the molecular dynamics application GROMACS [57] (see Table 4). This test suite is from a mixture of scientific domains, thereby enabling us to have a fair evaluation of FT-Pro across a broad spectrum of HPC applications.

Application traces are collected on an IA32 Linux cluster at Argonne National Laboratory (part of the TeraGrid). The cluster consists of 96 nodes, each equipped with two 2.4-GHz Intel Xeon processors and 4G-Mbyte memory. All the nodes are connected via Gigabyte Ethernet. A 4-Tbyte storage is shared among the nodes via NFS. The operation system is SuSE Linux v8.1, and the MPICH-V is of version 0.76.

Table 5 lists the measured data. The data includes a single-process checkpoint image, the checkpointing overhead, and the migration overhead. Due to the special requirement on computation scale, the number of computation nodes used for BT and SP has to be in the form of N^2 (N is an integer).

According to the table, the size of a single-process checkpoint image decreases linearly with the increasing scale of computation. This is understandable due to the divide-and-conquer principle. An interesting feature is with C_{ckp}. It first drops and then starts to increase as the number of processors increases. This is caused by the increasing synchronization overhead by using coordinated checkpointing. It implies that process coordination can be a potential performance bottleneck when the computation scale is substantially large [5]. Migration cost C_{pm}, in general, increases with the growing computation scale. The main reason is that the stop-and-restart migration mechanism is used and the current MPICH_VCL device instantiates the processes in a sequential order. As shown in the table, generally, $C_{pm} \leq 3C_{ckp}$.

6.3 Failure Trace

Rather than using synthetic failure events, we use a failure trace collected from a production system at NCSA [27]. The machine has 520 two-way SMP 1-GHz Pentium-III nodes (1,040 CPUs), 512 of which are compute nodes (2-Gbyte memory), and the rest are storage nodes and interactive access nodes (1.5-Gbyte memory). Table 6 gives the statistics of the failure trace. We randomly select 96 nodes to match the testbed.

The trace-based simulator scans the failure trace in the time order and simulates a failure when a real failure entry is encountered. The prediction accuracy is emulated as follows:

1. *Recall.* If there exists a failure on a node between the current and the next adaptation point, the predictor reports a failure of its type with the probability of $recall$ on the node.
2. *Precision.* Suppose the predictor has totally reported x failures for the intervals with actual failures.

TABLE 6
Statistics of Failure Events

Failure Type	Percentage	Downtime (in hrs)
software	83%	0.7
hardware	1%	100.7
maintenance	16%	1.2

According to the definition of $precision$, for intervals without an actual failure, the predictor randomly selects $\frac{x\times(1-precision)}{precision}$ intervals and gives a false alarm on each of them.

6.4 Results

Table 7 lists our trace-based simulation results. Here, T_{appl} denotes the application execution time in a failure-free computing environment, and T_{ckp} and T_{ft-pro} represents the application completion times in the presence of failures by using periodic checkpointing and FT-Pro, respectively. We increase application-failure-free execution times to simulate long-running applications. In the case of using FT-Pro, an additional spare node is allocated to accommodate proactive actions. The parenthesized numbers in the table denote performance overheads (in percentage) on the application by using periodic checkpointing or FT-Pro; it is defined as $\frac{T_{ckp}-T_{appl}}{T_{appl}}$ when using periodic checkpointing and $\frac{T_{ft-pro}-T_{appl}}{T_{appl}}$ when using FT-Pro. Note that the performance overhead includes the application recovery time and the delay time caused by fault management.

As we can see in the table, the overhead caused by checkpointing is not trivial. For example, when the computing scale is 64, the extra overhead introduced by checkpointing is more than 50 percent for BT, SP, CG, and ENZO. In contrast, the performance overhead introduced by FT-Pro is usually less than 3 percent. Further, for both SP and ENZO, we observe that the application completion times on 64 computation nodes are longer than those on 32 nodes by using periodic checkpointing, whereas FT-Pro is able to reduce them as the computing scale grows. It implies that FT-Pro has better scalability.

Fig. 9 shows the $Time\_Reduction$ and $SU\_Reduction$ introduced by FT-Pro with these applications. It shows that in general, both metrics increase with the growing scale of computation. The larger the scale of an application is, the higher probability that it has to experience failures, thereby resulting in more opportunities for FT-Pro to improve its performance.

