Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability

Adam J. Oliner*
Stanford University
Dept. of Computer Science
Palo Alto, CA 94305 USA
oliner@cs.stanford.edu

Larry Rudolph
MIT
CSAIL
Cambridge, MA 02139 USA
rudolph@csail.mit.edu

Ramendra K. Sahoo
IBM
T.J. Watson Research Center
Hawthorne, NY 10532 USA
rsahoo@us.ibm.com

ABSTRACT

Cooperative checkpointing increases the performance and robustness of a system by allowing checkpoints requested by applications to be dynamically skipped at runtime. A robust system must be more than merely resilient to failures; it must be adaptable and flexible in the face of new and evolving challenges. A simulation-based experimental analysis using both probabilistic and harvested failure distributions reveals that cooperative checkpointing enables an application to make progress under a wide variety of failure distributions that periodic checkpointing lacks the flexibility to handle. Cooperative checkpointing can be easily implemented on top of existing application-initiated checkpointing mechanisms and may be used to enhance other reliability techniques like QoS guarantees and fault-aware job scheduling. The simulations also support a number of theoretical predictions related to cooperative checkpointing, including the non-competitiveness of periodic checkpointing.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart, fault-tolerance*; C.4 [**Performance of Systems**]: Fault Tolerance

General Terms

Algorithms, Experimentation, Measurement, Reliability

Keywords

Cooperative checkpointing, RAS, high-performance computing, supercomputing, parallel computing, simulations

*Work was performed as part of the MIT VI-A Internship Program, a cooperative education program with Industry, on a fellowship funded by IBM. Some experiments were done at the T.J. Watson Research Center, as part of the Blue Gene/L System Software team.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS06, June 28-30, Cairns, Queensland, Australia.
Copyright 2006 ACM 1-59593-282-8/06/0006 ...$5.00.

1. INTRODUCTION

Periodic checkpointing, the standard method for providing reliable completion of long-running jobs, cannot cope with many realistic reliability challenges on large-scale systems; *cooperative checkpointing*, in which checkpoint requests may be skipped, provides greater performance and reliability by enabling flexible behavior. With cooperative checkpointing, the application programmer, the compiler, and the runtime system are all part of the decisions regarding when and how checkpoints are performed. Specifically, the programmer inserts checkpoints at locations in the code where the application state is minimal, placing them liberally wherever a checkpoint would be efficient. The compiler then removes any state which it finds to be superfluous, checks for errors, and makes various optimizations that reduce the overhead of the checkpoint. At runtime, the application *requests* a checkpoint. The system *grants* or *denies* the checkpoint based on various system-wide heuristics, including disk or network usage and reliability information. A key benefit of cooperative checkpointing is that checkpoints less likely to be used for recovery can be *skipped*, thereby improving overall performance.

Standard practice is to checkpoint periodically, at an interval determined primarily by the overhead and the failure rate of the system. Although such a scheme is optimal under an exponential (memoryless) failure distribution, real systems may not exhibit such failure behavior [11, 18, 20] and there is no reason to believe different systems will share reliability characteristics. Furthermore, periodic checkpointing does not scale with the growing size and complexity of systems [17]. Cooperative checkpointing performs well under all tested failure distributions and system parameters, including those where periodic checkpointing fails.

The system has an opportunity to skip requested checkpoints at runtime, and thus may be considered a hybrid of application-initiated and system-initiated checkpointing. The application requests checkpoints, and the system either grants or denies each one. Without cooperative checkpointing, all application-initiated checkpoints are taken, even if system-level considerations would have revealed that some are inefficient or have a low probability of being used for recovery. Cooperative checkpointing also leads to more portable code; an application instrumented with checkpoint requests can be run under many different failure distributions.

We verify our claims of robustness by means of extensive simulations of large-scale systems. For example, using cooperative checkpointing in one instance reduced bounded

slowdown by a factor of nine, improved system utilization, and lost no more work to failures than periodic checkpointing; this occurred even when event prediction had a 90% false negative rate. We also confirm the theoretical result that periodic checkpointing can be arbitrarily bad for non-exponential failure distributions.

2. BACKGROUND

Checkpointing for computer systems has been a major area of research over the past few decades. Work has typically focused on determining the optimal periodic checkpoint interval [19, 23], reducing the overhead [12] or size of a checkpoint (e.g. incremental checkpointing [2, 5]), and improving the models used to describe system reliability [18, 22]. One area that merits greater attention is the flexibility and robustness of current checkpointing approaches.

In order for any reliability scheme to be effective, one must develop useful models of the failure behavior of supercomputers. Several early studies in the 1980's and 1990's looked at the failure trends and developed theoretical models for small to medium-scale computer systems [4, 9]. Recent studies harvested failures from both large-scale clusters of commodity machines, as well as from a Blue Gene/L prototype [11]. Critical event prediction algorithms on real system traces have seen accuracies up to 80% [10, 20].

Tantawi and Ruschitzka [22] developed a theoretical framework for performance analysis of checkpointing schemes. In addition to considering arbitrary failure distributions, they recognize the importance of flexibility by presenting the *equicost* checkpointing strategy, which varies the checkpoint interval according to a balance between the checkpointing cost and the likelihood of failure. The costs of checkpointing at arbitrary points in a program's execution is one of the many undesirable characteristics that make system-initiated checkpointing inappropriate for large-scale systems.

Cooperative checkpointing is addressed in several unpublished documents [13, 16, 17], but the only published manuscript [14] deals exclusively with the theoretical models and mathematical analysis. All of our listed contributions are exclusive to this publication. This is the first paper to propose cooperative checkpointing as a practical solution (with a discussion of gatekeeper heuristics and checkpointing scheme properties), the first to motivate the need for cooperative checkpointing as an alternative to periodic checkpointing, and the first to empirically evaluate cooperative checkpointing.

3. TERMS AND DEFINITIONS

Define a *failure* to be any event in *hardware or software* that results in the immediate failure of a running application. At the time of failure, any unsaved computation is lost and execution must be restarted from the most recently completed checkpoint.

When an application initiates a checkpoint at time t, progress on that job is paused for the *checkpoint overhead* (C) after which the application may continue. The *checkpoint latency* (L) was shown [18] to have an insignificant impact on checkpointing performance for realistic failure distributions. Therefore, we treat $C \approx L$. We assume downtime and recovery time are independent of the checkpointing choices; regardless, we make no further mention of them, nor L, as they do not impact cooperative checkpointing analysis.

