Cooperative Checkpointing Theory

Adam Oliner[1,4], Larry Rudolph[2], and Ramendra Sahoo[3]

[1]Stanford University
Dept. of Computer Science
Palo Alto, CA 94305 USA
oliner@cs.stanford.edu

[2]MIT
CSAIL
Cambridge, MA 02139 USA
rudolph@csail.mit.edu

[3]IBM
T.J. Watson Research Center
Hawthorne, NY 10532 USA
rsahoo@us.ibm.com

Abstract

Cooperative checkpointing *uses global knowledge of the state and health of the machine to improve performance and reliability by dynamically deciding when to skip checkpoint requests made by applications. Using results from cooperative checkpointing theory, this paper proves that periodic checkpointing is not expected to be competitive with the offline optimal. By leveraging probabilistic information about the future, cooperative checkpointing gives flexible algorithms that are optimally competitive. The results prove that simulating periodic checkpointing, by performing only every d^{th} checkpoint, is not competitive with the offline optimal in the worst case; a simple modification gives a provably competitive algorithm. Calculations using failure traces from a prototype of IBM's Blue Gene/L show an application using cooperative checkpointing may make progress 4 times faster than one using periodic checkpointing, under realistic conditions. We contribute an approach to providing large-scale system reliability through cooperative checkpointing and techniques for analyzing the approach.*

1. Introduction

Periodic checkpointing, the standard method for providing reliable completion of long-running jobs, is non-optimal for realistic failure distributions; *cooperative checkpointing*, in which checkpoint requests may be skipped, provides greater performance and reliability in the face of such distributions. With cooperative checkpointing [6, 7], the application programmer and the runtime system are both part of the decisions regarding when and how checkpoints are performed. Specifically, the programmer inserts checkpoints at locations in the code, perhaps where the application state is minimal, placing them liberally wherever a checkpoint would be efficient. At runtime, the application *requests* a checkpoint. The system finally *grants* or *denies* the checkpoint based on various heuristics, including disk or network usage and reliability information.

Checkpointing involves periodically saving a sufficient amount of the state of a running job to stable storage, allowing for that job to be restarted from the last successful checkpoint. Checkpoints have an associated overhead, usually dictated by the bottleneck to the stable storage system. Therefore, while there is a risk associated with not checkpointing, there is also a direct and measurable cost associated with performing the checkpoints. As systems grow larger, this overhead may increase due to the greater coordination necessary to guarantee a consistent checkpoint, more data that requires saving, and interference with other applications. Optimally, every checkpoint is used for recovery and every checkpoint is completed immediately preceding a failure. A central insight of cooperative checkpointing is that *skipping* checkpoints that are less likely to be used for recovery can improve reliability and 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 do not generally exhibit such failure behavior [5, 9, 11, 14]. Moreover, applications need to be recoded when they are ported to a system with different reliabilty characteristics. Cooperative checkpointing allows for irregular checkpoint intervals by giving the system an opportunity to skip requested checkpoints at runtime. Therefore, cooperative checkpointing may be thought of as a hybrid of

[4]Work was performed while a Master's student at the Massachusetts Institute of Technology.

1-4244-0054-6/06/$20.00

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 grossly inefficient or have a low probability of being used for recovery. If the heuristics used by the system are reasonably confident that a particular checkpoint should be skipped, a benefit is conferred to both parties. That is, the application may finish sooner or at a lower cost because checkpoints were performed at more efficient times, and the system may accomplish more useful work because fewer checkpoints were wasted.

This paper formally describes cooperative checkpointing and develops a theoretical model. The analysis of this model is a variation of competitive analysis, which uses a *value* function in place of a cost function. The model is used to prove that, in the worst case, periodic checkpointing is non-competitive; a small modification to periodic checkpointing yields a cooperative checkpointing algorithm that is competitive. Furthermore, we prove that in the expected case under certain failure distributions, naïve periodic checkpointing is not competitive with the offline optimal, but there is a simple cooperative checkpointing algorithm that is optimally competitive. It may be surprising to note that failure distributions found in practice exhibit these properties, and that the distribution observed on a prototype of IBM's Blue Gene/L (BG/L) implies that an application using cooperative checkpointing would have made progress four times faster than with optimally-placed periodic checkpoints.

2. Background

High performance computing systems are tending toward being larger and more complex. For example, a 64-rack 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 or months. Despite a design focus on reliability, failures on such a large-scale machine will be relatively frequent. Checkpointing is still the best solution for providing reliable completion of these jobs on inherently unreliable hardware. There is an I/O bottleneck facing these massive clusters when they attempt to save their state. It is clear that standard checkpointing techniques must be reevaluated [4].