As presented in Figs. 9a and 9b, $Time\_Reduction$ is in the range of 2 percent-43 percent, depending on the applications and computation scales. The value is relatively small with GROMACS than with other applications. This is due to the use of a small-sized computation domain with GROMACS. As shown in Table 5, a small checkpoint image per process is observed with GROMACS, thereby reducing the potential gain that can be brought by removing unnecessary checkpoints by using FT-Pro.

According to Figs. 9c and 9d, when the computation scale is smaller than 10, FT-Pro may result in negative $SU\_Reduction$. A major reason is that the allocation of one spare node by FT-Pro is not trivial when the computation scale is small (e.g., four, eight, or nine). If the time reduction brought by FT-Pro is small, then the use of additional computing resources can overshadow its gain, thereby resulting in negative gain on $SU\_Reduction$. In general, FT-Pro provides positive results in terms of $Time\_Reduction$ and $SU\_Reduction$ when the computing scale is larger than 16.

We also plot the gain achieved through proactive migrations on these applications (see Fig. 10). Note that FT-Pro improves over checkpointing from two aspects: one is to avoid failures via preventive migrations, and the other is to skip unnecessary checkpoints. The figure only plots the first part, and the second part can be easily inferred from the figure. These results are consistent with those shown in Fig. 4, that is, failure avoidance through proactive migrations is the dominant factor for improvement. In general, more than

TABLE 7
Application Completion Times by Using FT-Pro and Periodic Checkpointing

Appl.	N_W	T_{appl} $(hours)$	T_{ckp} $(hours)$	T_{ft-pro} $(hours)$	Appl.	N_W	T_{appl} $(hours)$	T_{ckp} $(hours)$	T_{ft-pro} $(hours)$
BT	9	666	720.31 (8.2%)	675.23 (1.4%)	CG	4	657.81	708.32 (7.7%)	663.37 (0.8%)
	16	408	465.06 (14.0%)	413.58 (1.4%)		8	410.52	424.14 (3.3%)	412.59 (0.5%)
	25	286	374.17 (30.8%)	291.63 (2.0%)		16	290	319.19 (10.1%)	294.46 (1.5%)
	36	227	322.70 (42.2%)	230.22 (1.5%)		32	188	244.69 (30.2%)	190.63 (0.7%)
	64	166	269.71 (62.5%)	169.70 (2.2%)		64	128	236.28 (84.6%)	132.20 (3.3%)
LU	4	1625	1704.47 (4.9%)	1636.91 (1.0%)	ENZO	4	991	1014.22 (2.3%)	996.41 (0.5%)
	8	862	925.68 (7.4%)	867.64 (0.7%)		8	590	610.92 (3.5%)	592.93 (0.5%)
	16	528	622.70 (17.9%)	532.68 (0.9%)		16	320	374.13 (16.9%)	322.28 (0.7%)
	32	419	520.58 (24.2%)	422.78 (0.9%)		32	197	260.64 (32.3%)	199.37 (1.2%)
	64	350	502.12 (43.5%)	354.58 (1.3%)		64	169	302.67 (79.1%)	170.75 (1.5%)
SP	9	915	1000.00 (9.3%)	924.04 (1.0%)	GROMACS	4	4466	4934.17 (10.5%)	4848 (8.6%)
	16	592	673.36 (13.7%)	596.28 (0.7%)		8	2529	2592.60 (2.5%)	2537.78 (0.3%)
	25	409	488.17 (19.4%)	413.94 (1.2%)		16	1589	1702.12 (7.1%)	1595.61 (0.3%)
	36	293	383.63 (30.9%)	296.94 (1.3%)		32	1328	1506.12 (13.4%)	1337.72 (0.7%)
	64	259	412.79 (59.4%)	264.11 (2.0%)		64	2328	2711.71 (16.5%)	2348.84 (0.9%)

The parenthesized numbers in the table are performance overheads (in percentage) on the application by using periodic checkpointing or FT-Pro. The interval I is set to 0.56 hours. With FT-Pro, in addition to N_W, one spare node is allocated. Both $precision$ and $recall$ are set to 0.7.

