Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery

Elmootazbellah N. Elnozahy, *Member*, *IEEE*, and James S. Plank, *Member*, *IEEE Computer Society*

Abstract—Over the past two decades, rollback-recovery via checkpoint-restart has been used with reasonable success for long-running applications, such as scientific workloads that take from few hours to few months to complete. Currently, several commercial systems and publicly available libraries exist to support various flavors of checkpointing. Programmers typically use these systems if they are satisfactory or otherwise embed checkpointing support themselves within the application. In this paper, we project the performance and functionality of checkpointing algorithms and systems as we know them today into the future. We start by surveying the current technology roadmap and particularly how Peta-Flop capable systems may be plausibly constructed in the next few years. We consider how rollback-recovery as practiced today will fare when systems may have to be constructed out of thousands of nodes. Our projections predict that, unlike current practice, the effect of rollback-recovery may play a more prominent role in how systems may be configured to reach the desired performance level. System planners may have to devote additional resources to enable rollback-recovery and the current practice of using "cheap commodity" systems to form large-scale clusters may face serious obstacles. We suggest new avenues for research to react to these trends.

Index Terms—Distributed systems, distributed applications, performance of systems, fault tolerance, modeling techniques, reliability, availability, serviceability, measurement, evaluation, modeling, simulation of multiple-processor systems.

1 Introduction

Rollback-recovery has been particularly popular in high-performance computing systems, where applications run from hours to weeks. In this paper, we leap forward to the end of the decade and attempt to project the future of rollback-recovery in light of current technology trends. The goal of this study is twofold, one to predict how rollback-recovery as we know it today may fare in the future within the context of large-scale systems with increasing scalability demands, and another to identify potential problems and research areas. To provide a context for the study, we propose to study rollback-recovery in a "notional" system that could deliver one Peta Flop per second (PF/s) throughput circa 2010. Many researchers in the high-performance computing area consider such a performance level as the next grand challenge.

A system that delivers 1PF/sec will be a challenge to build because of projected technology trends, which stipulate that the continuous improvement in semiconductor technology that the industry enjoyed over the past 40 years may slow down [13]. The so-called Moore's law about the doubling of chip density every 18 months [16] may not apply in the future due to the difficulties of powering, cooling, and manufacturing denser chips. These difficulties may limit the harnessed performance that can be obtained from a single processor. Moreover, shrinking the devices in size implies transistors may become more susceptible to soft error rates due to cosmic rays. Therefore, system performance could suffer further as designers deploy more rigorous mechanisms to harden the circuits from which future computers will be built, thereby slowing down the basic devices or diverting away resources that otherwise could be used to boost performance [10].

Conducting a study like this is difficult as there are many unknowns, and predicting basic system performance has a lot of caveats. In fact, some may argue that the systems of the future may not look at all like the systems of today and ambitious performance goals may require fundamentally different architectures and system structures. We do not wish to enter into such arguments, and we note that software inertia will inevitably require backward compatibility with existing system structures. Therefore, we consider an evolution of systems as we know them today.

We have selected coordinated checkpointing style of rollback-recovery as the basis for our study. This selection is motivated by the nature of high-performance computing applications, which often follow the data parallel approach in which *all* available processing nodes partition the work on a data set and cooperate via message passing to synchronize. A popular form of synchronization in these applications is to use barriers followed by data exchange messages between consecutive iterations of compute-intensive loops. This style of structuring applications may render ineffective more efficient checkpointing protocols that exploit application communication patterns to limit the number of processing nodes that need to participate in a coordinated checkpoint [14].

We use validated analytic and simulation models to predict the requirements and performance of rollback-recovery in a "notional" system that delivers a 1PF/sec

- *E.N. Elnozahy is with IBM Research, 11501 Burnet Rd., Austin, TX 78750. E-mail: mootaz@us.ibm.com.*
- *J.S. Plank is with the Department of Computer Science, University of Tennessee, 203 Claxton Complex, 1122 Volunteer Blvd., Knoxville, TN 37996-3450. E-mail: plank@cs.utk.edu.*

Manuscript received 15 May 2004; revised 26 July 2004; accepted 10 Aug. 2004.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0083-0604.

TABLE 1
Summary of Our Conclusions

System node Size	Scalability required	Resilience to transient failures	Nodes devoted to spare cycles	Resilience to permanent failures	Relative cost for 1PF/sec deployment
Large	100's	Best	Few	Poor	Highest
Mid size	1000's	Good	Average	Acceptable	High
Small	10000's	Acceptable	Average	Good	Moderate
Smallest	100000's	Poor	Many	Best	Cheapest

system, using different plausible configurations that feature different node sizes. Our study uses different checkpointing intervals, Mean Time to Failure (*MTTF*), and Mean Time to Repair (*MTTR*). It suggests that the current practice of using small, "cheap commodity" nodes to build clusters of supercomputers may break down because the large scale deployment may cause a very large number of failures. We argue also that building such clusters of "expensive" very large-scale shared memory systems (e.g., 1,024-way Non-Uniform Memory Access (NUMA) systems) may not be resilient toward some types of hardware failures. A good trade off appears to be in the use of small to mid-size Symmetric Multiprocessors (SMP) systems. Regardless of the size of the node, however, we argue that reliability requirements in the future may increase to the point where some system resources may be devoted to providing spare cycles to account for the overhead of rolling back and restarting. The conclusions of the study are summarized in Table 1.

The rest of the paper provides the study, its conclusion, and avenues for future research. In Section 2, we review the current state of the art in rollback-recovery. In Section 3, we discuss future technology trends and their effects and present the bulk of the paper—the simulation study. We summarize our conclusions and discuss avenues for future research in Section 4.

2 Rollback-Recovery in Distributed Systems

Rollback-recovery is suitable where system availability requirements can tolerate the outage of computing systems during recovery. It offers a resource-efficient way of tolerating failures compared to other techniques such as replication or transaction processing. For the purpose of this paper, we assume fail-stop failure semantics in which a failed host ceases to operate without sending malicious messages [22]. One may argue that this assumption may not be realistic in the future in light of the projected increases in soft error rates and that more latent errors should be expected in systems by that time. We believe, however, that market expectations, as well as the essential requirement of software backward compatibility, will mandate that the reliable system abstraction that has characterized modern computing systems will be preserved. Therefore, we expect that system designers will opt for hardening system components at the expense of reduced performance in order to maintain the reliable abstraction to which computer users and programmers are accustomed.

We assume a distributed system consisting of a collection of application processes that communicate through a network that does not partition. The processes have access to a stable storage device that persists throughout all tolerated failures. Such a device may be built by a distributed storage farm, such as a future version of System Area Network (SAN) technology, providing the necessary bandwidth and scalability required by the system. Processes achieve fault tolerance by using this device to save recovery information in checkpoints periodically during failure-free execution. Upon a failure, a failed process uses the saved checkpoints to restart the application.