In order for any reliability scheme to be effective, one must develop useful models of the failure behavior of supercomputers. Failure events in large-scale commodity clusters as well as the BG/L prototype have been shown [5, 14] to be neither independent, identically distributed, Poisson, nor unpredictable. A realistic failure model for large-scale systems should admit the possibility of critical event prediction. Only recently have these predictions been used effectively to improve system performance [3, 10]. The idea of using event prediction for proactive system management has also been explored [13, 15]. A hybrid algorithm [14] was able to predict critical failures with up to 70% accuracy on an AIX cluster with 350 nodes.

Checkpointing for computer systems has been a major area of research over the past few decades. There have been a number of studies on checkpointing based on certain failure characteristics [12], including Poisson distributions. Tantawi and Ruschitzka [18] developed a theoretical framework for performance analysis of checkpointing schemes. In addition to considering arbitrary failure distributions, they present the concept of an *equicost* checkpointing strategy, which varies the checkpoint interval according to a balance between the checkpointing cost and the likelihood of failure.

System-initiated checkpointing is a part of many large-scale systems. The system can checkpoint any application at an arbitrary point in its execution. It has been shown that such a scheme is possible for any MPI application, without the need to modify user code [17], and that it can be used to improve QoS [8]. Such an architecture has several disadvantages, however: implementation overhead, time linear in the number of nodes to coordinate the checkpoint, lack of compiler optimization for checkpoints, and a potentially large amount of state to save.

Application-initiated checkpointing is the dominant approach for most large-scale parallel systems. Agarwal et al [2] developed application-initiated checkpointing schemes for BG/L. There are also a number of studies reporting the effect of failures on checkpointing schemes and system performance. Most of these works assume Poisson failure distributions and fix a checkpointing interval at runtime. A thorough list can be found elsewhere [11], where a study on system performance in the presence of real failure distributions concludes that Poisson failure distributions are unrealistic. Similarly, a recent study by Sahoo et al [16], analyzing the failure data from a large scale cluster environment and its impact on job scheduling, reports that failures tend to be clustered around a few sets of nodes, rather than following a particular distribution. Only in the past year (2004) has there been a study on the impact of realistic large-scale cluster failure distributions on checkpointing [9]. Our main motivation for this work is to establish a theoretical basis for checkpointing in the presence of arbitrary failure distributions. Addi-

tional theoretical results can be found elsewhere [6], as can experimental results verifying the validity of the theory and demonstrating the robustness of cooperative checkpointing [7].

3. Cooperative Checkpointing

This section introduces basic terms and definitions related to checkpointing and reliability. The section concludes with the introduction of cooperative checkpointing, an approach to reliability that addresses many outstanding challenges.

3.1. 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) is defined such that job failure between times t and $t+L$ will force the job to restart from the previous checkpoint, rather than the current one; failure after time $t + L$ means the checkpoint was successful and the application can restart as though continuing execution from time t. It was shown [11] that L typically has an insignificant impact on checkpointing performance for realistic failure distributions. Therefore, this paper treats $C \approx L$.

There is a *downtime* parameter (D) which measures for how long a failed node is down and unable to compute, and a *checkpoint recovery* parameter (R) which is the time required for a job to restart from a checkpoint. Although R and D are included in this model, they are of no importance when performing a competitive analysis, so long as they are uncorrelated with the decisions made by the checkpointing algorithm.

Figure 1 illustrates typical application behavior. 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. Applications that run for weeks or months will have hundreds of these checkpoints, most of which will never be used.

From a system management perspective, the most valuable resource in a supercomputer system is node time. Define a unit of *work* to be a single node occupied for one second. That is, occupying n nodes for e seconds consumes work $n \cdot e$ node-seconds. 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*, which is the total execution time minus its wasted time. Saved work (or committed 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. This definition highlights an important observation: checkpointing wastes valuable time, so (ideally) it should be done only when it will be used in a rollback to reduce recomputation. The concept of saved work is critical to understanding the analysis of cooperative checkpointing in Section 4. Figure 1 shows some example executions that help illustrate the concepts of saved and wasted work.

3.2. 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.

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.

Cooperative checkpointing appears to an observer as irregularity in the checkpointing interval. If we model failures as having an estimable MTBF, but not much else, then periodic checkpointing is sensible (even optimal). But once these failures are seen to be predictable,