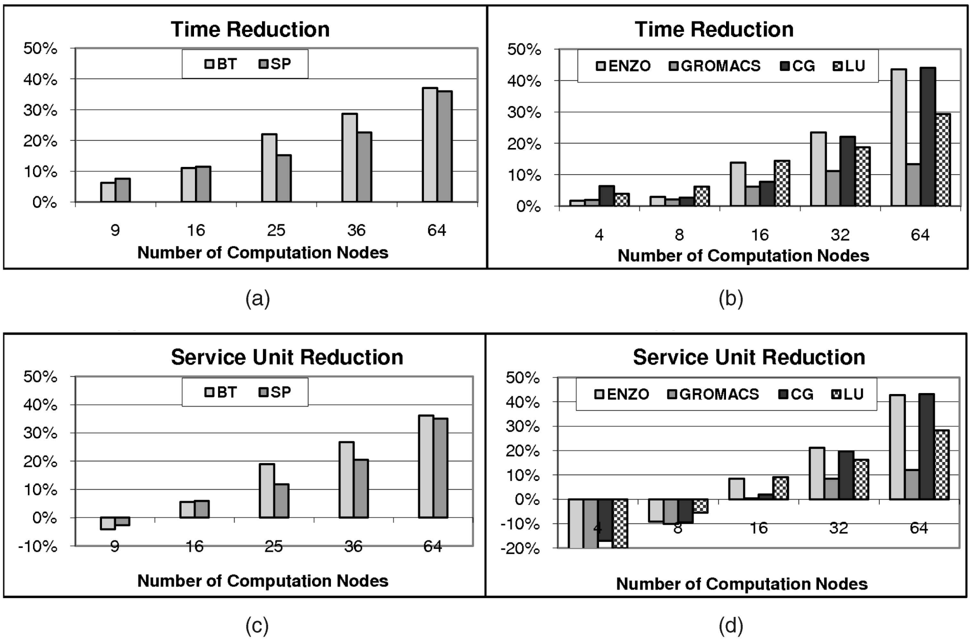


Fig. 9. $Time\_Reduction$ and $SU\_Reduction$ achieved by FT-Pro against periodic checkpointing. The performance gain achieved by FT-Pro increases as the size of computation increases.

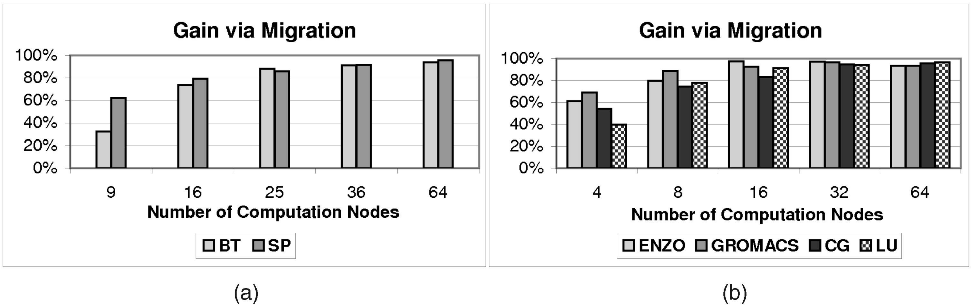


Fig. 10. Performance benefit achieved by FT-Pro through proactive migration (in percentage).

50 percent of performance gain is achieved by proactive actions, and this percentage is increased to nearly 100 percent when the computation scale is increased to 64. Again, it demonstrates that in order to effectively utilize failure prediction, proactive migration is indispensable for substantially improving application performance under failures.

We have also evaluated the performance of FT-Pro with these applications by changing spare node allocations and tuning prediction accuracies [59]. The results are similar to those shown in Section 5. For instance, when the computation scale is set to 64, by allocating one or two spare nodes, the relative gain achieved by FT-Pro over checkpointing is between 14 percent and 43 percent, and FT-Pro is more sensitive to false negatives.

6.5 Summary of Case Studies

In summary, our trace-based simulations with six different applications have shown that FT-Pro has the potential to reduce application completion times in realistic HPC environments. The results are consistent with those obtained by using stochastic modeling. Our studies show that the performance overhead caused by FT-Pro is very low (i.e., less than 3 percent). Further, FT-Pro can be easily integrated with existing checkpointing tools by adding the adaptation manager as a new module.

7 Conclusions

In this paper, we have presented an adaptive fault management approach called FT-Pro for parallel applications. An adaptation manager has been proposed to dynamically choose an appropriate action from SKIP, CHECKPOINT, and MIGRATION at runtime in response to failure prediction. We have studied FT-Pro under a wide range of parameters through stochastic modeling and case studies with parallel applications.

Experimental results demonstrate that FT-Pro can effectively improve the performance of parallel applications in the presence of failures. Specifically, 1) FT-Pro outperforms periodic checkpointing, when both $precision$ and $recall$ are greater than 0.3, 2) a modest allocation of spare nodes (i.e., less than 5 percent) is usually sufficient for FT-Pro to provide the aforementioned performance gain, and 3) the performance overhead caused by FT-Pro is very low, e.g., less than 3 percent on the applications tested.