In the context of cooperative checkpointing, I refers to the length of the periodic *request* interval. Thus, a checkpoint will be requested by the application every I seconds, but not necessarily taken each time. This is an intentional simplification of cooperative checkpointing, which does not require that requests be made periodically.

From a system management perspective, supercomputer node time is a valuable resource. Define a unit of *work* to be a single node occupied for one second. That is, occupying n nodes for k seconds consumes work $n \cdot k$ node-seconds. Thus, a node sitting idle, recomputing work lost due to a failure, or performing a checkpoint is considered *wasted work*. We find it better to use the complementary metric, *saved work* (or committed work), which is the total execution time minus its wasted time. Saved work never needs to be recomputed. Checkpointing overhead is considered wasted work and is never included in the calculation of saved work. For example, if job j runs on n_j nodes, and has a failure-free execution time (excluding checkpoints) of e_j, then j performs $n_j \cdot e_j$ node-seconds of saved work. If that same job requires E_j node-seconds, including checkpoints, then a failure-free execution effectively *wastes* $E_j - e_j$ node-seconds.

Figure 1 shows typical application behavior and illustrates the concepts of saved and wasted work. Periods of computation are occasionally interrupted to perform checkpoints, during which job progress is halted. Job failure forces a rollback to the previous checkpoint; any work performed between the end of that checkpoint and the failure must be recomputed and is considered wasted. Long-running applications will have hundreds of these checkpoints, many of which may never be used.

A failure-free interval (FFI) is a period of time during which the application can perform useful work, and so it excludes downtime and recovery time. An FFI is ended by a failure, and an FFI length is the span of useful computation time between two consecutive failures.

4. MOTIVATION

High performance computing systems continue to grow in size and complexity. For example, a 64-rack Blue Gene/L (BG/L) system contains 65,536 nodes and more than 16 terabytes of memory [1]. Applications on these systems are designed to run for days to months. Despite a design focus on reliability, there is little evidence that reliability will improve faster than the increases in machine size. If we hope to use checkpointing to provide reliable completion of these jobs on inherently unreliable hardware, it is clear that standard checkpointing techniques must be reevaluated [6].

Periodic checkpointing is not the best solution to providing reliable execution of jobs under realistic system conditions. Consider Figure 2, which shows the performance of a large-scale cluster at different checkpointing overheads and checkpoint intervals. Average bounded slowdown is a standard performance metric that considers the ratio of time a job actually spends in a system (from submission to completion) to its execution time without checkpoints. Work lost measures the amount of work that had to be recomputed due to failures. We want to minimize both metrics. This particular set of results used a real job log (LLNL T3D [7]) from a toroidal architecture machine similar in configuration to BG/L and a failure distribution harvested from a 350-node AIX cluster [20]. We performed similar experiments with other interconnect architectures, job logs, and scheduling algorithms (more details on the other simulations

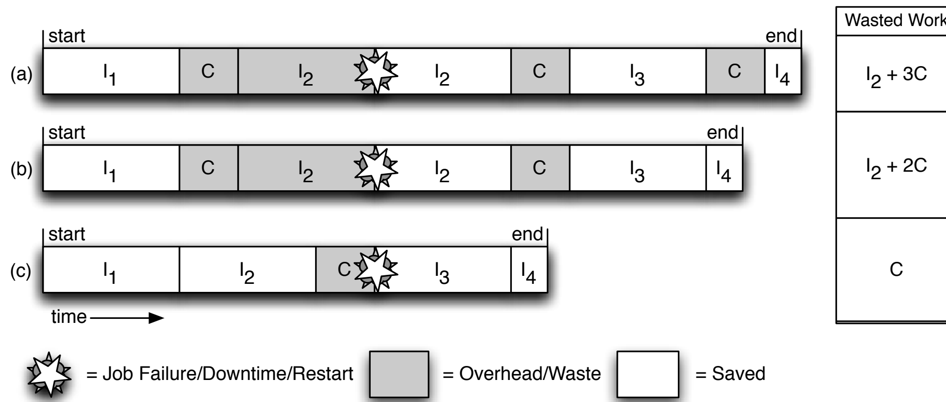

Figure 1: Three runs of a job in which different checkpoints are skipped. Run (a) shows typical periodic behavior, in which every checkpoint is performed. In run (b), the final checkpoint is skipped, perhaps because the critical event predictor sees a low probability that such a checkpoint will be used for rollback, given the short time remaining in the computation. Finally, run (c) illustrates optimal behavior, in which a checkpoint is completed immediately preceding a failure. The figure lists the amount of work wasted in each execution.

can be found elsewhere [17]).

In all experiments, the conclusion was that periodic checkpointing, while the most common method of mitigating job-level loss from failures, may not be appropriate for improving system-level metrics. Indeed, the act of checkpointing can be more detrimental than the failures, themselves. Not only that, but the metrics give conflicting advice: Figure 2(a) argues for checkpointing as infrequently as possible, while Figure 2(b) implies that frequent checkpoints are effective (to a point). There is a need for a practical checkpointing technique that both reduces lost work and improves system-level metrics. Cooperative checkpointing does exactly that: it acts to improve job-level and system-level metrics with a mechanism that is both practical and flexible.

5. COOPERATIVE CHECKPOINTING

Cooperative checkpointing is a set of semantics and policies that allow the application, compiler, and system to jointly decide when checkpoints should be performed. Specifically, the application requests checkpoints, which have been optimized for performance by the compiler, and the system grants or denies these requests. The general process consists of two parts:

1. The application programmer inserts *checkpoint requests* in the code at places where the state is minimal, or where a checkpoint is otherwise efficient. These checkpoints can be placed liberally throughout the code, and permit the user to place an upper bound on the number and rate of checkpoints. The compiler optimizes these requests by catching errors, removing dead variables, and assisting with optimization techniques such as incremental checkpointing.

2. The system receives and considers checkpoint requests. Based on system conditions such as I/O traffic, critical event predictions, and user requirements, this request is either *granted* or *denied*. The mechanism that handles these requests is referred to as the checkpoint gatekeeper or, simply, *the gatekeeper*. The request/response latency for this exchange is assumed to be negligible.