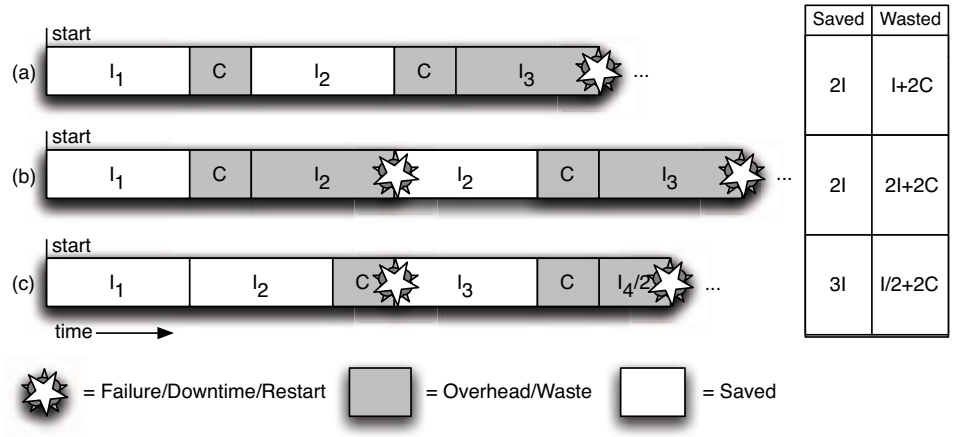

Figure 1. Three execution prefixes in which failures cause work to be lost, but checkpoints manage to save some work as well. The amount of work saved or lost in each fragement is listed.

or other factors are considered, this irregularity can be exploited. The behavior of applications as they choose to skip different checkpoints is illustrated in Figure 2.

The primary policy question with regard to cooperative checkpointing is, "How does the gatekeeper decide which checkpoints to skip?" A discussion of the possible heuristics the gatekeeper could employ is not included in this paper, but may involve such factors as network traffic, disk usage, the job scheduling queue, event prediction, logically connected components, and QoS guarantees. 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. This observation is central to cooperative checkpointing.

4. Algorithmic Analysis

Cooperative checkpointing can be understood and analyzed mathematically. This section presents a formalization that helps elucidate the intuition behind cooperative checkpointing, thereby empowering users to construct near-optimal checkpointing schemes for any kind of failure distribution. The model and metrics reveal that the ability to consider failure distributions other than exponentials permits users to make the system perform significantly better.

Although most competitive analyses use a cost function that represents how expensive operations are, we found it beneficial to use a *value function* that measures the benefit conferred by the algorithm.

Throughout this paper, many results and proofs are shortened or omitted; a more complete account of cooperative checkpointing theory can be found elsewhere [6].

4.1. Worst-Case Competitive Analysis

We model cooperative checkpointing by considering the execution of a program that makes a checkpoint request every I seconds. This request period I is a characteristic of the program, not the online algorithm. For example, say a checkpoint is requested at time r. If that checkpoint is skipped, the next request will be made at time $r + I$; if the checkpoint is taken, the next request will be made at $r + I + C$.

The analysis focuses on *failure-free intervals* (*FF intervals*), which are periods of *execution* between the occurrence of two consecutive failures. Such periods are crucial, because only that work which is checkpointed within an FF interval will be saved. Let F be a random variable with unknown probability density function. The varying input is a particular sequence of failures, $Q = \{f_1, f_2, \ldots, f_n\}$, with each f_i generated from F. Each f_i is the length of an FF interval, also written as $|FFI|$. The elements of Q determine when the program fails but not for how long the program is down. Thus, if execution starts at $t = 0$, the first failure happens at f_1. No progress is made for some amount of time following the failure. After execution begins again at time $t = t_1$, the second failure happens at $t = t_1 + f_2$. Figure 3 illustrates a partial execution including three FF intervals.

Note that these failure times are independent of the decisions made by the online algorithm, and that they happen at particular times rather than at specific points in the program. Therefore, the algorithm's

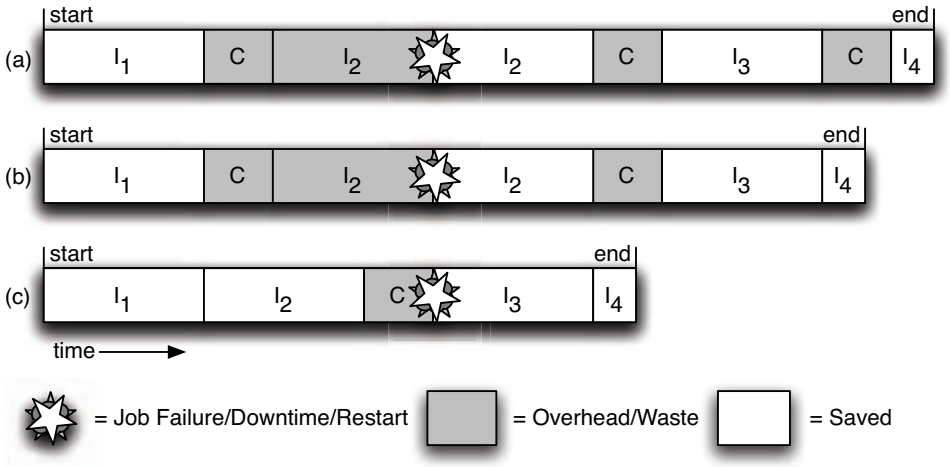

Figure 2. Three job runs in which different checkpoints are skipped. Run (a) shows typical periodic behavior, in which every checkpoint is performed. In run (b), the final checkpoint is skipped, perhaps because the critical event predictor sees a low probability that such a checkpoint will be used for 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.

decisions affect where in the execution the failure occurs; this may beneficently move a failure to just after a completed checkpoint or may detrimentally cause more work to be lost.