2.1 Rollback-Recovery in Distributed Systems

In a message-passing distributed system, messages induce interprocess dependencies during failure-free operation. Upon a failure of a process (or more), these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called rollback propagation [8]. To see why rollback propagation occurs, consider the situation where a sender of a message m rolls back to a state that precedes the sending of m. The receiver of m must also roll back to a state that precedes m's receipt; otherwise, the states of the two processes would be inconsistent because they would show that message m was received without being sent, which is impossible in any correct failure-free execution. Under some scenarios, rollback propagation may extend back to the initial state of the computation, losing all the work performed before a failure. This situation is known as the domino effect [21].

Broadly classified, rollback-recovery falls into two categories, one that uses checkpointing only and another that combines checkpointing with message logging [8]. In the first category, there are three styles of checkpointing, namely, *independent*, *coordinated*, and *communication-induced*. In independent checkpointing [4], each process is allowed to take its checkpoints independently, regardless of the dependencies that arise among processes due to message exchange. This style of checkpointing is susceptible to the domino effect, making it undesirable in practice. In contrast, coordinated checkpointing requires the processes to coordinate their checkpoints in order to save a system-wide consistent state [7]. This consistent set of checkpoints can then be used to bound rollback propagation. We have found that this technique is the one of choice in practical systems due to the simplicity of recovery. The scalability of coordinated checkpointing, however, is a concern since it may force all processes to take a checkpoint concurrently. Several studies have suggested variations on this technique to limit the number of processes that must participate in a coordinated checkpoint by exploiting the message passing

patterns of the application [14]. Nevertheless, even these variations may reduce to coordinated checkpointing, depending on the application, which is often the case in practice. Alternatively, communication-induced checkpointing forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes [5]. Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect. This technique, however, may record checkpoints that will never be used in any future rollback-recovery [11], and the number of checkpoints depends on the message passing pattern of the application. These properties lead to unpredictable checkpointing rates, making it difficult to use such techniques in practice [1].

The second category of rollback-recovery combines checkpointing with logging of nondeterministic events. This category relies on the *piecewise deterministic* (*PWD*) assumption [24], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged. By logging and replaying the nondeterministic events in their exact original order, a process can deterministically recreate its prefailure state even if this state has not been saved. Many algorithms to implement various forms of logging have been studied, and a full survey exists elsewhere [8].

For the purpose of this study, we focus on coordinated checkpointing. We believe that this is a reasonable focus given that this is the method currently being used in practice, e.g., supercomputing clusters. Independent checkpointing and communication-induced checkpointing do not seem to have caught much following in the user community due to various complications in real implementations. Furthermore, log-based recovery has not been in wide use due to the resource requirements imposed by message logging in message-passing systems and other considerations [12]. These logs tend to consume a substantial fraction of resources in main memory, stable storage, and network bandwidth.

2.2 Implementation Flavors

We consider three ways to implement checkpointing in practical systems, namely, by the application, by the operating system, or the by the compiler. Application-level implementations require the programmer to identify points in the program at which checkpoints are taken [3]. The programmer chooses points within the execution of the application such that the collections of checkpoints taken by all processes will yield consistent checkpoints. Furthermore, the programmer needs to implement the checkpointing and recovery code, including the decision of which data structures need to be stored on stable storage at each checkpoint. The recovery code reads the stored data structures in the checkpoints and reconstructs the connections among the processes in the application. Checkpointing implementation at the application level has several advantages and disadvantages. The programmer exploits knowledge about the application to insert checkpoints at the points in the execution where the amount of data to be stored is small. The programmer also has flexibility in controlling the rate of checkpointing. The principal disadvantage of this technique is the involvement of the programmer. Including checkpointing and recovery code in the application increases the programmer burden and reduces productivity. Furthermore, error-prone decisions about what to checkpoint entail the dangerous proposition that a bug in the checkpointing or recovering code may prevent recovery. Finally, the programmer must include in the recovery code routines to recreate the operating system's state that supports the application upon recovery from failure.

Checkpointing can also be implemented by the operating system [17]. The operating system effectively makes a conservative decision of saving all application's state, including dead variables that may not be used any further. Multihost coordination can be implemented by the operating system or by system services, and it has been shown that it can be implemented efficiently in earlier prototypes [9]. Additional techniques can be used by the operating system to reduce the overhead. For example, copy-on-write state capture [2] can be used to implement a version of nonblocking checkpointing that allows the application to proceed in parallel with saving the checkpointing. This technique has been implemented in a publicly available library that supports checkpointing at a very low performance overhead [18]. Other techniques include the use of incremental checkpointing, where the operating system uses the memory management subsystem to decide which data change between consecutive checkpoints [9]. Other techniques for reducing the amount of state that needs to be saved have also been studied [19].

A third alternative is to rely on the compiler to support the checkpointing implementation [15]. In this variation, the compiler performs data and control flow analysis on the program and inserts the checkpoints at points where the amount of data to be saved is small. Also, the compiler can decide which data structures are to be saved based on the data use analysis. Dead variables, for example, can be safely ignored by the compiler. Compared to the other two techniques, compiler-assisted checkpointing has advantages and disadvantages. Like operating system checkpointing, it operates transparently without burdening the programmer with the error-prone and tedious tasks of maintaining the recovery code within the application. It also is safer than programmer-assisted checkpointing since compiler analysis tends to be conservative, eliminating the case where an important data structure may be omitted from a checkpoint by way of error. Unlike operating system-assisted checkpointing, however, it cannot recover the system state directly.

2.3 State of the Art

Several production systems today use checkpointing to implement rollback-recovery, e.g., IBM's AIX operating system. These tend to be deployed in large-scale parallel processing environments, such as those used in high-performance and technical computing. Typically, coordinated checkpointing is implemented at the application level for those applications that run for hours or days. The

TABLE 2
Different Configurations for a Notional PF/Sec System

Node type	Processors in node	Node performance	Number of nodes N
Large-Scale NUMA	1000	10TF/sec	100
Large SMP	25	250GF/sec	4000
Small SMP	4	40GF/sec	24000
System-on-a-chip	1	10GF/sec	100000

checkpointing interval is usually measured in hour fractions (e.g., every 30 minutes) and, therefore, the performance impact of checkpointing is mitigated over the long time it takes to run the application. It has been observed that the bottleneck tends to be in the bandwidth to stable storage, which is also used for regular I/O access.

Publicly available libraries for checkpointing also exist, but they are not maintained at the level that is required to support production use. As a result, checkpointing code tends to be application-specific and nonportable. Lack of standards and checkpointing interfaces is a problem that hampers programmer's productivity in this area. We also note that there are no prototypes for compiler-assisted checkpointing available for production systems.