Our study has some limitations that remain as our future work. First, we will investigate how to modify our algorithm to work with other checkpointing mechanisms such as log-based [4], [13] and live migration [19], [20], [44]. Second, we plan to provide a theoretic proof on the optimal allocation of spare nodes. Last, we are in the process of integrating our prediction work [11], [48], [49] with FT-Pro. Upon completion, we will evaluate it with parallel applications on production systems.

Appendix A
Description of FSPN Modeling

A.1 Subnet of Failure Behavior

We first describe failure behaviors on computation nodes. When the application starts, all the computation nodes are in the W_{up} state. A firing of the timed transition T_{prob} represents a failure arrival, and the corresponding node enters the vulnerable state W_{prob}. If the failure event is predicted via the transition T_{detect}, the node enters the state $W_{detected}$; otherwise, it enters W_{missed} via the transition T_{missed}. The node at $W_{detected}$ goes to W_{down} with a firing of T_{fail} if there are enough spare nodes available. The nodes at W_{missed} will eventually enter W_{down} via a deterministic transition T_{fail}. The crashed nodes at W_{down} recover back to W_{up} when $T_{noderepair}$ fires. The transition $T_{falsefail}$ simulates the false alarm behavior of the predictor. When it fires, the nodes at W_{up} enters $W_{falsedetected}$ and then automatically goes back to W_{up} via $T_{falsefail}$.

The spare nodes have similar state transitions, except that failures on spare nodes do not pose a direct performance penalty on the application.

A.2 Subnet of Adaptation Manager

We use the state P_{timer} and the deterministic transition $T_{timeout}$ to represent the adaptation interval. The firing of $T_{timeout}$ makes the subnet enter the $P_{decision}$ state, where FT-Pro makes a runtime decision. Upon invocation, the subnet enters one of the three states: 1) P_{skip} when T_{skip} fires, which means that a SKIP action is taken, and the subnet enters P_{skip} and immediately returns to the state P_{timer}, 2) P_{ckp} when T_{ckp} fires, which means that a CHECKPOINT action is taken, and the subnet waits for the firing of the timed transition $T_{checkpoint}$ (i.e., representing the checkpointing overhead) and then returns to P_{timer}, and 3) P_{pm} when T_{pm} fires, which means that a MIGRATION action is taken, and the subnet waits for the firing of the timed transition $T_{migrate}$ (i.e., representing the migration cost) and then returns to P_{timer}. Further, the firing of $T_{migrate}$ swaps vulnerable nodes at $W_{detected}$ and $W_{faseledetected}$ with the spare nodes at S_{up} and S_{missed}.

A.3 Subnet of Application Performance

In this subnet, we use fluid places to model the continuous quantities like time and workload. The transition T_{time} pumps fluid to the place P_{exec} with a constant rate of 1.0, representing the elapsed time. Similarly, T_{work} pumps fluid to the place P_{vol}, representing the accumulated volatile work. Through three inhibitor arcs, T_{work} is disabled if the subnet is at P_{ckp}, P_{pm}, or W_{down}. P_{ckp}, P_{pm}, and W_{down} represent the checkpointing overhead, the migration overhead, and the recovery cost. Through the impulse arcs to T_{fail}, the work at P_{vol} is flushed out to zero, representing the work loss due to failures. The work is flushed out to P_{saved} via the impulse arcs to $T_{migrate}$ or $T_{checkpoint}$, representing the work saved to a stable storage. Once the accumulated work at either P_{vol} or P_{saved} exceeds the application workload, T_{finish} fires, and the subnet enters P_{finish}. The fluid at P_{exec} is the application completion time.

Acknowledgments

The authors appreciate the valuable comments and suggestions from the anonymous referees. Many thanks are due to the members of the Scalable Computing Systems Laboratory at the Illinois Institute of Technology. This work was supported in part by US National Science Foundation Grants CNS-0720549, CCF-0702737, and NGS-0406328 and a TeraGrid Compute Allocation. Some preliminary results of this work were presented in [50] and [59].

