Cooperative Checkpointing for Supercomputing Systems

by

Adam Jamison Oliner

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by .....
José E. Moreira
VI-A Company Thesis Supervisor

Certified by .....
Larry Rudolph
M.I.T. Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Cooperative Checkpointing for Supercomputing Systems

by

Adam Jamison Oliner

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

A system-level checkpointing mechanism, with global knowledge of the state and health of the machine, can improve performance and reliability by dynamically deciding when to skip checkpoint requests made by applications. This thesis presents such a technique, called *cooperative checkpointing*, and models its behavior as an online algorithm. Where C is the checkpoint overhead and I is the request interval, a worst-case analysis proves a lower bound of $(2 + \lfloor \frac{C}{I} \rfloor)$-competitiveness for deterministic cooperative checkpointing algorithms, and proves that a number of simple algorithms meet this bound. Using an expected-case analysis, this thesis proves that an optimal periodic checkpointing algorithm that assumes an exponential failure distribution may be arbitrarily bad relative to an optimal cooperative checkpointing algorithm that permits a general failure distribution. Calculations suggest that, under realistic conditions, an application using cooperative checkpointing may make progress 4 times faster than one using periodic checkpointing. Finally, the thesis suggests an embodiment of cooperative checkpointing for a large-scale high performance computer system and presents the results of some preliminary simulations. These results show that, in extreme cases, cooperative checkpointing improved system utilization by more than 25%, reduced bounded slowdown by a factor of 9, while simultaneously reducing the amount of work lost due to failures by 30%. This thesis contributes a unique approach to providing large-scale system reliability through cooperative checkpointing, techniques for analyzing the approach, and blueprints for implementing it in practice.

Thesis Supervisor: José E. Moreira
Title: VI-A Company Thesis Supervisor

Thesis Supervisor: Larry Rudolph
Title: M.I.T. Thesis Supervisor

Acknowledgments

This thesis would not have been possible without the wisdom and support of my advisor, Larry Rudolph, my managers, José Moreira and Manish Gupta, and, of course, my mentor, Ramendra Sahoo. I will be forever grateful to Larry for teaching me how to do academic research, to Ramendra for teaching me how to do industrial research, and to José and Manish for making my time at IBM so fun and productive. In addition to being a mentor, Ramendra has also been a generous and thoughtful friend, who made me feel like a part of his family.

I am indebted to IBM and the T.J. Watson Research Center for giving me the time and resources to pursue this research; I was given more flexibility, access, and help than I could have asked for. The community at IBM, specifically in the BlueGene System Software Group, made me feel welcome and important. It was a thrill being a part of that project, and an honor to work with so many amazing people.

My time at IBM would never have come about, however, were it not for the MIT VI-A program. The VI-A program gave me the chance to experience research from both the academic side and the industrial side. It was truly a unique opportunity. I am grateful to Markus Zahn for his role as the director of that program. A thousand thanks to Kathy Sullivan for her tireless work making my VI-A experience so fantastic.

I am indebted to the many teachers and mentors I have had at MIT. Jimmy Lin for getting me started at the lab formerly known as the MIT AI Lab, and Boris Katz for his support and wisdom during my time as a researcher in natural language processing. Many thanks to Patrick Winston, who has played the roles of my teacher, my employer, my colleague, and my friend.

Finally, I would like to thank my family and friends for their love and support. This thesis is dedicated to my parents.

Contents

1 Introduction **13**

2 Cooperative Checkpointing **17**

2.1 Terms and Definitions 18

2.2 Failure Behavior 21

2.2.1 Event Prediction 23

2.2.2 Fault-aware Job Scheduling 24

2.3 Checkpointing 24

2.3.1 System-Initiated Checkpointing 25

2.3.2 Application-Initiated Checkpointing 26

2.3.3 Characteristics 26

2.4 Cooperative Checkpointing 28

2.4.1 Policy 29

2.4.2 Characteristics 30

3 Algorithmic Analysis **33**

3.1 Worst-Case Competitive Analysis 34

3.1.1 Offline Optimal and Cooperative Checkpointing 36

3.1.2 Periodic Checkpointing 41

3.1.3 Exponential-Backoff Algorithms 43

3.1.4 Comments 45

3.2 Expected Competitive Analysis 45

3.2.1 Failure Probability Density 45

3.2.2 Work Function 47
3.2.3 Competitive Ratio 49
3.2.4 Example Analyses 51
3.3 Dynamic Competitive Analysis 58
3.3.1 Example Analysis 61