The average length of these intervals is related to the Mean Time Between Failures or MTBF. The f_i, however, do not include the downtime and recovery time. They are the periods between failures during which the program can make progress. This input Q is independent of the choices made by the checkpointing algorithm, and so f_i also includes checkpoint overheads.

Given knowledge of the past behavior of the program and the system, a cooperative checkpointing algorithm decides whether to grant or skip each checkpoint request, as it arrives. Let P be some program, A be some cooperative checkpointing algorithm, and Q be some failure sequence of n elements. Say P has infinite execution time, but n is finite, as are the elements $f_i \in Q$. Consider the period of execution starting at $t = 0$, before the first failure, and ending with the n^{th} failure (the *span* of Q).

Define $V_{A,Q}$ to be the cumulative amount of work saved by algorithm A during the time spanned by Q. When discussing an individual FF interval, it is acceptable to refer simply to V_A, which is the amount of work saved by A in that interval.

Definition 1 *An online checkpointing algorithm A has* competitive ratio α *(A is α-competitive) if, for every failure sequence Q, the amount of work saved by the optimal offline algorithm (OPT) is at most α times the amount of work saved by A. That is, $V_{OPT,Q} \leq \alpha V_{A,Q}$.*

It is worth emphasizing that the definition compares the quality of the algorithm in terms of the amount of work that was saved in an execution with worst-case failure behavior, rather than the work that is lost and recomputed. (When using lost work, most analyses devolve into considering infinite FF interval lengths, and do not give sensible relationships among checkpointing algorithms.) In a sense, this definition compares value instead of cost. When α is infinite, we say that A is *not competitive*. Work is typically defined to be execution time multiplied by the size of the program in nodes; for competitive analysis, let the program have unit size. Before discussing this definition in more detail, it is necessary to define the behavior of the optimal offline cooperative checkpointing algorithm.

Recall that the overhead for performing a checkpoint is a constant C for every checkpoint. Note that the downtime (D) and recovery time (R) are paid by every checkpointing algorithm after every element in Q.

Definition 2 *The* optimal offline cooperative checkpointing algorithm *(OPT) performs the latest checkpoint in each FF interval such that the checkpoint completes before the end of the interval (if one exists), and skips every other checkpoint.*

A *critical checkpoint* is any checkpoint that is used for recovery at least once. A *wasted checkpoint* is any checkpoint that is completed but never used for recovery, or which fails just as it is completing. In an FF interval in which more than one checkpoint is performed, the last checkpoint is a *critical checkpoint* and the rest

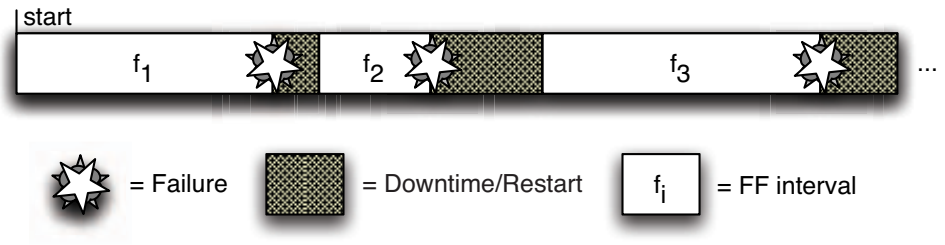

Figure 3. The execution of a program with failures, shown up to n = 3. The length of the FF intervals (f_i) varies. The downtime and recovery time following a failure is variable, as well, but is not included in the failure sequence Q. The execution may continue beyond what is shown.

are *wasted checkpoints*. Skipping a wasted checkpoint does not necessarily increase the amount of work that is saved in an FF interval, because doing so may or may not allow a later checkpoint to be performed. Later on, Lemma 1 formalizes how many checkpoints must be skipped to be advantageous. On the other hand, skipping a critical checkpoint will always result in less saved work, because rollback must then be done to an earlier checkpoint. In order for an algorithm to be made incrementally more like the optimal, two things can be done: skip wasted checkpoints and don't skip checkpoints that would make better critical checkpoints.

As defined, there are many failure sequences Q such that no checkpointing algorithm, including the optimal, will permit a program to make progress. According to Definition 1, however, there need not be progress with every failure sequence. More importantly, the worst-case input that is considered in the competitive analysis is not the worst-case input for OPT, but the input that gives the algorithm in question (A) the worst performance relative to OPT (highest ratio of V_{OPT} to V_A). For most executions in which A cannot make progress, neither can OPT.

The key parameters introduced so far are I and C.