References

[1] The Top500 Supercomputer Site, http://www.top500.org, 2007.
[2] D. Reed, C. Lu, and C. Mendes, "Big Systems and Big Reliability Challenges," *Proc. Int'l Conf. Parallel Computing (ParCo),* 2003.
[3] B. Schroeder and G. Gibson, "A Large Scale Study of Failures in High-Performance-Computing Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2006.
[4] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys,* vol. 34, no. 3, 2002.
[5] E. Elnozahy and J. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 2, Apr.-June 2004.
[6] V. Castelli, R. Harper, P. Heldelberger, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert, "Proactive Management of Software Aging," *IBM J. Research and Development,* vol. 45, no. 2, 2001.
[7] S. Chakravorty, C. Mendes, and L. Kale, "Proactive Fault Tolerance in Large Systems," *Proc. First Workshop High Performance Computing Reliability Issues (HPCRI),* 2005.
[8] R. Vilalta and S. Ma, "Predicting Rare Events in Temporal Domains," *Proc. IEEE Int'l Conf. Data Mining (ICDM),* 2002.
[9] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, and S. Ma, "Critical Event Prediction for Proactive Management in Large-Scale Computer Clusters," *Proc. ACM SIGKDD,* 2003.
[10] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Blue Gene/L Failure Analysis and Prediction Models," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2006.
[11] P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White, "A Meta-Learning Failure Predictor for Blue Gene/L Systems," *Proc. Int'l Conf. Parallel Processing (ICPP),* 2007.
[12] A. Oliner and J. Stearley, "What Supercomputers Say: A Study of Five System Logs," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2007.
[13] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "Mpich-V: A Multiprotocol Automatic Fault Tolerant MPI," *Int'l J. High Performance Computing and Applications,* 2005.
[14] J. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," *Proc. 10th European PVM/MPI Users' Group Meeting,* 2003.
[15] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2004.
[16] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proc. Usenix Winter Technical Conf.,* 1995.
[17] J. Duell, P. Hargrove, and E. Roman, "Requirements for Linux Checkpoint/Restart," Technical Report LBNL-49659, Berkeley Lab, May 2002.

[18] E. Gabriel et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," *Proc. 11th European PVM/MPI Users' Group Meeting,* 2004.

[19] C. Du and X. Sun, "MPI-Mitten: Enabling Migration Technology in MPI," *Proc. Sixth IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid),* 2006.

[20] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance," *Proc. 21st Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2007.

[21] J. Young, "A First Order Approximation to the Optimal Checkpoint Interval," *Comm. ACM,* vol. 17, no. 9, 1974.

[22] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, "Min-Max Checkpoint Placement under Incomplete Failure Information," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2004.

[23] S. Toueg and O. Babaoglu, "On the Optimum Checkpoint Selection Problem," *SIAM J. Computing,* vol. 13, no. 3, 1984.

[24] O. Babaoglu and W. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page Reference Bits," *Proc. Eighth Symp. Operating Systems Principles (SOSP),* 1981.

[25] J. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg, "On the Feasibility of Incremental Checkpointing for Scientific Computing," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2004.

[26] J. Plank, K. Li, and M. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 10, Oct. 1998.

[27] C.-D. Lu, "Scalable Diskless Checkpointing for Large Parallel Systems," PhD dissertation, Univ. of Illinois at Urbana-Champaign, 2005.

[28] G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," *Proc. IEEE Int'l Conf. Cluster Computing (Cluster),* 2004.

[29] B. Allen, "Monitoring Hard Disks with Smart," *Linux J.,* Jan. 2004.

[30] Hardware Monitoring by LM Sensors, http://secure.netroedge.com/-lm78/info.html, 2007.

[31] Health Application Programming Interface, http://www.renci.org, 2007.

[32] Intelligent Platform Management Interface, http://www.intel.com/design/servers/ipmi, 2007.

[33] K. Trivedi and K. Vaidyanathan, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," *Proc. 10th Int'l Symp. Software Reliability Eng. (ISSRE),* 1999.

[34] G. Weiss and H. Hirsh, "Learning to Predict Rare Events in Event Sequences," *Proc. ACM SIGKDD,* 1998.

[35] G. Hoffmann, F. Salfner, and M. Malek, "Advanced Failure Prediction in Complex Software Systems," *Proc. 23rd Int'l Symp. Reliable Distributed Systems (SRDS),* 2004.

[36] G. Hamerly and C. Elkan, "Bayesian Approaches to Failure Prediction for Disk Drives," *Proc. 18th Int'l Conf. Machine Learning (ICML),* 2001.

[37] J. Hellerstein, F. Zhang, and P. Shahabuddin, "A Statistical Approach to Predictive Detection," *Computer Networks: The Int'l J. Computer and Telecommunications Networking,* 2001.

[38] A. Gara et al., "Overview of the Blue Gene/L System Architecture," *IBM J. Research and Development,* vol. 49, nos. 2/3, 2005.