4 Embodiment **63**
4.1 System Description 63
4.2 Infrastructure 65
4.2.1 System Health Monitor and Modeler 65
4.2.2 Event Predictor 66
4.2.3 Gatekeeper 67
4.3 Simulations 68
4.3.1 Simulation Environment 68
4.3.2 Workload and Failure Traces 70
4.3.3 Results 71

5 Contributions **89**

List of Figures

2-1 Typical Job Behavior 19
2-2 Saved and Wasted Work 20
2-3 Job Behavior with Cooperative Checkpointing 29

3-1 Failure-Free Intervals 34
3-2 Example Executions 37
3-3 Work Functions 48

4-1 Basic System Structure 64
4-2 System with Cooperative Checkpointing 65
4-3 Bounded Slowdown, SDSC, $C = 720$ seconds 73
4-4 Bounded Slowdown, NASA, $C = 720$ seconds 74
4-5 Bounded Slowdown, SDSC, Torus, $C = 720$ seconds 75
4-6 Bounded Slowdown, SDSC, $C = 720$ seconds (zoomed) 76
4-7 Bounded Slowdown, NASA, $C = 720$ seconds (zoomed) 77
4-8 Bounded Slowdown, SDSC, Torus, $C = 720$ seconds (zoomed) 78
4-9 Bounded Slowdown, SDSC, $C = 3600$ seconds 79
4-10 Response Time, SDSC, $C = 720$ seconds 80
4-11 Wait Time, SDSC, $C = 720$ seconds 81
4-12 System Utilization, SDSC, Torus, $C = 720$ seconds 82
4-13 System Utilization, SDSC, Torus, $C = 720$ seconds (zoomed) 83
4-14 Lost Work, SDSC, $C = 720$ seconds 84
4-15 Lost Work, SDSC, Torus, $C = 720$ seconds 85
4-16 Lost Work, SDSC, Torus, $C = 720$ seconds (zoomed) 86

4-17 Policy Comparison: Bounded Slowdown, SDSC 86
4-18 Policy Comparison: System Utilization, SDSC 87
4-19 Policy Comparison: Lost Work, SDSC 87

List of Tables

2.1 Standard Checkpointing Characteristics 27
2.2 Cooperative Checkpointing Characteristics 31
4.1 Workload Characteristics. 71

Chapter 1

Introduction

"It is through cooperation, rather than conflict, that your greatest successes will be derived."
- Ralph Charell

A system-level checkpointing mechanism, with global knowledge of the state and health of the machine, can improve performance and reliability by dynamically deciding when to skip checkpoint requests made by applications. Consequently, the debate regarding which agent (the application or the system) should be responsible for deciding when to perform these checkpoints is a false dichotomy. This thesis proposes a system called *cooperative checkpointing*, in which the application programmer, the compiler, and the runtime system are all part of the decision 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 knows 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 finally *grants* or *denies* the checkpoint based on various heuristics. These may include disk or network usage information, reliability statistics, and so on.

High-performance computing systems typically exploit massive hardware par-

allelism to gain performance, at the cost of the applications being more failure prone. For example, the installation of IBM's BlueGene/L supercomputer (BG/L) at Lawrence Livermore National Labs (LLNL) will have 65,536 nodes [1], and millions of components. The failure of an individual hardware or software component in a running job's partition can often cause the failure of that job. We refer to any such component failure simply as a *failure*. Because components do not fail independently, the probability of a job failing grows superlinearly with the number of nodes in its partition; such growth outpaces the improvements in component reliability. Jobs on these systems may run for weeks or months at a time, so it is vital to minimize the amount of recomputed work following a failure. The lack of an effective method for mitigating the cost of failures would be catastrophic.

A standard solution to this problem is to implement a checkpointing (and rollback-recovery) scheme. Checkpointing involves periodically saving the state of a running job to stable storage, allowing for that job to be restarted from the last successful checkpoint in the event of a failure. 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 a direct and measurable cost associated with performing the checkpoints. As systems grow larger, this overhead increases. That is because there is greater coordination necessary to guarantee a consistent checkpoint, as well was more data that requires saving. The central question in the checkpointing literature asks, "When should a job perform checkpoints?" Optimally, every checkpoint is used for recovery and every checkpoint is completed immediately preceding a failure. A central insight of this thesis is that *skipping* checkpoints that are less likely to be used for recovery can improve reliability and performance.