3 Projections

3.1 Technology Trends

The computing industry has enjoyed a long time of uninterrupted scaling of semiconductor technology. This growth has allowed higher levels of integration and speeds, consequently improving the performance of computing systems to the point that they have become an integral part of our modern infrastructure. The historical growth has followed the famous Moore's law, which stipulates that advances in lithography will lead to doubling chip density every 18 months. System clock frequency also followed this trend. This phenomenal growth, though, seems to be slowing down. Moreover, it is now feared that soon the growth will not be sustainable because foreseeable trends in cooling technologies and device fabrication cannot seem to support further increases in chip density. Therefore, the increasing fabrication problems and soft-error rates will require chip designers to deploy more robust mechanisms at the circuit level to detect and mask these errors at the same coverage rate as in today's systems. Performance will be reduced as more resources are devoted to ensuring reliable operation.

The combined effects of the problems aforementioned will yield less performance per chip than would be available following the traditional or historical improvement of performance over time. Therefore, more chips will need to be networked together to keep the performance trajectories according to historical trends. This will impose more demand for scalability on rollback-recovery algorithms and implementations.

3.2 Checkpointing in Peta-Scale Systems: A Simulation Study

In this section, we present a study of the scalability of coordinated checkpointing in future systems. We have selected coordinated checkpointing for its simplicity and because it is the current method of choice in real systems. For peta-scale system implementations, however, we project that the checkpoints will be taken locally in each node, then stored into a highly available stable storage system in a staggered fashion to avoid the bottleneck resulting from all nodes "ganging up" on the stable storage device [26].

To provide a context for the study, we consider various configurations for a notional Peta-Flop/sec system. Considering that today's processors have a typical performance of about 1GF/sec after allowing various inefficiencies in software and hardware, we project a 10-fold improvement in performance by the end of the decade due to technology. This is a fraction of the 16GF/sec that one would expect following the traditional technology trajectory. This conservative projection accounts for the technology slowdown that we expect as discussed previously. Considering such a building block, several configurations are possible to build a 1PF/sec system as shown in Table 2.

The table shows typical clusters using different system sizes. The top configuration corresponds to a large shared-memory system of about 1,000 processor chips used as a single node, typically using NUMA technology. One hundred such nodes are connected by a cluster to deliver a PF/sec system-wide performance. The second row corresponds to a smaller node of about 25-way Symmetric Multiprocessor System, while the third row shows a more popular 4-way SMP and the last one a single-chip "system-on-a-chip" that will be typically the "commodity" systems of the future. Traditionally, cost has been proportional to the node size and, therefore, we consider the system in the first row as the most expensive, while the fourth row as the cheapest. The number of nodes N for each configuration is shown in the table. We define N as the number of nodes in the system and consider a node as the unit of failure. We further make the assumption that the mean time to failure is independent of the node size. This assumption is consistent with what we have observed in actual installations, where failures are mostly due to operator and software errors. One may argue that this is not a reasonable assumption given that a large NUMA machine, for instance, may have more hardware components and, therefore, is likely to fail more often. We contend, however, that, even though such a

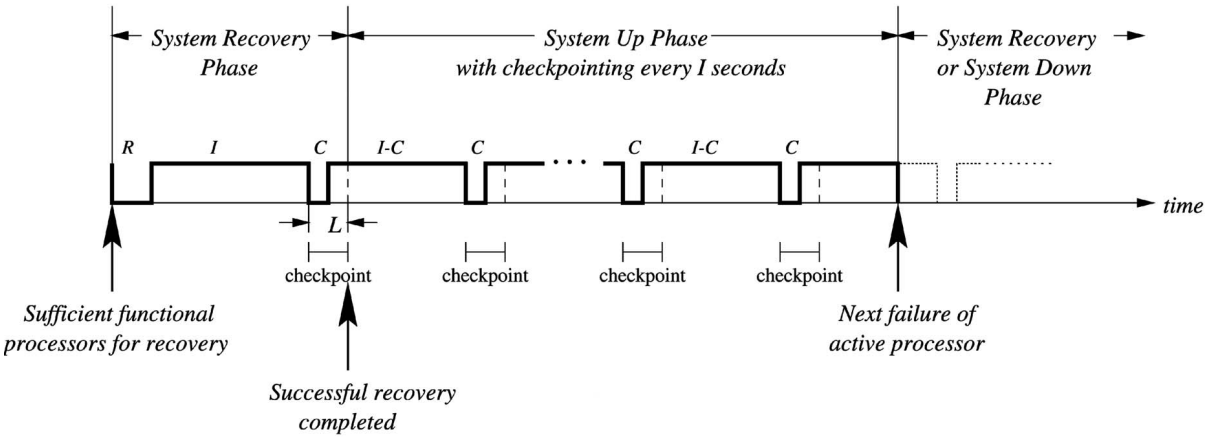

Fig. 1. A segment of computation, from recovery to failure of an active node. The vertical axis denotes progress in the computation.

machine may have a larger number of components than a low-end machine, more reliability features are added to ensure that the combination of many components does not bring down the reliability of the system.

To simulate coordinated checkpointing on a cluster of N processing nodes, we adopt the standard methodology and nomenclature of [25]. A node may be in one of two states: functional, meaning it is available for use, or failed, meaning that it is unavailable for use. Failed nodes may be repaired and become functional. Our computation proceeds on a fixed number, $a \leq N$, of *active* nodes. The remaining $(N - a)$ *spare* nodes will be held in reserve to be available for use upon the failure of an active node.

The system is functional when there are at least a active nodes. Upon a failure of one or more nodes, nodes from the spare pool are redesignated as active. All active nodes then restore the last committed checkpoint. The time that this takes is denoted by R. Once recovery is complete, the nodes resume normal operation. We also assume that a coordinated checkpoint is taken every I seconds after computation begins until an active node fails. Coordinated checkpointing algorithms are numerous, and a summary is available elsewhere [8]. The checkpoint latency, L, is the time that it takes for the checkpoint to be available for recovery and includes all phases of checkpointing, such as synchronization, logging of outstanding messages, state saving, and garbage collection. The overhead of the checkpoint, C, is the time that it takes from the computation. Standard checkpointing techniques, such as copy-on-write [2], [17], allow a large amount of checkpointing to proceed concurrently with the running application, so C is often quite a bit less than L.

When one of the active nodes fails, the computation stops and, if there are a remaining functional nodes, then recovery begins anew. If there are fewer than a remaining functional nodes, then the system remains idle until nodes are repaired, and a nodes are functional again.

A segment of such a computation is denoted in Fig. 1. The computation cycles through three distinct phases: 1) a System Recovery phase, which starts with recovery from a checkpoint, and finishes when the first checkpoint completes; 2) a System Up phase, where the nodes periodically checkpoint themselves every I seconds; 3) a System Down phase, where there are fewer than a functional nodes available for computation. A failure of an active node will take the system in the Recovery or Up phases either to the Down phase (if there are no longer a functional nodes) or to the Recovery phase (if there are still a functional nodes). A repair of a failed node will take the system from the Down phase to the Recovery phase when there are a functional nodes following the repair.