There are many possible implementations of this technique; it can be made to guarantee distributed process consistency, support incremental checkpointing, and so on, based on the requirements of the particular system. For example, state-of-the-art application-initiated checkpointing schemes [21] already have a predicate-based checkpointing decision point, currently triggered on elapsed time. This can easily be changed to query a system-level cooperative checkpointing gatekeeper that incorporates other factors as well. Failure prediction capabilities for BG/L have been completed [10] and an implementation of cooperative checkpointing for BG/L is underway.

Cooperative checkpointing appears to an observer as irregularity in the checkpointing interval. When good heuristics are used, this irregularity produces greater resilience against a variety of failure distributions. The behavior of applications as they choose to skip different checkpoints is illustrated in Figure 1.

The primary policy question with regard to cooperative checkpointing is, "How does the gatekeeper decide which checkpoints to skip?" There are many heuristics that the gatekeeper may use, including:

- *Network Traffic.* Network I/O is a bottleneck with respect to saving state to disk. The gatekeeper may choose to skip a checkpoint if traffic conditions suggest that the checkpoint would take unacceptably long.

- *Disk Usage.* Similarly, the shared stable storage itself may be the bottleneck if the network bandwidth leading to the disks outpaces the media's available write bandwidth.

- *Job Scheduling Queue.* If a high-priority job is waiting for a running job's partition, it may be profitable to risk skipping checkpoints to allow that waiting job to run sooner. For example, if a single-node job is blocking a 128-node job, then we would rather skip some

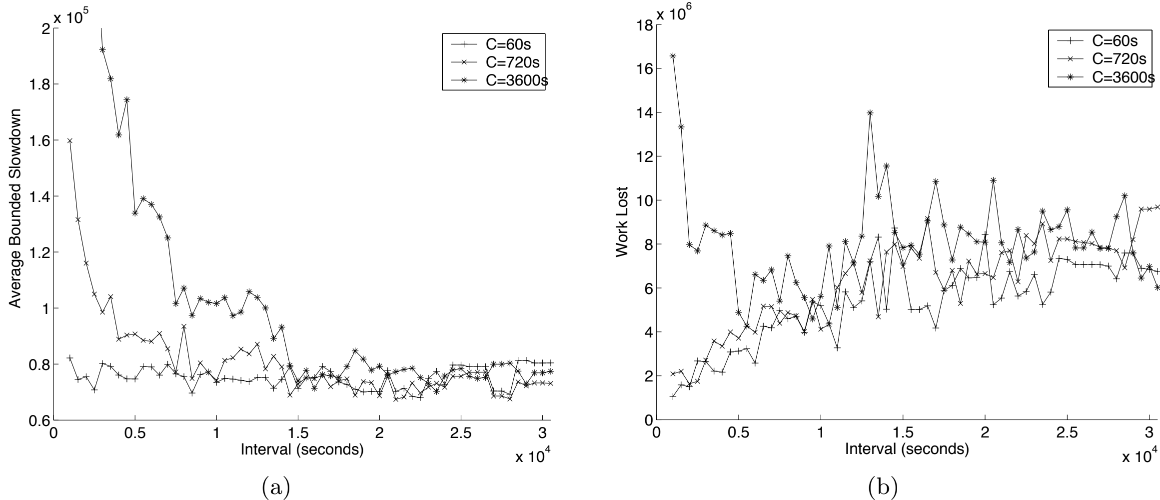

Figure 2: Simulations of periodic checkpointing under realistic large-scale system conditions (a) suggest that the optimal system-level strategy is to never checkpoint at all. At high enough overheads (b), periodic checkpointing is not even effective at reducing the amount of work lost to failures. C is the checkpoint overhead in seconds.

of the small job's checkpoints to free up that node as soon as possible.

- *Event Prediction.* If a failure is likely to occur in the near future, the gatekeeper should choose to save the state before that happens. On the other hand, if system conditions are stable, performing the checkpoint may be a waste of time and resources. Failure prediction on real systems can be very accurate [10, 20].

- *QoS Guarantees.* Many systems make QoS guarantees to users in the form of deadlines or minimum throughput. Cooperative checkpointing can be used as a tool to help keep those promises [15]. For example, a job that started later than expected can be made to skip checkpoints in order to reduce its effective running time, thereby potentially meeting a deadline it would otherwise have missed.

Note that most of these heuristics *cannot* and *should not* be considered by the application programmer at compile-time. At the same time, there are many aspects of the internal logic of an application (data semantics, control flow) that *cannot* and *should not* be considered by the system at runtime. Neither application-initiated nor system-initiated checkpointing satisfactorily considers all these factors in deciding when to perform checkpoints.

Table 1 compares characteristics of cooperative checkpointing to system-initiated and application-initiated checkpointing. Certainly, the table is neither complete nor strictly precise. The entries in the table, however, act as a useful reminder of the tradeoffs made by designers attempting to construct reliable systems. The features we consider are:

- *Semantics.* The checkpointing scheme is aware of the semantics of the data, and can save only what is needed to recreate the application state.

- *Min-State.* The checkpoints are performed at places in the code where application state is minimal, such as at iterations of an outer loop.

- *Portable.* Checkpoints may be used for restart on machines that are different from the ones on which the checkpoint was made. This is useful for heterogeneous systems.

- *Compiler.* At compile time, the application can be optimized to more efficiently perform the checkpoints.

- *Runtime.* The checkpointing policy is decided at runtime, and can consider such factors as the size of the application's partition, system health, and network traffic.

- *Kernel State.* The checkpointing mechanism is able to save and restore kernel-level information, such as PID or PPID.

- *Transparent.* User intervention is not required to accomplish checkpointing; checkpoints are placed and performed transparently.

Transparency is difficult to achieve without sacrificing knowledge of the data semantics and application state behavior. That knowledge translates to smaller checkpoints, and, consequently, smaller checkpointing overheads. Using compiler or preprocessing techniques [3, 21], however, it is possible to minimize user intervention.