The literature generally advises to checkpoint periodically, at an interval determined primarily by the overhead and the failure rate of the system. These schemes can be further subdivided into two distinct categories: application-initiated and system-initiated checkpointing. In application-initiated checkpointing, the job itself is ultimately responsible for determining when checkpoints are performed. The application programmer places checkpoints in the code in a quasi-periodic manner, often corre-

sponding to iterations of an outer loop. These checkpoints are performed irrespective of system-level considerations, such as network traffic. System-initiated checkpointing is precisely the opposite: the system forces the job to pause and saves its entire state. These checkpoints require greater coordination overhead, and are often needlessly large.

Working to decide between application-initiated and system-initiated checkpointing, however, is to beg the question. Both the application and the system have information relevant to the value of a checkpoint; to ignore either one is to relinquish a potential gain. The system presented in this thesis may be thought of as a hybrid between application-initiated and system-initiated checkpointing, because it permits cooperation between two such mechanisms. The application requests checkpoints, and the system either grants or denies each one. Currently, all application-initiated checkpoints are taken, even if system-level considerations would have revealed that the checkpoint is inadvisable. (And vice versa.) Many checkpoints are taken even though they 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 thesis introduces and describes cooperative checkpointing, develops a theoretical analytical model, and addresses the questions involved with implementing it in practice. The chapters of the thesis answer the following broad questions in order:

- What are some of the challenges related to reliability and performance in large-scale systems, and what approaches have previously been attempted to address them? What is cooperative checkpointing and how does it confer the advantages of previous approaches while simultaneously avoiding many of their pitfalls? (Cooperative Checkpointing, Chapter 2)

- How can we model and analyze cooperative checkpointing? Under what con-

ditions is cooperative checkpointing provably better than other methods, and how much better is it? (Algorithmic Analysis, Chapter 3)

- How might cooperative checkpointing be used in a real-world system? What supporting infrastructure does it require? What kind of performance and reliability improvements might be seen in practice? (Embodiment, Chapter 4)

The final chapter (Contributions, Chapter 5) summarizes the results, outlines some open research questions, and reviews the unique contributions of this thesis.

Chapter 2

Cooperative Checkpointing

"It is one of the beautiful compensations of this life that no one can sincerely try to help another without helping himself."
- Charles Dudley

In order to understand and appreciate cooperative checkpointing, it is necessary to understand the terminology, challenges, and previous approaches to providing reliability through checkpointing. This chapter contains a summary of recent results regarding checkpointing techniques and of the failure behavior of real systems. In addition, it introduces basic terms and definitions related to checkpointing and reliability. The material presented here will suggest two important trends in supercomputing systems, with implications for checkpointing:

- Increasing system complexity will necessitate more frequent checkpoints.
- Increasing system size will imply larger checkpointing overheads.

Taken together, these trends imply that periodic checkpointing will not be feasible as a long-term solution to system failures for providing reliability. The chapter concludes with the introduction of cooperative checkpointing, a unique approach to checkpointing that addresses these challenges.

High performance computing systems are tending toward being larger and more complex. For example, a 64-rack BlueGene/L system contains 65,536 nodes and more than 16 terabytes of memory. Applications on these systems are designed to run for days to 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. Unfortunately, the rate at which the data sets are growing is outpacing growth in the speed of stable storage and networks. In other words, there is an I/O bottleneck facing these massive clusters when they attempt to save their state. Taken together, it is clear that standard checkpointing techniques must be reevaluated [10].

Many high-performance computing applications are executed repetitiously [16], making their behavior amendable to modeling. After a single run of the application, a plethora of information can be harvested to improve the performance of future runs. For example, users will be able to more accurately estimate the running time of the application if the system can say, "Last time you ran this job on input X, it took time T." Memory usage, caching strategies, physical thread layout on the machine, and so on, can all be improved by learning behaviors from previous runs of a job. This kind of application profiling can provide the user and system with information that can be useful for improving both the performance and utility of the machine.