We define the concept of Useful Work (W), which is the time that the system spends on the computation that will not be lost following a failure. In the System Recovery phase, this is I seconds of computation. In the System Up phase, it is $(C - I)$ seconds per completed checkpoint. There is none in the System Down Phase. The ratio of useful work to total time is the Useful Work Factor, U, which indicates how much of the a active nodes' time is devoted to move the computation forward.

To summarize, we use the following definitions:

- C: The overhead per checkpoint.
- L: The checkpoint latency.
- R: The recovery time from a checkpoint.
- $MTTF$: The mean time to failure of a processing node.
- $MTTR$: The mean time to repair of a processing node.
- N: The total number of processing nodes in the system as defined above.

Failures and repairs are assumed to be independent and exponentially distributed. The user selects a and I to optimize performance. Both values involve trade offs in performance: Larger values of a mean more processing power devoted to advancing the computation; however, they also mean that there are fewer spares, exposing the system to the possibility of having more Down phases. Similarly, larger values of I mean that the system spends less time checkpointing and, therefore, may have less induced overhead. However, they also mean that the system may lose more work as a result of a failure. We have written a stochastic simulator which simulates failures and repairs coming from their respective distributions and runs until the Useful Work factor converges on a value to a

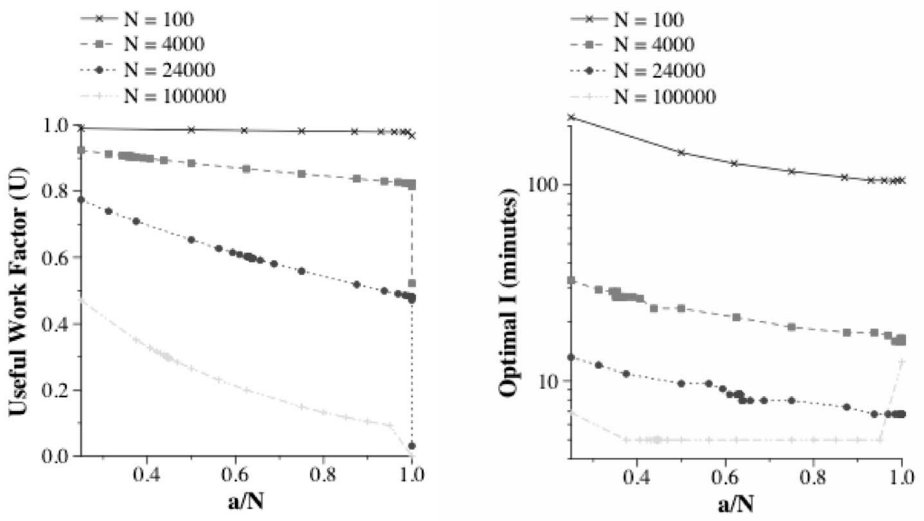

Fig. 2. Useful work factor as a function of N and a/N and the optimal checkpoint interval I as a varies from $N/4$ to N.

given tolerance. All data presented in this section come from the simulator.

3.2.1 Performance as the Number of Nodes Scales

For the first experiment, we selected the following parameters for study: Given the parameters of Peta-Scale computing defined above, C is set to one minute, L and R to 5 minutes. These are reasonable values for checkpointing the memory state and assume advanced checkpointing implementations such as incremental and copy-on-write [2], [9], [17]. There are no published studies about node failure rates for machines such as these. However, researchers at Los Alamos National Lab have anecdotally reported that the 1,024-node ASCI-Q supercomputer stays up for roughly 6 hours at a time. This corresponds to a single-node MTTF of 256 days, when failures are independently and exponentially distributed. Rather than work in units of 256 days, we selected an MTTF of one year per node, reflecting a slight improvement in operator, hardware, and software qualities for years to come. We selected an MTTR of one hour, a conservative estimate of the time to reset a failed node on a complex system. We also highlight the four configurations that we described in the previous section, namely, for $N = 100$, $4,000$, $24,000$, and $100,000$, although we ran the simulation for many intermediate points of N that we use to plot the figures below and to study if there are any discontinuities in the graphs.

We first present the results of these simulations in Fig. 2. We vary a from $N/4$ to N and plot the best U for each value of a. To determine the best value of U, we perform a binary search on the checkpoint interval, from its minimum value ($L = 5$ minutes) to a maximum value, refining at each iteration. The figure also plots the optimal I for each configuration under study. Two trends emerge from these figures. First, as N and a get larger, U decreases. This is because there are more nodes that can fail and trigger System Recovery. Also, U decreases sharply when a approaches N. This is because more time is spent in the System Down phase, where no spare nodes are available for recovery. Finally, when $N = 100,000$ and $a > 40,000$, the optimal checkpoint interval becomes equal to L, meaning the system is constantly checkpointing. The reason for this behavior is that the mean time to the first failure among a nodes is less than (1 year)/(40,000) = 13 minutes, which is less than the 15 minutes that it takes to get out of the System Recovery Phase.

The Useful Work Factor impacts the performance of the application in the following way: Suppose we characterize the application according to Amdahl's Law—its running time is composed of a serial portion S and a parallelizable portion P. Thus, when running on a nodes with no checkpointing and no failures, the running time is $S + P/a$. Let the running time of the program be normalized on one node so that $S + P = 1$ or, alternatively, $P = (1 - S)$. Then, the speedup of a program is defined as:

$$\text{speedup}(S, a) = 1/(S + (1 - S)/a) = a/((a - 1)S + 1).$$

The above equation does not take failures or checkpointing into account. To do so, we determine the Useful Work Factor U and multiply it by the speedup: $\text{speedup}(S, a, U) = U\ \text{speedup}(S, a)$. This is the expected speedup of the program over its optimal serial performance when employing a active nodes with checkpointing.

Fig. 3 displays the results of applying these calculations to our simulation scenario. For each value of N displayed and for four values of S (0, 0.0001, 0.001, 0.01), we determine the value of a that maximizes $\text{speedup}(S, a, U)$. This determination was done using a second binary search—for each potential value of a (starting with 1 and N as the endpoints), the value of I which maximizes U is determined. Then, the binary search proceeds in refining values of a which maximize $\text{speedup}(S, a, U)$. The left side of Fig. 3 shows the best speedup attained in this way, and the right side shows the best values of a.

The main feature of this figure is that, for each value of S, there comes a point beyond which adding more nodes does not improve the performance of the computation. There are two potential reasons. First, the computation may not scale past a certain point. Second, the failure rate becomes too high relative to the costs of checkpointing.