5.1 Checkpointing Algorithms

The space of deterministic cooperative checkpointing algorithms is countable. Each such algorithm is uniquely identified by the sequence of checkpoints it skips and performs. One possible way to encode these algorithms is as binary sequences, where the kth digit is 1 if the kth checkpoint should be performed and 0 if it should be skipped. For example, an algorithm A that skips every third checkpoint could be written as: $A = \{1, 1, 0, 1, 1, 0, 1, \ldots\}$

We use *deterministic* to refer to a cooperative checkpointing algorithm that decides which checkpoints to perform at the beginning of the failure-free interval, and *dynamic* to

Feature	System	Application	Cooperative
Semantics		×	×
Min-State		×	×
Portable		×	×
Compiler		×	×
Runtime	×		×
Kernel State	×		×
Transparent	×		

Table 1: Comparison of the characteristics of the two major approaches in addition to cooperative checkpointing. Cooperative checkpointing provides nearly all of the benefits of the other schemes, with the exception of transparency. In the absence of better compilers or developer tools, however, transparency necessarily comes at the cost of smaller, more efficient checkpoints; that is not an acceptable tradeoff for most high performance applications.

refer to those that make decisions online based on whatever information is available at that point. The following algorithms are considered in this paper.

5.1.1 Offline Optimal

Denoted as OPT, this algorithm nondeterministically performs the latest checkpoint in a failure-free interval that it can complete before the failure, and no others. It is used as a point of reference for the performance of more practical algorithms.

5.1.2 Periodic

The naïve implementation of **periodic** checkpointing performs the dth checkpoint, the $2d^{th}$ checkpoint, the $3d^{th}$ checkpoint, and so on. When dI is optimal according to Young's approximation [23] and is used under an exponential failure distribution, periodic checkpointing is roughly optimal for a deterministic algorithm. A similar algorithm, **revised periodic** checkpointing, always performs the first checkpoint, then the $(d+1)^{th}$ checkpoint, the $(2d+1)^{th}$ checkpoint, and so on. Examples of these algorithms in binary notation follow:

$$\begin{aligned}[\text{ Periodic, d=2 }] &= \{0,1,0,1,0,1,\ldots\}\\ [\text{ Revised Periodic, d=3 }] &= \{1,0,0,1,0,0,1,\ldots\}\end{aligned}$$

5.1.3 Risk-Based

This dynamic cooperative checkpointing algorithm makes online decisions about whether or not to skip each checkpoint request using a probabilistic analysis. When deciding whether to perform checkpoint i, it asks how much work it expects to lose before checkpoint $i+1$ would be completed. If that measure is greater than the cost of checkpointing, then it performs the checkpoint. This strategy is also called **risk-based** checkpointing. Let p_f be the probability that a failure will happen before checkpoint $i+1$ completes (determined from the failure density function), and let d be the number of consecutive preceding intervals that have elapsed since a checkpoint was completed. The expected cost of skipping the checkpoint is $p_f((d+1)I+C)$, with no cost if a failure does not occur. The cost of performing the checkpoint is $p_f(I+2C)+(1-p_f)C$. This gives the heuristic for risk-based checkpointing, which is $p_f dI \geq C$. If the inequality holds, the algorithm performs the checkpoint.

5.1.4 Exponential Backoff

Let **backoff** be a cooperative checkpointing algorithm that doubles the amount of saved work at the completion of each checkpoint. Thus, in each failure-free interval, it performs the 1st, 2nd, 4th, 8th, etc. checkpoints:

$$[\text{ Backoff }] = \{1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,1,\ldots\}$$

All of the algorithms described in this section are technically cooperative checkpointing algorithms, because they each involve skipping checkpoint requests as part of their behavior. Risk-based, however, captures the essence of cooperative checkpointing by explicitly weighing the cost of checkpointing against the benefit from doing so using information beyond merely the mean time between failures.

6. ROBUSTNESS OF COOPERATIVE CHECKPOINTING

Robustness implies the ability to withstand changes in the environment, rather than being optimal for some particular set of conditions and metrics. We show that cooperative checkpointing is robust to system parameters and failure behavior where periodic checkpointing is not.

6.1 Robust to System Parameters

Before considering a variety of failure distributions, we revisit the experiments of Section 4 and see that cooperative checkpointing is robust under a variety of system parameters. These experiments simulated the scheduling of more than 30 million jobs over 600,000 machine hours using job logs from actual supercomputers and a failure trace from a 350-node AIX cluster.

Recall Figure 2, which showed periodic checkpointing experiencing terrible performance degradation at short checkpoint intervals; when the overhead became very large it even failed to reduce lost work. Now, consider a system using cooperative checkpointing as in Figure 3. "Risk A" means risk-based checkpointing whose predictor has an accuracy of A; there are no false positives, so this also implies a false negative rate of (1-A). "Work" means work-based checkpointing, which checkpoints whenever unsaved work exceeds checkpoint overhead.

Cooperative checkpointing improves both system-level metrics like bounded slowdown and job-level metrics like work lost due to failures. According to extensive experimentation, cooperative checkpointing retains good performance at low prediction accuracies (at most 10% is needed); toroidal or flat interconnect architectures; job logs from LLNL, SDSC, and NASA [7]; checkpoint overheads of 720 seconds and 3600 seconds; a huge range of checkpoint request intervals; and considering metrics like bounded slowdown, response time, wait time, average system utilization, and total work lost to failures. An application using periodic checkpointing would have to be reconfigured for each of these cases, likely with programmer intervention, and would still not obtain the performance and reliability of cooperative checkpointing. We believe the potency of low prediction accuracy is a function of the temporal and spatial clustering of failures in the real system trace, a trend which we have observed in many other systems. Predicting a member of such a cluster effectively predicts them all.