2.1 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. There is a *downtime* parameter (D) which measures for how long a failed node is down and unable to compute. For software errors that simply cause an application to crash, D may be negligible. If, instead, the failure is the permanent destruction of a critical hardware component, and no spare components can be used in its place, the downtime may be much longer.

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. This overhead may be treated either as a constant (noted as C) or as being dependent on the system conditions (C_i for some i). This thesis addresses both cases. 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 [22] that L typically has an insignificant impact on checkpointing performance for realistic failure distributions. Therefore, this thesis treats $C \approx L$. There is also a *checkpoint recovery* parameter (R) which is the time required for a job to restart from a checkpoint.

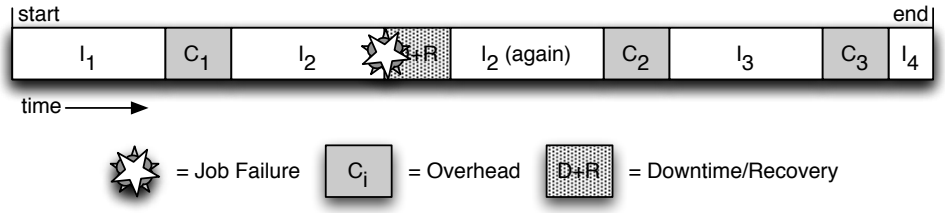

Figure 2-1: Behavior of a job that checkpoints roughly periodically. In this example, a job failure occurs just before the start of the second checkpoint, requiring I_2 to be recomputed. The intervals are not drawn to scale.

Figure 2-1 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. Because they are not used for rollback, C_2 and C_3 are also wasted. Applications that run for weeks or months will have hundreds of these checkpoints, most of which will never be used. One unfortunate characteristic of assuming a Poisson failure distribution is that any checkpoint is modeled as being equally likely to be used for rollback; making it difficult to argue for some being more important than others until after the fact.

From a system management perspective, the most valuable resource in a super-

computer system is node time. 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. For the purposes of job scheduling and checkpointing, work is the constrained resource for which we optimize. Thus, a node sitting idle, recomputing work lost due to a failure, or performing a checkpoint is considered *wasted work*. Similarly, *saved work* is node time spent performing calculations required by the job that are successfully saved by a checkpoint, or by the job's completion. Saved work never needs to be recomputed. Checkpointing overhead 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.

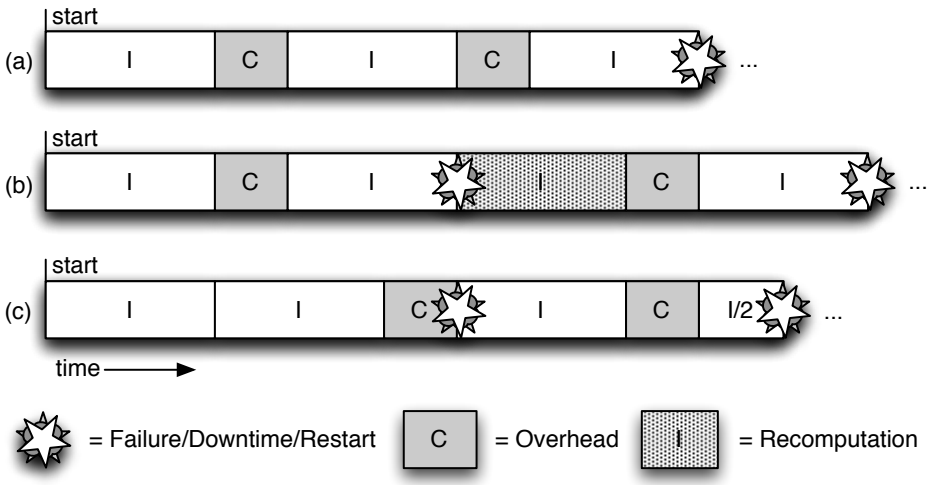

Figure 2-2: Three execution prefixes in which failures cause work to be lost, but checkpoints manage to save some work as well. The length of each interval in seconds is denoted by the inset text.