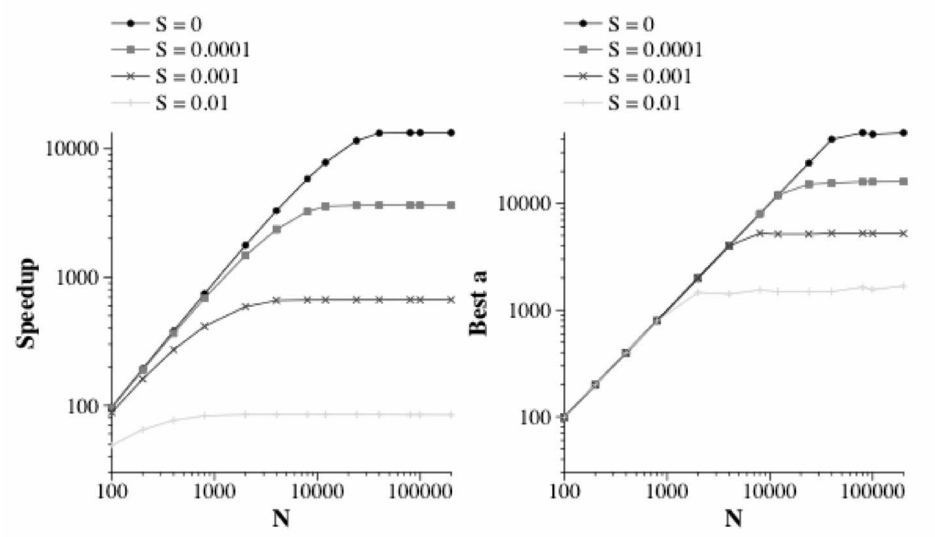

Fig. 3. Speedup for different degrees of parallelization.

To explore this further, we plot the same results in a different way in Fig. 4. Here, the speedup is presented as a percentage of the optimal speedup, $1/(S + (1 - S)/N)$. What is interesting about this graph is that for all values of $S > 0$, the percentage speedup follows the percentage for $S = 0$ very closely, until N increases past the natural scaling limit for each value of S. For $S = 0$, the percentage speedup decreases rapidly as N increases past 10,000.

There are two important implications on the system structure that the study reveals:

- There is a certain minimum performance that must be expected from each node, below which it may not be feasible to build a system with an overall performance that reaches a desired 1PF/sec. In the study, the minimum workable configuration is a 4-way SMP corresponding to $N = 4,000$. Going with cheaper nodes (single chip systems) may not work since about half the nodes must be used as spares, and the system may be continuously checkpointing. This is not likely to work in practice.
- It is important that spares are taken into account when configuring a system. We often find that systems are procured with the assumption that all nodes will be up all the time doing useful work. In the future, because of the large scale that will be required to reach the desired level of performance, a substantial number of cycles will be devoted to rolling back. Note that, in a real installation, it will be unlikely that the spares will be left unused while the system is in its Up phase, according to our simplifying assumption. However, it will be important to understand the effect of rollback-recovery on the system and there should be an allocation of resources that will count toward the cycles needed for rollback-recovery. This will depart from current practice and expectations.

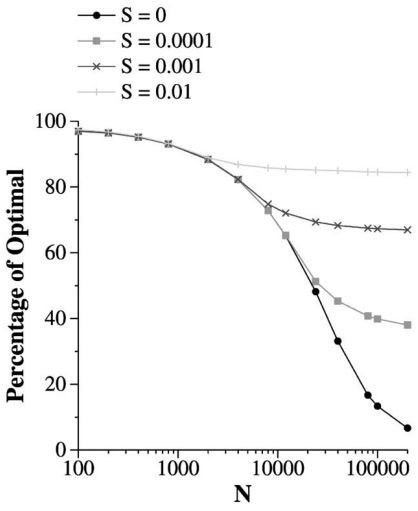

Fig. 4. Speedup, presented as a percentage of optimal speedup.

3.2.2 The Effect of Modifying C, L, R, $MTTF$, $MTTR$

In this section, we present the results of modifying some of the fixed parameters of the previous section to assess their impact on performance. All other parameters are held constant and have the same values as in the previous section. In the first test, we set C to zero and L and R to 10 microseconds. The purpose of this test is to project what happens when the performance impact of checkpointing and recovery is negligible. These parameters effectively describe a system where node failures are detected immediately and where a spare node substitutes a failed one with zero downtime. The results are in Fig. 5. Not surprisingly, when the overheads of checkpointing and repair are negligible, the system is able to attain nearly optimal performance. The reason is that, as long as the system does not enter the Down Phase (fewer than a nodes available), the useful work is nearly 100 percent. It should be noted that, even in this case, we do not attain perfect performance. Fig. 6 provides detail—in this figure, we plot a/N as a function of N for the case when $S = 0$. As the figure shows, although a is close to N, it is still necessary to

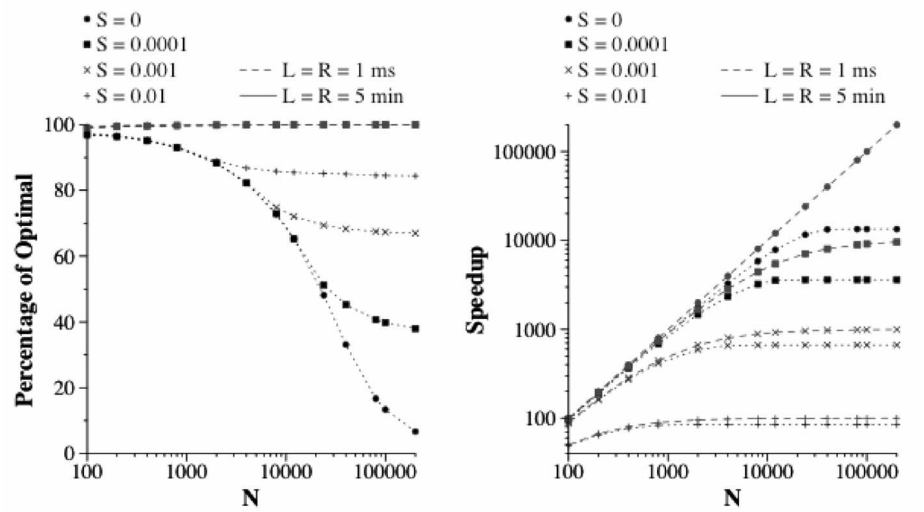

Fig. 5. Speedup when the overheads of checkpointing and repair are negligible.

have some spare nodes to prevent the system from entering the System Down Phase.

Modifying C. In Fig. 7, we display the effect of modifying C to 0 minutes and 5 minutes (while keeping L and R at 5 minutes each). As one would expect, the performance improves when C is reduced and gets worse when C is increased. Interestingly, though, the improvement is not drastic (as when C, L, and R are all reduced to negligible quantities). The reason is that the latency and recovery time are both responsible for nonuseful work being performed, especially in the initial interval following recovery, even when the checkpointing overhead is zero.