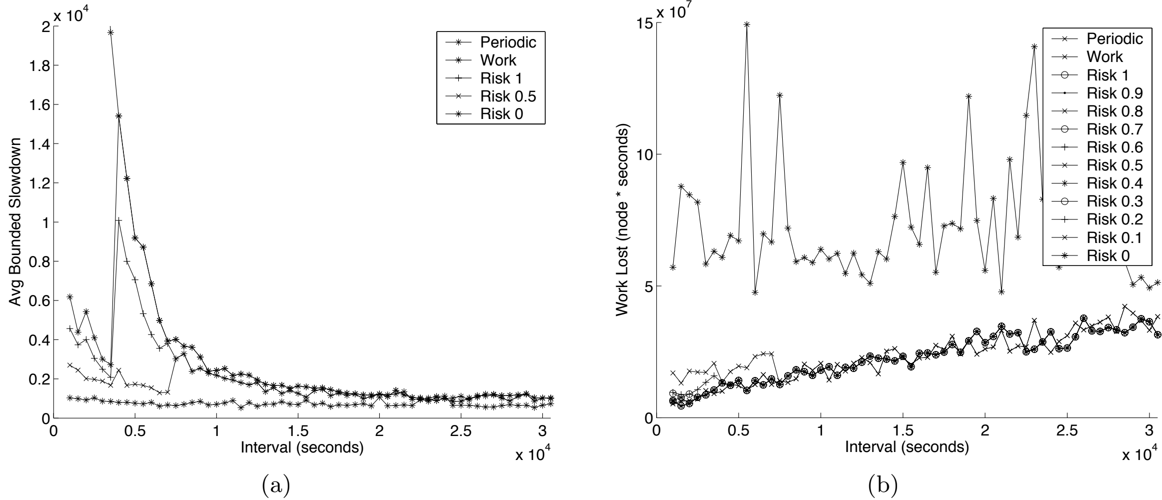

Figure 3: Cooperative checkpointing controls the exponential explosion of bounded slowdown as checkpoint intervals become smaller. Figure 3(b) shows that it simultaneously reduces the amount of work lost to failures, a feat that periodic checkpointing could not do for large overheads. This tremendous benefit was garnered even when the event predictions given to the cooperative checkpointing gatekeeper had a 90% false-negative rate.

C (sec)	I (sec)	T (sec)	# FFIs
720	$\frac{I_{OPT}}{2}$	60,480,000	100,000

Table 2: Simulation parameters. These values are assumed where not specified to be otherwise. The choice of I means that optimal periodic checkpointing will perform every second checkpoint, because the request interval is half the optimal periodic interval.

6.2 Robust to Failure Behavior

Cooperative checkpointing provides robustness against numerous failure behaviors that periodic checkpointing does not. These experiments were performed using a specially designed cooperative checkpointing simulator written in Java. The simulator uses a given failure density function, which is either probabilistic or based on an input trace, to generate FFI (failure-free interval) lengths. For each cooperative checkpointing algorithm, the simulator tests its behavior during that interval, recording both how much work was saved and what ratio it achieved relative to the offline optimal. We tested each of the algorithms described in Section 5.1. Most runs performed $100,000$ of these intervals for each algorithm for each failure distribution for each mean time between failures.

The checkpoint overhead C was set to 720 seconds (half the maximum estimated overhead for BG/L), and the maximum application running time was set at $T = 100$ weeks (larger than any observed FFI length). The checkpoint request interval was set to be half the optimal periodic checkpointing interval for a distribution with a given mean ($I = \frac{I_{OPT}}{2}$). This means that there are twice as many possible placements of checkpoints. It also means that periodic and revised periodic checkpointing will have $d = 2$ in these experiments (they will skip every other checkpoint). The parameters are summarized in Table 2.

Although a large amount of information was collected, this paper focuses on the presentation of two similar but not necessarily correlated metrics:

Average ratio: For a given interval i, let V_A^i be the amount of work saved by algorithm A and let V_{OPT}^i be the amount saved by the offline optimal. The average ratio computes the average over all intervals i of $\frac{V_A^i}{V_{OPT}^i}$.

Average saved work: The average saved work calculation sums the saved work over all intervals and divides by the number of intervals. It has units of node-seconds, but here we treat the cluster as a single node.

System reliability is commonly modeled by one of several probability densities that describe the distribution of time to failure: exponential, Weibull, and uniform. It is also possible to create an empirical probability distribution using machine traces, which can then be used as a model for future behavior. These distributions generate the time to failure for the system: if a distribution generates a value f_1, that means the system will fail after running for f_1 seconds. Remember that we are considering any failure that causes a job to fail, be it in hardware or software. More information on these distributions can be found elsewhere [8].

For each failure distribution, we varied the mean ($\frac{1}{\lambda}$) among 14 values, evenly distributed from an expectation of 10 failures per day (8640 seconds) to one failure every 5 days (432000 seconds). The Weibull distribution we used was a sum of three subpopulations of Weibull distributions to approximate the so-called "bathtub" reliability function. One subpopulation had $\beta = 0.5$ and represented early-life burn-in failures, one with $\beta = 1$ for steady state, and one with $\beta = 1.5$ for wear-out failures. The scale parameter, η, was determined from the desired mean:

$$\frac{1}{\eta} = \lambda\Gamma(1 + \frac{1}{\beta})$$

The uniform distribution was defined as a constant over $[0, \frac{2}{\lambda}]$ so that the mean is $\frac{1}{\lambda}$. This implies that there will always be a failure within that interval of time; the expected

rate of failure increases as the interval progresses. We defined a composite uniform distribution that consisted of two uniform distributions: one centered at $I + C$ and having weight 0.99, and one centered at $(\frac{1}{\lambda} - 0.99(I + C))/0.01$ with weight 0.01. In other words, nearly all the failures happen just after the first checkpoint (if taken) would be completed, while a small number occur far later.

The real machine trace was harvested from a 4,096-node prototype of IBM's Blue Gene/L supercomputer during the course of just over 18 weeks. More information on the machine and the original traces can be found elsewhere [11]. The logs were filtered to isolate individual critical failure events from those that were either not severe or non-critical informational events. The final filtered trace had 124 entries that correspond to processor or memory failures that would have caused an application using that node to crash. The FFI lengths in the trace are distributed almost exponentially at small inter-failure times, but there are several intervals that last for much longer.

6.2.1 Generated Trace Results

Figure 4 illustrates the results of runs under an exponential failure distribution with varying mean. The pair of plots represents simulated data for seven million intervals of failure-free execution. Figure 4(a) shows the average ratio with the optimal. At low means, both exponential backoff and revised periodic checkpointing do the best because they always perform the first checkpoint; at higher means those two do the worst, with backoff giving the most precipitous drop. Figure 4(b) shows average amount of saved work. In general, both graphs show performance increasing with reliability, with the exception of the ratio with backoff. This happens because backoff will achieve a poor ratio when the next scheduled checkpoint falls after the next failure, but on the whole saves a respectable amount of work. Both plots also exhibit an interesting inflection point. When the average interval is small, many intervals will not give any saved work and the average will be dominated by the large intervals. Past this inflection point, the amount of saved work grows roughly linearly with the mean time between failures.