The concept of saved work is critical to understanding the analysis of cooperative checkpointing in Chapter 3. Figure 2-2 shows some example executions that help illustrate the concepts of saved and wasted work. Run (a) shows two intervals of

length I seconds being executed and promptly checkpointed with overhead C seconds. These two intervals were checkpointed before a failure occurred, so they count toward saved work; the overheads do not. The execution ends with an interval of work being computed, but a failure occurs before it can be saved. That work is wasted. In all, execution (a) contains $I + 2C$ units of wasted work and $2I$ units of saved work. Run (b) contains two failures. The first failure happens just after an interval of executed work, but before that interval can be checkpointed. That interval must be recomputed. That recomputation is saved, and counts toward saved work. Thus, run (b) shows $2I + 2C$ units of wasted work, and $2I$ units of saved work. Finally, execution (c) performs two intervals of work before checkpointing, and a failure occurs immediately *after* that checkpoint is completed. As a result, both intervals are saved, and nothing must be recomputed. The second failure happens midway through an interval, meaning those $\frac{I}{2}$ units of work are lost. In all, run (c) gives $\frac{I}{2} + 2C$ units of wasted work and $3I$ units of saved work.

2.2 Failure Behavior

Algorithmic improvements or increased hardware resources are often overshadowed by reliability issues. 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 [6, 7, 8, 14]. There have been several approaches to dealing with reliability problems, including managing system failures. Most theoretical work focuses on providing fail-over mechanisms, such as hardware or software redundancy [13]. These efforts, in practice, not only add overhead and complexity to the programming environment, but also to the application running environments. Large-scale redundancy is typically not fiscally practical for supercomputers. Thus, the most common solution is checkpointing; that subject is covered in detail in Section 2.3. In order for any scheme to be effective, one must develop useful models of the failure behavior of supercomputers.

Based on data from smaller computer systems, researchers developed a model

of failures as behaving like a Poisson arrival process. That is, failure inter-arrival times are random and independent of one another. Furthermore, these failures were modeled as being independent and identically distributed across nodes in a system. Failures under this model are unpredictable and nodes are treated as being equivalent. Recent studies of large-scale systems, however, have revealed that these assumptions may not be correct [26, 15]. These studies harvested failures from both large-scale clusters of commodity machines, as well as from a BlueGene/L prototype. Failures were found to be

1. *Not independent.* Failures are dependent on system conditions both in the node itself and on physically/logically connected nodes. Particularly when jobs run on numerous nodes simultaneously, the assumption that failures occur independently does not hold. Failures frequently cause, and are caused by, other failures.

2. *Not Identically Distributed.* Failures seem to be temporally and spatially correlated. The probability distribution is not remotely identical across the system. In fact, both the AIX cluster and BG/L prototype showed that the reliability of nodes is distributed exponentially; there are a few troublesome nodes, but most of them are fairly reliable.

3. *Not Poisson.* In the past three years, a number of studies of supercomputer failure distributions [27, 18, 15] have agreed with an earlier study [22] of workstation failure distributions, which contends that failures in real systems do not behave like a Poisson arrival process. Furthermore, equations derived from this Poisson assumption and meant to predict the performance of checkpointing schemes do not accurately model the behavior of these schemes in the presence of real failures.

4. *Not Unpredictable.* Using simple statistical algorithms, it has been shown [27] that failures are not unpredictable; they are often preceded by regular patterns of misbehavior that permit prediction of both when and where many failures

will occur. For failures in a commodity cluster, an algorithm combining linear time-series modeling and Bayesian learning yielded accurate prediction of 70% of the failures well in advance.

2.2.1 Event Prediction

A realistic failure model for large-scale systems should admit the possibility of critical event prediction. Many previous research efforts have looked at analyzing event logs and other system health signals [6, 7], and some have used this to make event predictions [32, 34]. Only more recently have these predictions been used effectively to improve system performance [9, 19]. The idea of using event prediction for proactive system management has also been explored [28, 25], as has a mathematical approach [33].

Critical event prediction is not science fiction. For example, a BG/L prototype revealed itself to have one very predictable type of problem: ECC memory failures. Before ECC memory fails, it will first endure a period during which it is correcting bit errors. This activity can be monitored, and makes it simple to forecast a failure on the node containing that memory. These predictable ECC memory failures were one of the most common reasons for node and job failure during the period in which the system was monitored. It would be invaluable, from a system management perspective, to utilize this predictive power to improve performance and reliability.