[39] Cray, Cray XT Series System Management, http://docs.cray.com/books/S-2393-15/S-2393-15.pdf, 2005.

[40] C. Leangsuksun, T. Liu, T. Raol, S. Scott, and R. Libby, "A Failure Predictive and Policy-Based High Availability Strategy for Linux High Performance Computing Cluster," *Proc. Fifth LCI Int'l Conf. Linux Clusters,* 2004.

[41] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, and A. Sivasubramaniam, "Fault-Aware Job Scheduling for Blue Gene/L Systems," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2004.

[42] Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. Sahoo, "Performance Implications of Failures in Large-Scale Cluster Scheduling," *Proc. 10th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP),* 2004.

[43] T. Tannenbaum and M. Litzkow, "Checkpointing and Migration of Unix Processes in the Condor Distributed Processing System," *Dr. Dobbs J.,* Feb. 1995.

[44] C. Clark et al., "Live Migration of Virtual Machines," *Proc. Second Symp. Networked Systems Design and Implementation (NSDI),* 2005.

[45] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS),* 2006.

[46] G. Brown, D. Bernard, and R. Rasmussen, "Attitude and Articulation Control for the Cassini Spacecraft: A Fault Tolerance Overview," Jet Propulsion Laboratory technical report, 1997.

[47] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker, "Total Recall: System Support for Automated Availability Management," *Proc. First Symp. Networked Systems Design and Implementation (NSDI),* 2004.

[48] Z. Lan, P. Gujrati, Y. Li, Z. Zheng, R. Thakur, and J. White, "A Fault Diagnosis and Prognosis Service for Teragrid Clusters," *Proc. Second TeraGrid Conf.,* 2007.

[49] Z. Zheng, Y. Li, and Z. Lan, "Anomaly Localization in Large-Scale Clusters," *Proc. IEEE Int'l Conf. Cluster Computing (Cluster),* 2007.

[50] Y. Li and Z. Lan, "Exploit Failure Prediction for Adaptive Fault-Tolerance in Cluster Computing," *Proc. Sixth IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid),* 2006.

[51] J. Plank and M. Thomason, "Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems," *J. Parallel and Distributed Computing,* vol. 61, no. 11, 2001.

[52] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative Checkpointing Theory," *Proc. 20th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2006.

[53] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri Net Package," *Proc. Third Int'l Workshop Petri Nets and Performance Models (PNPM),* 1989.

[54] L. Wang, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2005.

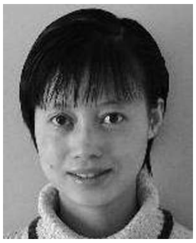
[55] NASA NAS Parallel Benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html, 2007.

[56] G. Bryan, T. Abel, and M. Norman, "Achieving Extreme Resolution in Numerical Cosmology Using Adaptive Mesh Refinement: Resolving Primordial Star Formulation," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2001.

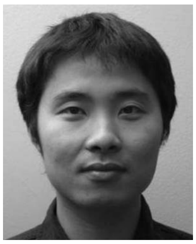
[57] H. Berendsen, D.V. der Spoel, and R. van Drunen, "Gromacs: A Message-Passing Parallel Molecular Dynamics Implementation," *Computer Physics Comm.,* vol. 91, pp. 43-56, 1995.

[58] Z. Lan, V. Taylor, and G. Bryan, "Dynamic Load Balancing for Structured Adaptive Mesh Refinement Applications," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2001.

[59] Y. Li and Z. Lan, "Using Adaptive Fault Tolerance to Improve Application Robustness on the Teragrid," *Proc. Second TeraGrid Conf.,* 2007.



Zhiling Lan received the BS degree in mathematics from Beijing Normal University in 1992, the MS degree in applied mathematics from the Chinese Academy of Sciences in 1995, and the PhD degree in computer engineering from Northwestern University in 2002. She has been an assistant professor in the Department of Computer Science, Illinois Institute of Technology, Chicago, since 2002. Her research interests are in the area of parallel and distributed systems, in particular, fault tolerance, dynamic load balancing, and performance analysis and modeling. She is a member of the IEEE Computer Society.



Yawei Li received the BS and MS degrees from the University of Electronic Science and Technology of China in 1999 and 2002, respectively. He is currently a PhD candidate in the Department of Computer Science, Illinois Institute of Technology, Chicago. He specializes in parallel and distributed computing and scalable software systems. His current research focuses on adaptive fault management in large-scale computer systems, checkpointing optimization, and load balancing in the Grid environment. He is a student member of the IEEE.