The results for the Weibull bathtub distribution are given in Figure 5. Simple periodic checkpointing and risk-based checkpointing are clear winners here. The benefits backoff and revised periodic showed at high failure rates under the exponential distribution are less pronounced here, because nodes that fail early only make up a third of the population. Deterministically performing the first checkpoint may bound the damage in some bad cases, but it is not always a desirable behavior.

There are several interesting features of Figure 6, which plots results for the uniform distribution. First is the sawtooth pattern in the average ratio of the backoff algorithm. This is a consequence of checkpoints that fall right around the failure time. If the checkpoint came just before the failure, backoff does very well. On the other hand, if that checkpoint doesn't finish before the failure, backoff saves only *half* as much work as if the checkpoint had finished, by virtue of the nature of the algorithm, which doubles the amount of saved work with each checkpoint.

The second interesting feature of Figure 6(a) is the unusually poor average ratio of risk-based checkpointing. This is a consequence of the derivation of the algorithm's heuristic, which implicitly assumed a memoryless distribution. There is also some question about the appropriateness of the uniform distribution as a reliability model, especially as it is treated here: there is *always* a failure before time $t = b$, and the expected failure rate increases as the interval elapses.

Figure 7 clearly demonstrates both the robustness of cooperative checkpointing and the difference between the two metrics. Despite performing well under the other distributions, Figure 7(a) shows that periodic checkpointing does poorly in terms of its ratio with the optimal. The other algorithms give an excellent ratio. Risk-based checkpointing did well under all distributions and was therefore demonstrated to be more robust.

On the other hand, Figure 7(b) looks very much like the saved work plots for every other distribution, suggesting that the metric lacks some crucial information that is needed to accurately identify the merit of a checkpointing algorithm. Periodic checkpointing almost never manages to save any work under this composite uniform distribution, because it skips the first checkpoint; this gives it a poor average ratio. When it does save work, however, it saves a huge amount; this causes the average saved work to look normal.

6.2.2 Machine Trace Results

Figure 8 plots the results of experiments using the Blue Gene/L harvested failure trace. The mean is not varied. Instead, each algorithm is given along the horizontal axis. Risk-based checkpointing would have resulted in the shortest application running time, but did not actually give the best average ratio (revised periodic did). Revised periodic performs better than periodic checkpointing under both metrics, suggesting that the small modification to periodic checkpointing translates to better performance in practice. Recall that this approximated failure distribution was generated from only 124 failure events; the performance may improve as more data is gathered.

In the BG/L trace, we observed what appeared to be an exponential failure distribution at shorter intervals. Combining that observation with the good performance of backoff at higher failure rates in Figure 4(a) might lead one to conclude that backoff should do well here. It seems, however, that the mean of the trace distribution is not low enough to bring out the benefits of performing many of the early checkpoints. Note that the full Blue Gene/L is 64 racks, and may exhibit failure rates significantly different from this trace.

The results were not very sensitive to the chosen simulation parameters. Varying the checkpoint overhead (C) does not significantly affect the relative performance of the algorithms, but does lead to an across-the-board decline in both average ratio and saved work. Backoff is hit hardest, because it pays for a fixed number of checkpoints in a given set of requests. As long as I is some integer fraction of I_{OPT}, and we neglect the request overhead, the impact of varying I on risk-based checkpointing is nil. With backoff, however, decreasing I significantly increases the number of checkpoints the algorithm performs early on, which quickly becomes a detriment to performance.

Cooperative checkpointing is designed to favor algorithms that do well in failure-free intervals of all lengths, rather than great in some and terrible in others. This makes them robust enough to handle many different failure distributions, without necessarily sacrificing average saved work. In every experiment, risk-based checkpointing did well in both metrics; the same cannot be said for periodic checkpointing.

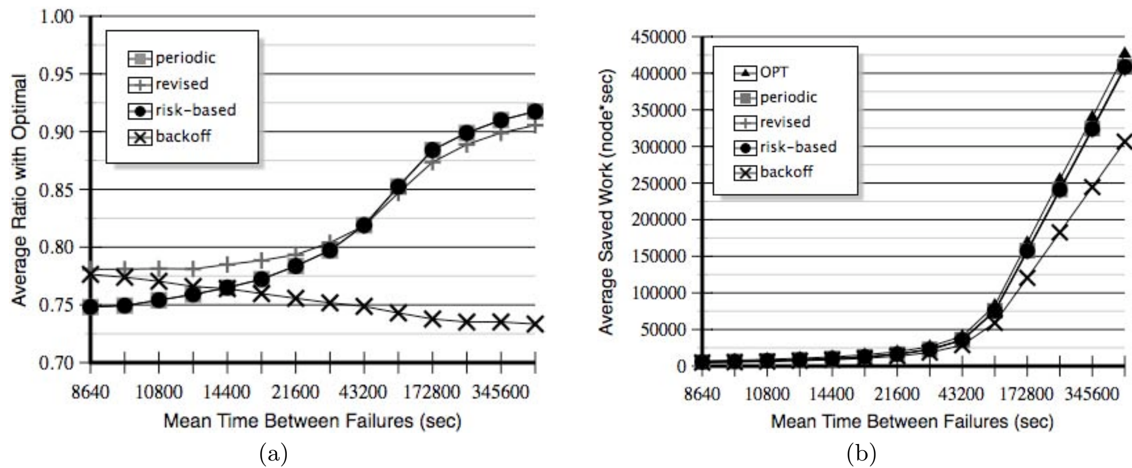

Figure 4: Exponential failure distribution. The exponential backoff algorithm is typically worse, but beats many of the others when the mean is small. Risk-based checkpointing performs comparably with periodic checkpointing, which is important to remember when we see risk-based do much better than periodic for other distributions.