Rare events, such as critical failures, can significantly affect system performance. Indeed, job failures could easily make a bigger impact on performance than a particular choice of job scheduling algorithm (for example). Fortunately, many of these failures happen in a predictable manner. Sahoo and Oliner [26] presented a hybrid algorithm for event prediction and demonstrated the effectiveness of this algorithm on actual cluster failure data. The algorithm was able to predict critical failures with up to 70% accuracy on a Linux cluster with 350 nodes, using a combination of Bayesian learning [3, 20, 4] and time-series modeling [5]. This work demonstrated the importance of system health monitoring and event prediction, as well as its feasibility.

2.2.2 Fault-aware Job Scheduling

In light of the significant impact of failures on performance, Oliner demonstrated that event prediction can be used to improve job scheduling [19]. In particular, he presented two new job scheduling algorithms for BlueGene/L systems [1] that built on Elie Krevat's original algorithm [12]. These two scheduling heuristics each used a different model of event prediction. One algorithm, the *balancing algorithm*, expected an event prediction system that took as input a partition and a time window and would return a level of confidence in the partition. For example, a return value of 0.95 would indicate a 95% chance that the given partition would fail in the given time window. The *tie-break algorithm* expected instead a predictor that would return a boolean: *true* if the partition was expected to fail and *false* otherwise. That predictor would be wrong with some probability.

These algorithms were tested on a BlueGene/L simulator, written specifically for that research. This event-driven Java simulator was fed real supercomputer job logs from systems at NASA, SDSC, and LLNL [11]. It was also fed failure data from a large AIX cluster; the data was modified to match with the architecture of the associated job logs. This work demonstrated two important results: event prediction only needs to be about 10% accurate to benefit job scheduling and even low-accuracy event prediction can improve performance by up to a factor of 2. Such significant results motivated research into other areas where event prediction could yield a benefit, namely, checkpointing.

2.3 Checkpointing

Checkpointing for computer systems has been a major area of research over the past few decades. The goal of high performance computing is to obtain maximum efficiency from given resources. System failures (hardware/software, permanent/transient), and the resulting job failures, are a significant cause of degraded performance. Recently, there have been a number of studies on checkpointing based on certain failure characteristics [23], including Poisson distributions. Plank and Elwasif [22] carried out

a study on system performance in the presence of real failure distributions and concluded that it is unlikely that failures in a computer system would follow a Poisson distribution.

The job workloads themselves can be of importance for the performance evaluation of high performance computing systems. The workloads considered by Plank and Elwasif for their study are artificial. Similarly, the communication/network topologies can play an important role in regard to checkpointing or job scheduling strategies for HPC systems. Tantawi and Ruschitzka [31] 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. Such a strategy would be costly in practice, because it is expensive to checkpoint at arbitrary points in a program's execution.

Some applications may work with state already in stable storage. For most high performance applications, however, the state is kept entirely in memory. Indeed, one of the primary reasons for using a supercomputer is not the speed, but the ability to work on a larger problem size. As such, this thesis does not address issues that arise when state on disk is modified, and assumes that all application state is in memory. The checkpointing questions are: what must be saved to stable storage, and when should it be saved?

2.3.1 System-Initiated Checkpointing

System-initiated checkpointing is a part of many large-scale systems, including IBM SPs. This means that 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 [30]. 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. For these reasons, many supercomputing systems, such as BG/L, do not support system-initiated checkpointing. Still,

the ability of the system to force a checkpoint is a powerful tool that can be used to improve QoS [17].

2.3.2 Application-Initiated Checkpointing

Application-initiated checkpointing is the dominant approach for most large-scale parallel systems. Recently, 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 [22], where a study on system performance in presence of real failure distributions concludes that Poisson failure distributions are unrealistic. While that work considers real failure distributions, it uses artificial job logs. Similarly, a recent study by Sahoo et. al. [29, 35], 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. They also report how a job scheduling process can be tuned to take into account the failures on a large-scale cluster by following certain heuristics [35]. Only in the past few months (2004) has there been a study on the impact of realistic large-scale cluster failure distributions on checkpointing [18].

2.3.3 Characteristics

Application-initiated and system-initiated checkpointing each have pros and cons. This section gives a summary of some of these properties, and presents them in Table 2.1.

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

- *Minimal-State Placement.* 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 Optimizations.* 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. Typically, this means picking a periodic checkpointing interval at runtime.

- *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.