Modifying L **and** R. In Fig. 8, we display the results of modifying L and R to 70 seconds (just 10 seconds more than the checkpoint overhead), and to 10 minutes. Fig. 8 is quite interesting compared to Fig. 7. Again, as we would expect, decreasing L and R improves performance and increasing them penalizes performance. However, modifying C has a greater impact on the programs with less parallelism ($S = 0.01$ and $S = 0.001$). Modifying L and R has a greater impact on the programs with more parallelism ($S = 0.0001$ and $S = 0$). The reason for this is that C has more impact when many checkpoints are taken between recovery and failure. L and R have more impact when zero or few checkpoints are taken between recovery and failure—their effect is amortized as more intervals pass. When S is larger, extremely large numbers of active nodes are of little use (e.g., see the bottom line in the right side of Fig. 3). Therefore, as N grows beyond this number, the effect of the additional nodes is negligible because a remains fixed. If a is small enough (e.g., < 10,000), then the mean time in the System Up phase will have multiple checkpointing intervals. Thus, lowering C has greater impact.

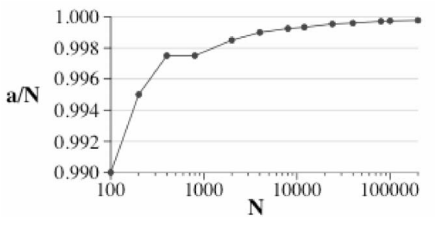

Fig. 6. The ratio of active nodes to N when the overheads of checkpointing and recovery are negligible, $S = 0$.

However, when S is small or zero, increasing a to large numbers still improves performance. Thus, as N grows, a grows to the point where it is unlikely that even one checkpoint is taken before a failure occurs. In these cases, L and R have a greater impact than C.

Modifying $MTTF$. In Fig. 9, we present the impact of the $MTTF$ on the performance of the system. Like L and R, the s has a significant impact. Thus, increasing the $MTTF$ by a factor of two has a significant beneficial impact on the performance of the system and decreasing it by a factor of two induces a similar penalty.

Modifying $MTTR$. In Fig. 10, we modify the mean time to repair significantly, from one hour, to one month, to six months. Increasing the $MTTR$ beyond an hour models failures that are of a nature that requires maintenance beyond simple detection and system reboot. For example, this models hardware failures. Although a six month repair time does not seem to be realistic, we include it to help understand the impact of modifying the $MTTR$. The major effect of increasing the $MTTR$ is to penalize the performance of the smaller values of N and the more parallel programs. Again, this makes sense because the true effect of increasing the $MTTR$ is to reduce the average number of nonfailed nodes at any one time. If a is large enough with respect to N, increasing the $MTTR$ results in the system spending more time in the Down phase, where the number of nonfailed nodes is less than a. When the system is not in the Down phase, the $MTTR$ has no impact. When N is small and S is small, the optimal values of a are large, relative to N, and modifying the $MTTR$ has significant impact. As N grows and S grows, the optimal values of a with respect to N are smaller and the $MTTR$ has little or no impact. Indeed, for N greater than 100,000, even the huge $MTTR$ of six months has no impact on the performance of the system. Similarly, for $S = 0.01$, the optimal values of a do not grow larger than 2,000 (again, see Fig. 10), the performance of all $MTTR$s is the same once N grows past 2,000.

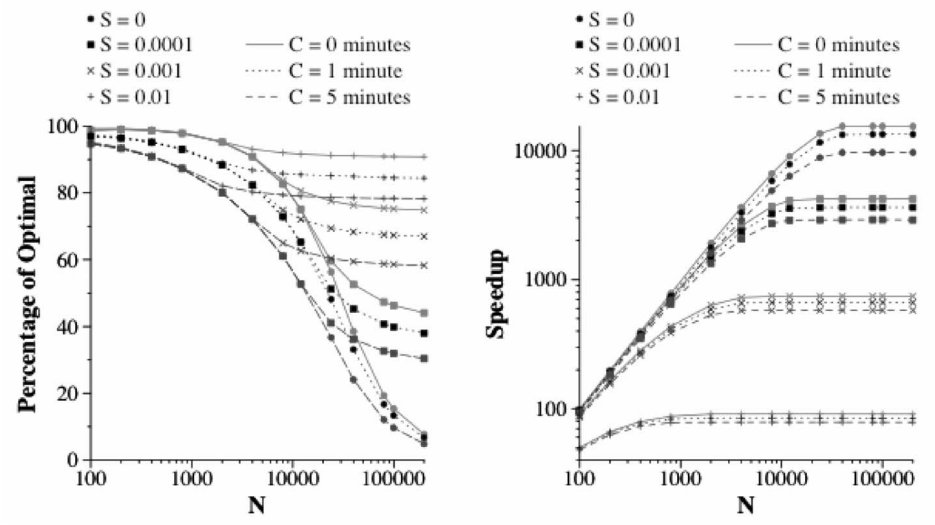

Fig. 7. The effect of modifying C.

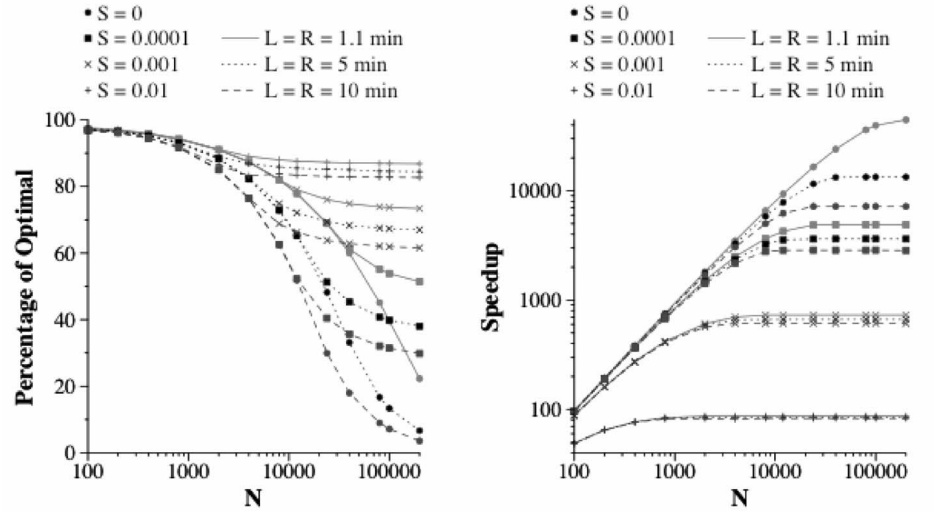

Fig. 8. The effect of modifying L and R.