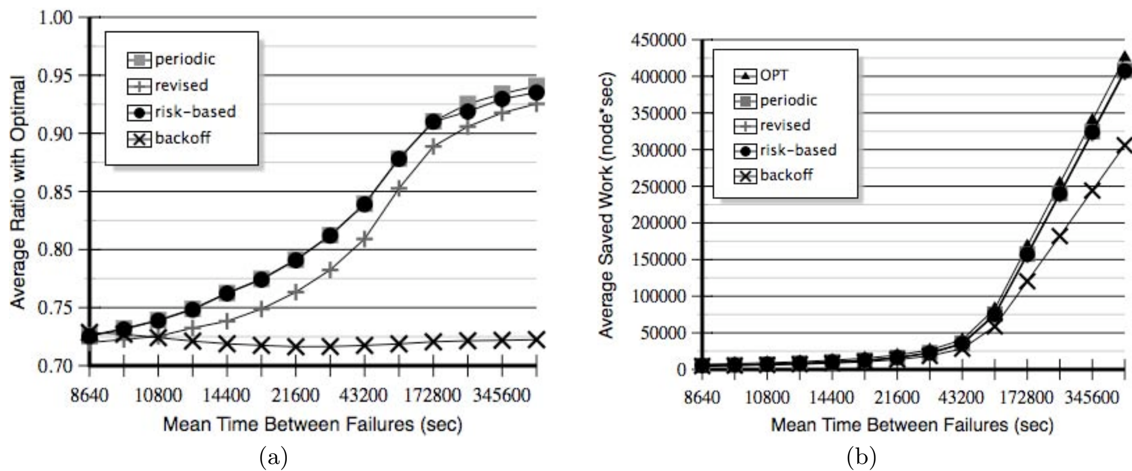

Figure 5: Weibull bathtub failure distribution. Revised periodic checkpointing does consistently worse than the naïve implementation, meaning that performing the first checkpoint is not always ideal behavior. Risk-based checkpointing once again keeps pace with periodic checkpointing.

7. CONTRIBUTIONS

Cooperative checkpointing, which empowers a system to skip checkpoints requested by applications at runtime, is more robust than periodic checkpointing to variations in failure distributions and system parameters. To summarize: the user inserts checkpoint requests liberally where they would be efficient, the compiler optimizes these possible checkpoint locations, and the system (gatekeeper) may consider any number of runtime factors when deciding which checkpoints to skip. Some of the contributions of this paper are as follows:

- Motivates the work by showing that periodic checkpointing does not scale under realistic failure distributions (Figure 2) and that even low accuracy prediction can be a powerful resource worth exploiting (Figure 3).
- Argues that cooperative checkpointing is a simple, efficient, portable, and flexible reliability tool that can be practically implemented on top of existing application-initiated checkpointing mechanisms, and its infrastructure can be used to facilitate numerous other system tasks such as QoS guarantees or job scheduling (Section 5).
- Describes a cooperative checkpointing simulator written in Java and the experiments performed with that software. Using large-scale system parameters and real traces (where appropriate), we simulated hundreds of millions of machine hours and verified our claims of robustness (Figures 3-8).
- Demonstrates the behavior of several cooperative checkpointing algorithms under a number of reliability conditions. These algorithms include the offline optimal, periodic checkpointing, revised periodic checkpointing, risk-based checkpointing, and exponential backoff.

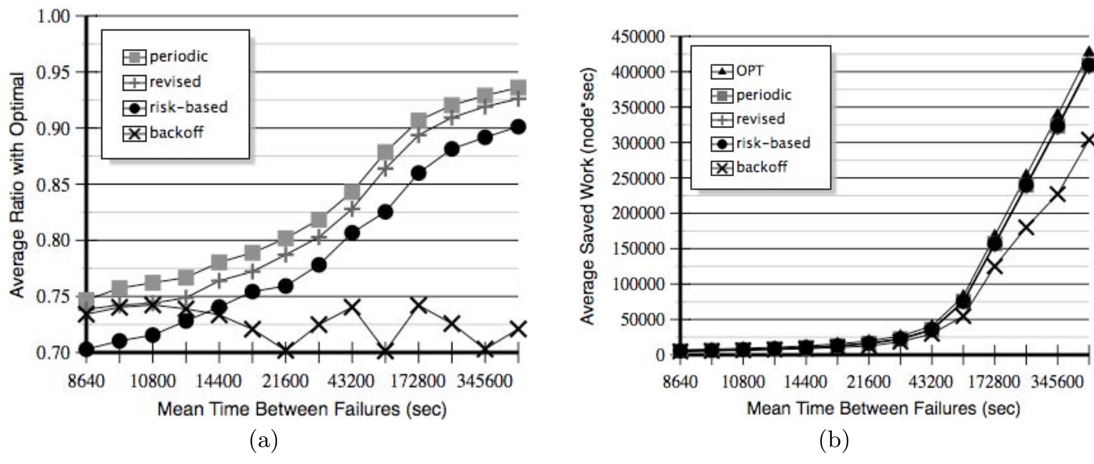

Figure 6: Uniform failure distribution. Risk-based checkpointing does uncharacteristically poorly, because the heuristic was derived under a memoryless assumption. Exponential backoff exhibits a sawtooth pattern; parameter variations cause a late checkpoint to fall before or after the failure, which has a factor of 2 impact on saved work and the ratio with the optimal.

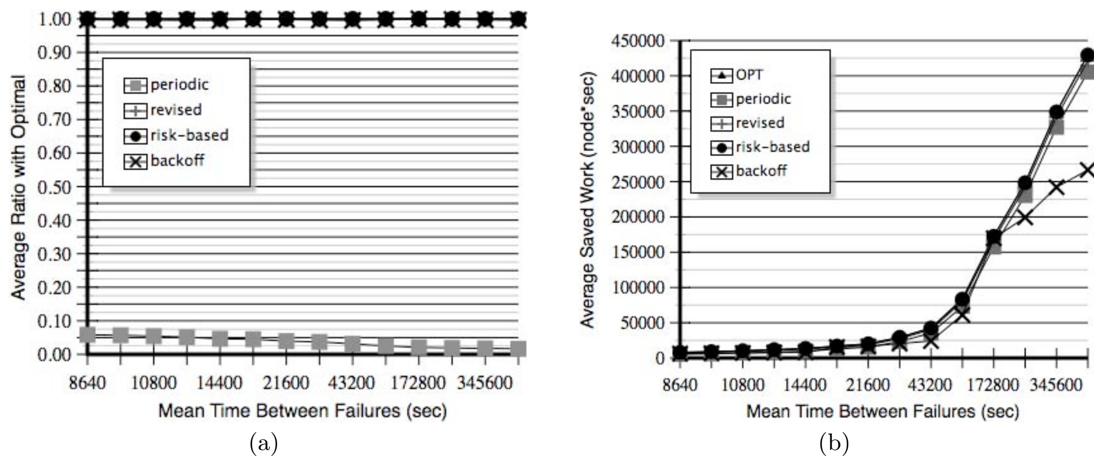

Figure 7: Composite uniform distribution. As predicted, periodic checkpointing is abysmally non-competitive against all the other algorithms in terms of the ratio with the optimal. Surprisingly, this is not apparent when looking at average saved work.