Characteristic	**System**	**Application**
Semantics		×
Minimal-State Placement		×
Portable		×
Compiler Optimizations		×
Runtime	×	
Kernel State	×	
Transparent	×	

Table 2.1: Comparison of the characteristics of the two major approaches to checkpointing: application-initiated and system-initiated. Generally, system-initiated provides transparent, coarse-grained checkpointing; application-initiated provides more efficient, fine-grained checkpointing. Ideally, a checkpointing system would possess all of these features.

Certainly, Table 2.1 is neither complete nor strictly precise. For example, systems that are responsible for checkpointing user applications may use some portable intermediate representation, thus providing *Portability*. The entries in the table, however, apply to most checkpointing schemes; the table is a useful reminder of the tradeoffs made by designers attempting to construct reliable systems.

2.4 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 three 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 compiler optimizes these requests by catching errors, removing dead variables, and assisting with optimization techniques such as incremental checkpointing. In the case of cooperative checkpointing, the compiler may move the checkpoint request to a slightly earlier point in time; this permits a number of additional performance improvements.

3. 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, and other factors are considered, this irregularity allows us to do much better. The behavior of applications as they choose to skip different checkpoints is illustrated in Figure 2-3.

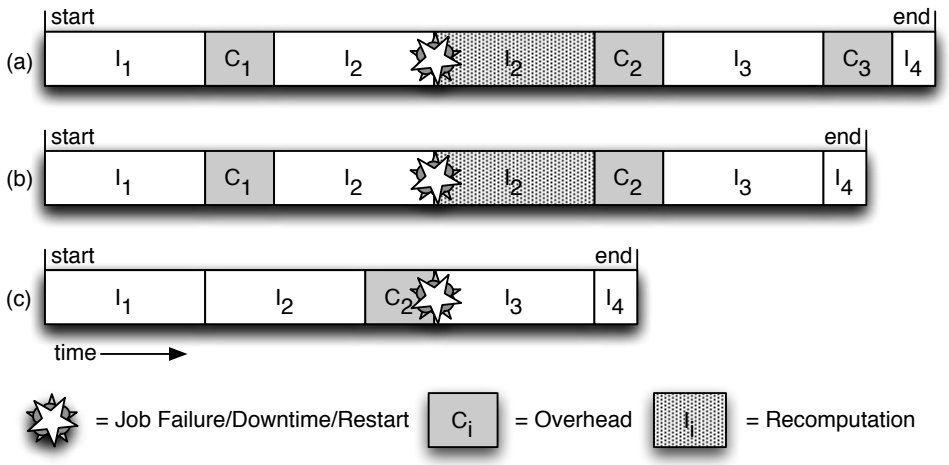

Figure 2-3: Three job runs in which different checkpoints are skipped. Run (a) shows typical periodic behavior, in which every checkpoint is performed. In run (b), the final checkpoint is skipped, perhaps because the critical event predictor sees a low probability that such a checkpoint will be used 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.

2.4.1 Policy

The primary policy question with regard to cooperative checkpointing is, "How does the gatekeeper decide which checkpoints to skip?" Chapter 4 suggests a gatekeeper that uses a combination of network traffic data and critical event predictions to make its decisions. There are, however, many heuristics that the gatekeeper may use, including:

- *Network Traffic.* Network I/O is the central bottleneck with respect to saving state to disk. The gatekeeper may choose to skip a checkpoint if the 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 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.

- *Logically Connected Components.* Recent work [24] has explored the notion of a *connected component* of processes in a job. Roughly, a connected component is a subset of processes among which there is a dependency. Thus, the failure of one connected component may not necessitate the rollback of the entire job, but merely that component.

- *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. 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. This observation is central to cooperative checkpointing.

2.4.2 Characteristics

Recall Table 2.1 from Section 2.3.3, which summarized some of the advantages and disadvantages of application-initiated and system-initiated checkpointing. Table 2.2

includes the addition of cooperative checkpointing, which confers nearly all the benefits of the two standard schemes.

Characteristic	**System**	**Application**	**Cooperative**
Semantics		×	×
Minimal-State Placement		×	×
Portable		×	×
Compiler Optimizations		×	×
Runtime	×		×
Kernel State	×		×
Transparent	×		

Table 2.2: 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.

Checkpoint requests are placed by the application programmer, and can be positioned so as to minimize the size of the checkpointed state. Similarly, the semantics of the data can be captured by the behavior of the checkpoint if the request is granted, and the checkpoints can easily be made portable. Because the potential checkpoint positions are fixed at compile-time, the compiler may make optimizations.