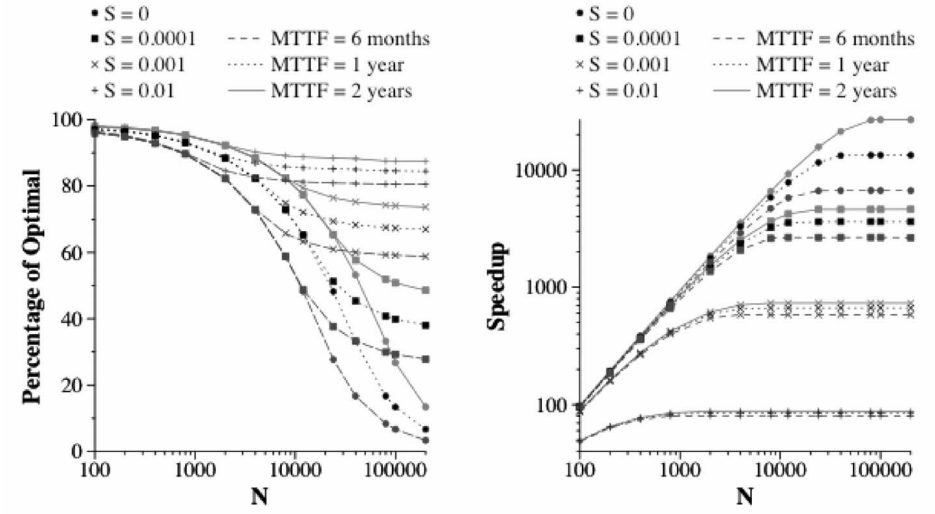

Fig. 9. The effect of modifying the $MTTF$.

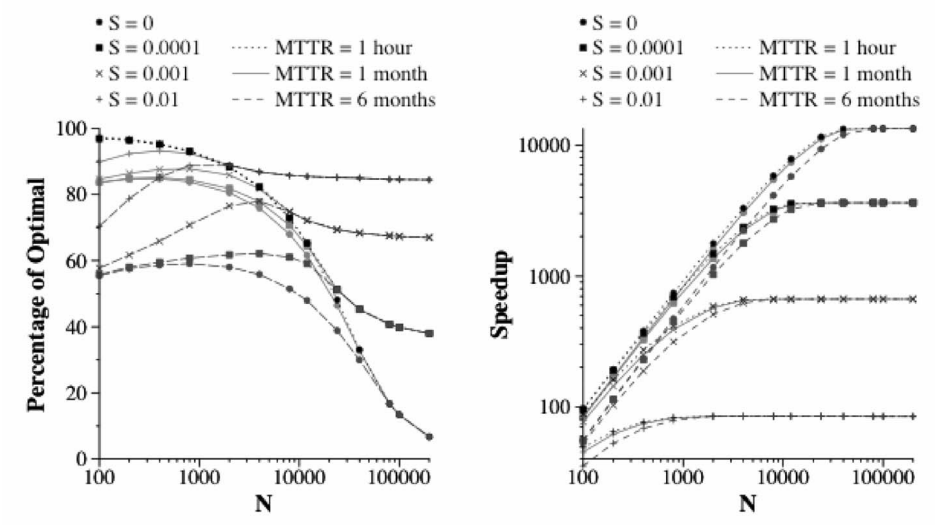

Fig. 10. The effect of modifying the $MTTR$.

Combining this study with the previous ones indicates that clusters of relatively large, powerful nodes will fare better if most failures are transient. Nontransient failures will take away a substantial portion of the processing power of the system, impacting scalability and performance. Systems composed of small, least powerful nodes will be resilient in the face of nontransient failures, but the frequency of failures will increase. Combining the two, it seems that medium sized nodes will offer a good trade off in the face of both transient and nontransient failures.

3.2.3 Penalty of Deviating from the Optimal Checkpointing Interval

When N grows into the 10,000s, the optimal checkpointing interval starts to approach the checkpoint latency. While this may optimize the Useful Work Factor, it means that, while the system is up, it will be checkpointing nearly constantly. Since checkpointing makes extensive use of the system's resources (disk, networking), if the system spends most of its time checkpointing, these resources will be less available to the application, which may impact performance. To assess the penalty of choosing a checkpointing interval that is not optimal, we display the performance of the system when I is limited to values greater than 20 minutes. Fig. 11 shows the variations of speedup and the percentage of optimal performance for various values of N.

The figures show that restricting the checkpointing interval to no less than 20 minutes has a performance impact. For small values of $N \leq 4,000$, the optimal checkpointing interval is greater than 20 minutes and, therefore, the policy has no effect on speedup or efficiency. As N increases, the choice of a checkpointing interval less than the optimal one leads to a slight performance degradation, especially for applications that scale well. It is therefore necessary from a practical standpoint to reduce the checkpointing interval depending on the number of nodes in the system.

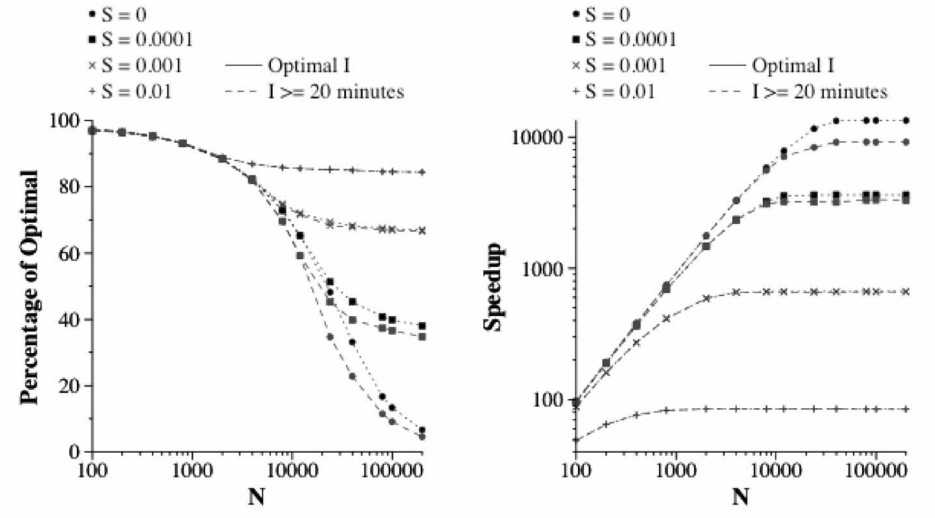

Fig. 11. The effect of restricting I to greater than 20 minutes.

4 Summary and Future Work

We have presented a study to predict the scalability of checkpointing as system size grows to meet the next performance goals of the high-performance computing community. We have argued that the pending technology slow down in device speed, together with cooling problems, power limitations, and the necessary overhead to overcome increased soft error rates, will reduce the relative improvement in the performance of future processors. As a result, the number of nodes in a typical super computer circa 2010 will increase by up to two orders of magnitude from today's 1,024-nodes range, depending on the node size. This will challenge the implementation of rollback-recovery algorithms.