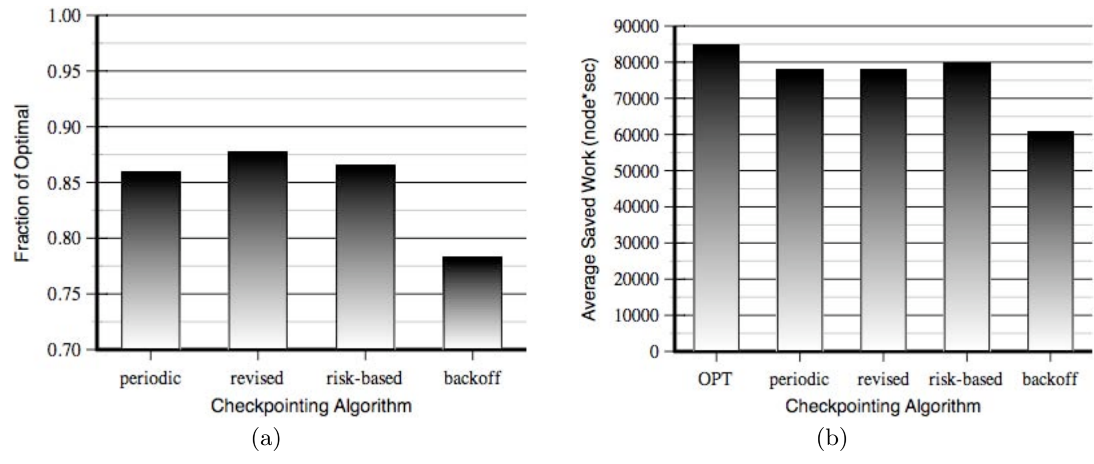

Figure 8: Failure trace from Blue Gene/L. Risk-based checkpointing takes second in the ratio, but comes closest to OPT in average saved work. Revised periodic checkpointing does better than periodic checkpointing, suggesting that performing the first checkpoint deterministically may be a good strategy in practice.

- Supports experimentally a number of theoretical results related to cooperative checkpointing, including the non-competitiveness of periodic checkpointing and the optimal competitiveness of simple cooperative checkpointing algorithms like backoff (Figure 7).
- Reveals interesting characteristics of cooperative checkpointing: optimally competitive algorithms may do worse than non-competitive algorithms under common failure distributions (Figure 6), and the competitive ratio may have no relationship with the expected rate of progress an application makes while remaining a relevant indicator of the quality of a checkpointing scheme (Figure 7).

The purpose of these experiments was not to argue the quality of one failure model over another, nor one cooperative checkpointing algorithm over another. The important conclusion is that periodic checkpointing lacks the flexibility to handle even a small variety of non-exponential traces or to scale with increasing failure rates and checkpoint overheads. Cooperative checkpointing provides that robustness in a manner that is practical, theoretically proven, and experimentally confirmed.

8. REFERENCES

[1] N. Adiga and T. B. Team. An overview of the bluegene/l supercomputer. In *Supercomputing, Technical Papers*, Nov. 2002.

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the Intl. Conf. on Supercomputing (ICS)*, pages 277–286, 2004.

[3] S.-E. Choi and S. J. Deitz. Compiler support for automatic checkpointing. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002.

[4] T. Dietterich and R. Michalski. Discovering patterns in sequence of events. In *Artificial Intelligence*, volume 25, pages 187–232, 1985.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, Houston, TX, Oct. 1992.

[6] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secur. Comput.*, 1(2):97–108, 2004.

[7] D. G. Feitelson. Parallel workloads archive. URL: http://cs.huji.ac.il/labs/parallel/workload/, 2001.

[8] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, New York, 1991.

[9] I. Lee, R. K. Iyer, and D. Tang. Error/failure analysis using event logs from fault tolerant systems. In *Proceedings of the 21st Intl. Symposium on Fault-Tolerant Computing*, pages 10–17, June 1991.

[10] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. K. Sahoo. Blue gene/l failure analysis and prediction models. In *Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN)*, 2006.

[11] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN)*, June 2005.

[12] A. N. Norman, S.-E. Choi, and C. Lin. Compiler-generated staggered checkpointing. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems (LCR)*, pages 1–8, New York, NY, USA, 2004. ACM Press.

[13] A. J. Oliner. Cooperative checkpointing for supercomputing systems. Master's thesis, Massachusetts Institute of Technology, 2005.

[14] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Cooperative checkpointing theory. In *Proceedings of IPDPS, Intl. Parallel and Distributed Processing Symposium*, 2006.

[15] A. J. Oliner, L. Rudolph, R. K. Sahoo, J. Moreira, and M. Gupta. Probabilistic qos guarantees for supercomputing systems. In *Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN)*, June 2005.

[16] A. J. Oliner and R. K. Sahoo. Evaluating cooperative checkpointing for supercomputing systems. In *IEEE IPDPS, Workshop on System Management Tools for Large-scale Parallel Systems*, Apr. 2006.

[17] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *IEEE IPDPS, Workshop on System Management Tools for Large-scale Parallel Systems*, Apr. 2005.

[18] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Proceedings of the 28th Intl. Symposium on Fault-tolerant Computing*, June 1998.

[19] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, November 2001.

[20] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the ACM SIGKDD, Intl. Conf. on Knowledge Discovery and Data Mining*, pages 426–435, August 2003.

[21] M. Schultz, 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. In *Supercomputing*, 2004.

[22] A. N. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. In *ACM Transactions on Computer Systems*, volume 110, pages 123–144, May 1984.

[23] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.