Lemma 1 ***(Skipping Lemma)*** *Let A be some deterministic algorithm. Consider a particular FF interval length such that A performs k wasted checkpoints. As a result of skipping those checkpoints, $V_{OPT} \geq V_A + I\lfloor\frac{kC}{I}\rfloor$.*

Recall that both V_A and V_{OPT} consider the same period of time. The proof of the Skipping Lemma is simple, but its consequences are far-reaching. In particular, it means that the worst-case competitive ratios of most algorithms will be functions of $\lfloor\frac{C}{I}\rfloor$.

The case of $C > I$ is not purely academic, especially because I is the request interval, not necessarily the checkpoint interval. Because such a situation is realistic, the fact that the competitiveness is a function of $\lfloor\frac{C}{I}\rfloor$ is worthy of note. It would be desirable to achieve k-competitiveness for some constant k, independent of the relationship among the parameters. The Skipping Lemma forbids this.

Another general difficulty in analyzing the competitiveness is the challenge of identifying the worst-case input. So far, this input has been described as a sequence of failures (Q) such that $\frac{V_{OPT,Q}}{V_{A,Q}}$ is maximized. It turns out that it is not necessary to consider the worst-case sequence, but merely the worst-case interval length. The following results have been proved elsewhere [6]; they are restated here for reference.

Theorem 1 *Algorithm A is α-competitive iff, $\forall$ FF intervals of length $f \in \mathbb{R}^+$, the amount of work saved by OPT is at most α times the amount of work saved by A.*

Corollary 1 *To determine the competitive ratio of algorithm A, it is sufficient to consider the f for which the ratio of V_{OPT} to V_A is largest. That ratio is α.*

Theorem 2 *Let A be a deterministic cooperative checkpointing algorithm that skips the first checkpoint in every interval. A is not competitive.*

Theorem 3 *There does not exist a deterministic cooperative checkpointing algorithm that is better than $(2 + \lfloor\frac{C}{I}\rfloor)$-competitive.*

The proof of Theorem 2 involves constructing a failure sequence in which a failure always occurs just before $2I+C$ seconds have elapsed, when the optimal has checkpointed once but A has not.

We now turn to competitive analyses of periodic checkpointing algorithms. Specifically, we describe how cooperative checkpointing can be used to simulate periodic checkpointing, and prove that the naïve implementation is not competitive.

Consider a program that uses cooperative checkpointing where requests occur every I seconds. There is some desired periodic checkpointing interval (I_p) that

the online algorithm is trying to simulate. If I_p mod $I = 0$, then exact simulation is possible. When I_p mod $I \neq 0$, an approximation is sufficient; the algorithm uses some d such that $dI \approx I_p$. The algorithm should perform, roughly, one out of every d checkpoint requests.

Let $A_{n,d}$ be the naïve implementation of this simulation, in which, for any FF interval, the algorithm performs the d^{th} checkpoint, the $2d^{th}$ checkpoint, the $3d^{th}$ checkpoint, and so on.

Theorem 4 *$A_{n,d}$ is not competitive for $d > 1$.*

Proof $A_{n,d}$ deterministically skips the first checkpoint in every FF interval. By Theorem 2, $A_{n,d}$ is not competitive for $d > 1$. □

The case of $d = 1$ is special. In the previous proof, $A_{n,d}$ did not make progress because it skipped checkpoints that were critical checkpoints for OPT. When $d = 1$, however, no checkpoints are skipped. Indeed, this is a special cooperative checkpointing algorithm whose behavior is to perform every checkpoint request it receives. Define A_{all} to be the algorithm $A_{n,1}$. It has been proved [6] that this algorithm is optimally competitive. That proof further shows that, asymptotically, V_{OPT} grows in proportion to $|FFI|$.

The original intention, recall, was to simulate periodic checkpointing using cooperative checkpointing. A_{all} doesn't simulate periodic checkpointing so much as it *is* periodic checkpointing. Instead, consider the following variation of $A_{n,d}$ that also performs only every d^{th} checkpoint, with a small change to avoid running up against Theorem 2.

Let $A_{p,d}$ (note the change in subscript) be a cooperative checkpointing algorithm that simulates periodic checkpointing by *always performing the first checkpoint*, and subsequently performing only every d^{th} checkpoint. As above, $d \approx \frac{I_p}{I}$ and $d > 0$, where I is the request interval and I_p is the periodic checkpointing interval that is being simulated. $A_{p,d}$ performs the 1^{st} checkpoint, the $(d+1)^{th}$ checkpoint, the $(2d+1)^{th}$ checkpoint, and so on.

Theorem 5 *$A_{p,d}$ is $(d + 1 + \lfloor \frac{C}{I} \rfloor)$-competitive.*

Proof Set $f = |FFI| = (d + 1)I + 2C$ such that $A_{p,d}$ performs the first checkpoint, skips $d - 1$ checkpoints, and fails just before completing the $(d + 1)^{th}$ request. $V_p = I$. As with A_{all}, the exact number of intervals OPT performs before taking its single checkpoint depends on the relationship between C and I: $V_{OPT} = (d + 1)I + \lfloor \frac{C}{I} \rfloor I$. The ratio for this interval length f is $d + 1 + \lfloor \frac{C}{I} \rfloor$.