We used the projections that we have in technology to study several potential configurations of a notional system that can deliver 1PF/sec of performance in the future. At one extreme, we used a 1,024-way node that is an example of NUMA machines, with 100 of these clustered together to deliver the needed performance. At the other extreme, we considered a cluster of "cheap commodity" systems using one-chip systems, requiring up to 100,000 nodes clustered together. Our simulation study shows that the characteristics and needs of rollback-recovery will play an important role in how systems should be configured in the future (more so than today's):

- As the number of nodes increases, the rate of failures increases. At the extreme, a 100,000-node system will devote a substantial portion of hardware resources to tolerating failures. The data-parallel model of most compute-intensive applications will force the system to be continuously checkpointing at the extreme.
- A system of relatively large nodes will not be resilient to permanent failures (however infrequent they may be). Therefore, and combined with the previous point, it appears that mid-sized nodes (4-way to 32-way SMP systems) will form the best trade off in terms of tolerating transient failures and yet be resilient to the loss of a few components to permanent failures.
- Regardless of the size of node used in the cluster, a fraction of a future system will need to be devoted to providing spare cycles to accommodate the needs of rollback-recovery in the system. This would be a departure from the current practice of configuring systems without allocating resources to accommodate the needs of rollback-recovery.
- For very large clusters (those of cheap commodity systems), there will be a need to set a maximum limit on the frequency of checkpointing (minimum checkpointing interval); otherwise, the checkpointing requirements will likely overwhelm storage and networking resources. We have shown that departure from the optimal checkpointing interval will have a tolerable impact on application performance, especially for those that have a high speed up.

For future work, we would like to study the impact of using checkpointing algorithms that limit the rollback of the system. This could be done simply by adapting Chandy and Lamport's algorithm to limit the rollback to those nodes that have acquired dependencies on the failed ones. Another idea is to divide the system into recovery domains [23] such that failures in one domain are confined to the domain and do not force further failure effects across domains. This would require that applications be deterministic and that interdomain messages be synchronously logged to stable storage.

While our study has focused on the performance and scalability aspects of checkpointing, we would like to emphasize that a lot of engineering work is needed to advance the usability and robustness of checkpointing implementation from the current state of the art. In particular, advances in automatic, programmer-transparent checkpointing would lead to better productivity, less susceptibility to bugs, and overall better performance.

Acknowledgments

E.N. Elnozahy was supported in part by the US Defense Advanced Research Project Agency under contracts NBCH30390004 and W0133090. J.S. Plank was supported in part by the US National Science Foundation under grants EIA0224441, ACI-0204007, ANI-0222945, and EIA-9972889.

References

[1] L. Alvisi, E.N. Elnozahy, S. Rao, S.A. Husain, and A. Del Mel, "An Analysis of Communication-Induced Checkpointing," *Proc. 29th Int'l Symp. Fault-Tolerant Computing,* June 1999.

[2] O. Babaoglu and W. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Reference Bits," *Proc. Symp. Operating Systems Principles,* pp. 78-86, 1981.

[3] A. Beguelin, E. Seligman, and P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," *J. Parallel and Distributed Computing,* vol. 43, no. 2, pp. 147-155, June 1997.

[4] B. Bhargava and S.R. Lian, "Independent Checkpointing and Concurrent Rollback for Recovery—An Optimistic Approach," *Proc. Symp. Reliable Distributed Systems,* pp. 3-12, 1988.

[5] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A Distributed Domino-Effect Free Recovery Algorithm," *Proc. IEEE Int'l Symp. Reliability, Distributed Software, and Databases,* pp. 207-215, Dec. 1984.

[6] M. Chandy and C.V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. Computers,* vol. 21, no. 6, pp. 546-556, June 1972.

[7] M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computing Systems,* vol. 3, no. 1, pp. 63-75, Aug. 1985.

[8] E.N. Elnozahy, L. Alvisi, Y.-M. Wang, and D.B. Johnson, "A Survey of Rollback-Recovery Protocols in Message Passing Systems," *ACM Computing Surveys,* vol. 34, no. 3, Sept. 2002.

[9] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems,* pp. 39-47, Oct. 1992.

[10] A.H. Johnston, "Scaling and Technology Issues for Soft Error Rates," *Proc. Fourth Ann. Research Conf. Reliability*, Oct. 2000.

[11] J.M. Hélary, A. Mostefaoui, R.H. Netzer, and M. Raynal, "Preventing Useless Checkpoints in Distributed Computations," *Proc. IEEE Symp. Reliable Distributed Systems,* pp. 183-190, Oct. 1997.

[12] Y. Huang and Y.-M. Wang, "Why Optimistic Message Logging Has Not Been Used in Telecommunication Systems," *Proc. 25th Int'l Symp. Fault-Tolerant Computing (FTCS-25),* pp. 459-463, June 1995.

[13] M. Kanellos, "Intel Hastily Redraws Road Maps," *CNET News,* May 2004.

[14] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. Software Eng.*, vol. 13, no. 1, pp. 23-31, Jan. 1987.
[15] C.C. Li and W.K. Fuchs, "CATCH: Compiler-Assisted Techniques for Checkpointing," *Proc. 20th Int'l Symp. Fault-Tolerant Computing (FTCS-20)*, pp. 74-81, June 1990.
[16] G. Moore, "Cramming More Components unto Integrated Circuits," *Electronics*, Apr. 1965.
[17] J.S. Plank, "Efficient Checkpointing on MIMD Architectures," PhD thesis, Princeton Univ., 1993.
[18] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Lipckpt: Transparent Checkpointing under UNIX," *Proc. USENIX Winter 1995 Technical Conf.*, pp. 213-223, Jan. 1995.
[19] J.S. Plank and K. Li, "Faster Checkpointing with N + 1 Parity," *Proc. 24th Int'l Symp. Fault-Tolerant Computing (FTCS-24)*, pp. 288-297, June 1994.
[20] J.S. Plank, J. Xu, and R.B. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," Technical Report CS-95-302, Univ. of Tennessee at Knoxville, Aug. 1995.
[21] B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 220-232, June 1975.
[22] R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Computer Systems*, vol. 1, no. 3, pp. 222-238, Aug. 1983.
[23] A. Sistla and J. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. Eighth Ann. ACM Symp. Principles of Distributed Computing (PODC)*, pp. 223-238, Aug. 1989.
[24] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 204-226, Aug. 1985.
[25] N. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 942-947, Aug. 1997.
[26] N. Vaidya, "On Staggered Checkpointing," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing*, Oct. 1996.

Elmootazbellah N. Elnozahy received the BSc degree with honors in electrical engineering from Cairo University in 1984 and the MS and PhD degrees in computer science from Rice University in 1990 and 1993, respectively. He is a senior manager and a master inventor at IBM Research in Austin, Texas. From 1993 until 1997, he was on the faculty at the School of Computer Science at Carnegie Mellon University, where he received a prestigious US National Science Foundation CAREER award. Since 1997, he has been with the IBM Austin Research Lab, where he started the Systems Software Department, which he currently leads. His research interests include distributed systems, operating systems, computer architecture, and fault tolerance. He has published 31 refereed articles and served on 21 technical program committees in these areas and was awarded 15 patents. He is a member of the IEEE.

James S. Plank received the BS degree from Yale University in 1988 and the PhD degree from Princeton University in 1993. He is currently an associate professor in the Computer Science Department at the University of Tennessee. His research interests are in fault tolerance, network storage, logistical networking, erasure coding, and Grid computing. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*, and a member of the IEEE Computer Society.