Again, we must consider the asymptotic behavior. In order to increase V_{OPT} by dI, it is necessary to increase f by exactly dI. To increase V_p by the same amount (dI), f must be increased by $dI + C$ to accommodate the additional checkpoint. The asymptotic ratio of V_{OPT} to $A_{p,d}$ is $\frac{dI+C}{dI} = 1 + \frac{C}{dI}$. This is always strictly less than $d+1+\lfloor \frac{C}{I} \rfloor$, so $f = (d+1)I+2C$ was the worst-case interval.

By Corollary 1, $A_{p,d}$ is $(d+1+\lfloor \frac{C}{I} \rfloor)$-competitive. □

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 first digit is 1 if the first checkpoint should be performed and 0 if it should be skipped. All the algorithms we have considered so far can be easily encoded in this way: $A_{all} = \{1, 1, 1, 1, \ldots\}$, $A_{n,2} = \{0, 1, 0, 1, 0, 1, \ldots\}$, $A_{p,3} = \{1, 0, 0, 1, 0, 0, 1, \ldots\}$.

An upper bound on the length of the FF interval is easily given the program's running time, so there is also a bound on the number of checkpoints and the length of these binary sequences. Consequently, each member of this finite set of deterministic algorithms can be identified by a little-endian binary number.

Let A_{2x} be a cooperative checkpointing algorithm that doubles V_{2x} at the completion of each checkpoint. In each FF interval, it performs the 1^{st}, 2^{nd}, 4^{th}, 8^{th}, etc. checkpoints:

$$A_{2x} = \{1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, \ldots\}$$

It has been shown that A_{2x} is $(2+\lfloor \frac{C}{I} \rfloor)$-competitive. The intention in presenting this algorithm is to highlight a characteristic of worst-case competitive analysis, which is that a number of very different algorithms can all be optimally competitive.

4.2. Expected-Case Competitive Analysis

This section proposes a more practical form of competitive analysis in which the algorithm can consider the actual failure distribution. The algorithms remain deterministic, but, with this information and the refined model, the analysis is significantly different.

Let F be a random variable whose value is the length of the failure-free interval and let $\chi(t)$ be the probability density of F. That is, $P(a \leq F \leq b) = \int_a^b \chi(t)dt$, and assume that $F \geq 0$, $\chi(t) \geq 0 \forall t$, and $\int_0^\infty \chi(t)dt = 1$. Properties of this probability distribution, like mean ($\mu = E(F)$), variance (σ), and standard deviation, are calculated in the usual way.

In the previous section, the offline optimal knew in advance that a failure would happen after executing for f seconds (the length of the FF interval), and was effectively using

$$\chi(t) = \delta(t - f) \tag{1}$$

where $\delta(t)$ is the Dirac delta function. The checkpointing algorithm knew nothing of this distribution, however, and was forced to choose a strategy that minimized the worst-case ratio.

Determining the density function in practice can be accomplished by using historical data to constantly refine an empirical distribution.

Every deterministic cooperative checkpointing algorithm (A) has a characteristic work function, $W_A(t)$. This function specifies, for all times t within an FF interval, how much work A has saved. More to the point, if $|FFI| = f$, then $V_A = W_A(f)$. The beginning of an FF interval is always $t = 0$. The work function will be used along with $\chi(t)$ to calculate the expected competitive ratio of the algorithm.

Work functions are nondecreasing, irregularly-spaced staircase functions, i.e.:

$$W_A(t) = 0, \;\; t \leq I + C \qquad\qquad W_A(t) = nI, \;\; n \in \mathbb{Z}^*$$

Lemma 2 *Let k be the number of checkpoints completed in a given FF interval by algorithm A at time t. Then $I\lfloor\frac{t-kC}{I}\rfloor \geq W_A(t) \geq kI$.*

The proof can be found elsewhere [6]. The work function for OPT is $W_{OPT}(t)$. This function is unique:

$$W_{OPT}(t) = I\lfloor\frac{t - C}{I}\rfloor$$

$W_{OPT}(t)$ gives an upper bound for the work functions of all other algorithms: $W_{OPT}(t) \geq W_A(t)\ \forall t$.

In the worst-case competitive analysis, recall that OPT was nondeterministic and had absolute knowledge about when the FF interval would end. Similarly, this work function for OPT does not obey the rules to which deterministic algorithms are bound. For example, after increasing by I, $W_A(t)$ cannot increase again until at least $I + C$ seconds later. $W_{OPT}(t)$, on the other hand, increases by I every I seconds. This is equivalent to an OPT that knows $\chi(t)$ as the function in Equation 1 and waits until the latest possible time before performing a single checkpoint.

At the beginning of a failure free interval, the cooperative checkpointing scheme selects some deterministic algorithm A based on what it knows about $\chi(t)$ for this interval. How this selection process proceeds is derived from the definition of *expected competitiveness* and the calculation of the expected competitive ratio ω.

Definition 3 *An online checkpointing algorithm A has* expected competitive ratio ω *(A is ω-competitive) if the expected amount of work saved by the optimal offline algorithm (OPT) is at most ω times the expected amount of work saved by A. That is, $E[V_{OPT}] \leq \omega E[V_A]$.*

By the definition of V_A, if a failure happens at time $t = h$, $V_A = W_A(h)$. Let T be the maximum FF interval length, easily determined by the running time of the program. We can now define ω in terms of finite integrals over the products of piecewise continuous functions and probability densities:

$$\omega = \frac{E[V_{OPT}]}{E[V_A]} = \frac{\int_0^T I\lfloor\frac{t-C}{I}\rfloor\chi(t)dt}{\int_0^T W_A(t)\chi(t)dt} \tag{2}$$

This concludes the presentation of the model and analysis tools. The following section investigates the implications of these results both for theory and for practice.

5. Conclusions and Contributions

Among the results in this section is a proof that naïve periodic checkpointing can be arbitrarily bad relative to cooperative checkpointing for certain failure distributions, and a case analysis demonstrating that, under realistic conditions, an application using cooperative checkpointing can make progress four times faster than one using periodic checkpointing.

The cooperative checkpointing scheme now behaves as follows. At the beginning of each FF interval, the system considers everything it knows about $\chi(t)$ and selects a deterministic algorithm A that maximizes the overlap between $W_A(t)$ and $\chi(t)$. Given some information about $\chi(t)$, A should be chosen to maximize the denominator in the equation for ω, because we want that quotient to be as small as possible. Intuitively, we want to match up $W_A(t)$ and $\chi(t)$.

The expected competitiveness model for checkpointing allows all kinds of distributions, making it far more general than a scheme that presumes a memoryless failure distribution. How powerful is this generality? Consider F, distributed with some failure probability density $\chi(t)$. Let A_λ be a periodic checkpointing algorithm; restrict A_λ, however, to only have access to an exponential distribution $\chi_\lambda(t) = \lambda e^{-\lambda t}$ with $\frac{1}{\lambda} = E[F]$. A_λ always has an accurate measure of the mean of the distribution $\chi(t)$, but may otherwise have no similarities. For example, the exponential variance will be $\frac{1}{\lambda^2}$ while the variance of $\chi(t)$ may be infinite. OPT, as usual, knows $\chi(t)$. Pick $\chi(t)$ such that the expected competitiveness is worst-case.

Theorem 6 *A_λ is not competitive.*

Proof Pick $\chi(t)$ to be the sum of two Dirac delta functions at t_1 and t_2: $\chi(t) = a\delta(t-t_1) + (1-a)\delta(t-t_2)$ with $0 \leq a \leq 1$. The mean of this distribution is $\frac{1}{\lambda} = E[F] = at_1 + (1-a)t_2$. Set $t_1 = I + C$. OPT will always make progress, because t_1 is large enough for OPT to save at least that one request interval before failing. To give A_λ every advantage, let it checkpoint with the optimal period: $I_{OPT} = \sqrt{\frac{2C}{\lambda}}$.

Make A_λ checkpoint with period $\geq I$; use $2I$ to be safe:

$$I_{OPT} = \sqrt{\frac{2C}{\lambda}} = \sqrt{2CE[F]} = \sqrt{2C(a(I+C) + (1-a)t_2)} \geq 2I$$

Isolating the variables over which we have control:

$$a(I+C) + (1-a)t_2 \geq \frac{2I^2}{C} \tag{3}$$

Using Equation 3 as the constraint, we want to make a as close to 1 as possible, so that the FF interval almost always ends after $I + C$, when OPT has made progress but A_λ has not. As $a \to 1$, $t_2 \to \infty$ in order to maintain the inequality (unless $C > I$). As before, T is the maximum length of an FF interval. If we allow T to go to infinity, A_λ will not make progress arbitrarily often, though OPT always will. The competitive ratio approaches infinity. □

Remark When a is set to 1, $\chi(t)$ is really just the worst-case failure scenario. Having set the periodic checkpointing interval to $2I$, this is forcing A_λ to be a deterministic algorithm that skips the first checkpoint. By Theorem 2, the algorithm is not competitive.

Corollary 2 *By Theorem 6, a periodic checkpointing algorithm that assumes an exponential failure distribution may be arbitrarily bad compared to a cooperative checkpointing algorithm that permits general probability density functions as failure distributions.*

In light of this result, it is worth asking how badly a real machine might do by using periodic checkpointing with an assumption of exponentially distributed failures, versus how it might do by using cooperative checkpointing. What is the potential gain under realistic conditions? Does anything like the failure distribution in the proof of Theorem 6 ever occur in practice? We repeat the comparison from Theorem 6 using real parameters from IBM's BG/L supercomputer.

First, set $C = 6$ minutes (360 seconds); the upper bound for checkpointing on BG/L was estimated at 12 minutes (720 seconds) so this is a reasonable average. Second, assume that the random variable F, the length of each FF interval, is independent and identically distributed with exponential distribution $\chi(t) = \lambda e^{-\lambda t}$. Third, consider an installation of BG/L that consists of 64 racks with a total of 65,536 nodes.

In practice, Sahoo et al [5] estimate the mean time between failures for a 4,096-node BG/L prototype to be 3.7 failures per day (MTBF = 6.5 hours). Presuming linear scaling, the 64-rack machine will have $E[F] =$ 24 minutes (1,459 seconds). The MTBF corresponds roughly to a 3-year component lifetime. Recall that this data is from a prototype, so the full machine may be more reliable than we project.

On the full BG/L machine, the optimal periodic checkpointing algorithm would checkpoint every 17 minutes (1,025 seconds). With an overhead of 6 minutes, it would spend 26% of the machine time performing checkpoints.

In order to comment on the performance of cooperative checkpointing, we must hypothesize a non-exponential failure distribution that might better describe BG/L's behavior. Use $\chi(t) = a\delta(t-t_1) + (1-a)\delta(t-t_2)$. In the prototype study [5], the maximum uptime was slightly more than 140 hours (504,000 seconds), far larger than the mean. This real data already echoes the construction of the proof of non-competitiveness for A_λ [6]; pick t_1 to be small, t_2 to be very large, and a to be nearly 1. Use $C = 360$ seconds. As before, pick I to be half the optimal periodic checkpointing interval, meaning that A_λ performs every other request starting with the second: $I = 512$ seconds. Set $E[F]$ at 1,459 seconds, t_2 at 504,000 seconds, and t_1 at $I + C = 872$ seconds:

$$E[F] = 1459 = at_1 + (1-a)t_2 = 872a + (1-a)504000$$

Which fixes a at 0.9988. In other words, 99.88% of the time the application fails at time $I + C$, a situation in which A_λ saves no work, but OPT saves I. The remaining 0.22% of the time,

$$V_\lambda = I\lfloor\frac{t_2}{2I+C}\rfloor = 512\lfloor\frac{504000}{1384}\rfloor = 186368 \texttt{ units}$$
$$V_{OPT} = I\lfloor\frac{t_2 - C}{I}\rfloor = 512\lfloor\frac{503640}{512}\rfloor = 503296 \texttt{ units}$$

The expected ratio for this example case is,

$$\omega = \frac{V_{OPT}}{V_\lambda} = \frac{0.9988(512) + 0.0022(503296)}{0.0022(186368)} = 3.95$$

This means roughly that, in an infinite execution under these system parameters and this failure distribution, cooperative checkpointing can accomplish 4 times as

much useful work in a given amount of machine time, compared to even the optimal periodic checkpointing algorithm. Certainly, increasing the throughput of the machine four-fold would be a worthwhile investment. This example illustrates that cooperative checkpointing is not merely of theoretical value; it can result in tangible improvements in reliability and performance over periodic checkpointing.

This paper makes the following contributions:

- Introduces cooperative checkpointing, where the application requests checkpoints and the system dynamically decides which to perform and which to skip.
- Proposes a model and metrics for checkpointing. The model allows for the creation of near-optimal checkpointing schemes for general failure distributions.
- Analyzes cooperative checkpointing as an online algorithm. The worst-case and expected-case analyses prove that cooperative checkpointing can do arbitrarily better than periodic checkpointing.
- Argues that empirical data suggest cooperative checkpointing would confer significant benefits in practice, with one case analysis projecting performance improvement by a factor of four.

References

[1] N. Adiga and T. B. Team. An overview of the bluegene/l supercomputer. In *SC2002, 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 *ICS 2004*, pages 277–286, 2004.

[3] P. Dinda. A prediction-based real-time scheduling advisor. In *IPDPS*, 2002.

[4] 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.

[5] 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.

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

[7] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Robustness of cooperative checkpointing. In *Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN), (submitted)*, June 2006.

[8] 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.

[9] 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.

[10] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *IEEE IPDPS, Intl. Parallel and Distributed Processing Symposium*, Apr. 2004.

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

[12] 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.

[13] R. K. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, and M. Gupta. Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems. In *SHAMAN, Workshop, ICS'02*, New York, June 2002.

[14] 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.

[15] R. K. Sahoo, I. Rish, A. J. Oliner, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Autonomic computing features for large-scale server management and control. In *AIAC Workshop, IJCAI 2003*, Aug. 2003.

[16] R. K. Sahoo, A. Sivasubramanian, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 772–781, June 2004.

[17] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

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