On the other side, the system also participates in the checkpointing process by granting or skipping each checkpoint request. It is empowered to consider runtime properties of the system when making these decisions, and can include kernel state when taking a checkpoint.

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. In the domain of supercomputing, performance trumps simplicity.

Chapter 3

Algorithmic Analysis

Cooperative checkpointing is unlike typical checkpointing schemes in that it decomposes the problem of deciding when to checkpoint into a static part (placing checkpoint requests in the code) and a dynamic part (online boolean decisions regarding which checkpoints to skip). Instead of considering the parameters of the system and deciding when to checkpoint before the program is run, cooperative checkpointing makes online choices about whether or not to perform individual checkpoints based on the information available at the time. The dynamic part can be modeled as an online algorithm, as opposed to an offline optimization algorithm.

This chapter considers the dynamic component of cooperative checkpointing and presents a competitive analysis of various algorithms. Although most competitive analyses use a cost function that represents how expensive operations are, this thesis uses a *value function* that measures the benefit conferred by the algorithm. Specifically, it compares the amount of work saved by a particular checkpointing algorithm rather than the amount of work lost to failures. In this way, the analysis is better able to incorporate the checkpoint overheads.

Among the results in this chapter is a lower bound on the worst-case competitiveness of a deterministic cooperative checkpointing algorithm and several algorithms that meet that bound, a proof that periodic checkpointing using an exponential failure distribution can be arbitrarily bad relative to cooperative checkpointing using an arbitrary failure distribution, 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.

3.1 Worst-Case Competitive Analysis

We model cooperative checkpointing by considering the execution of a program that makes periodic checkpoint requests. The length of this period, I, is a characteristic of the program, not the online algorithm. In other words, I is chosen as part of the static component of cooperative checkpointing, while the decisions of which checkpoints to perform is the dynamic algorithmic component. 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 3-1 illustrates a partial execution including three FF intervals.

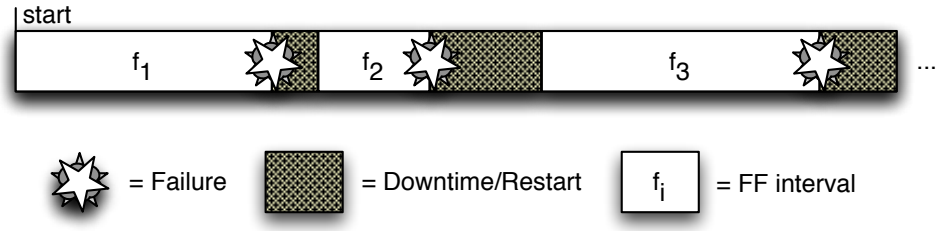

Figure 3-1: The execution of a program with failures, shown up to $n = 3$. The length of the FF intervals (f_i) varies. The downtime and recovery time following a failure is variable, as well, but is not included in the failure sequence Q. The execution may continue beyond what is shown.

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 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, $V_{A,Q} \leq \alpha V_{OPT,Q}$.*

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. Also note that worst-case failure behavior is not the worst sequence for OPT, but the sequence that results in the highest ratio of V_{OPT} to V_A. 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.

3.1.1 Offline Optimal and Cooperative Checkpointing

Recall that the overhead for performing a checkpoint is a constant C for every checkpoint. When a failure occurs, the program is unable to run for some amount of time, which is the sum of the downtime (D) of the nodes on which it is running and the recovery time (R) needed to recover state from the most recent checkpoint. This downtime and recovery time is 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.*

For example, consider the executions of algorithms A and OPT, illustrated in Figure 3-2, that both run for f seconds and then both fail simultaneously. Both A and OPT have exactly f seconds to save as much work as possible; they are competing. Let $I = \frac{f}{8}$ seconds and $C = \frac{f}{32}$ seconds. OPT skips the first 6 checkpoint requests in this FF interval, and performs the 7^{th}. Remember that A does not know f (the length of the FF interval), but OPT does. Let algorithm A take the first checkpoint, skip one, take the third, skip two, and so on. In this example, A takes checkpoints 1, 3, and 6. In this manner, OPT saves $\frac{7f}{8}$ units of work, while A saves $\frac{6f}{8}$ units. We say that, for this FF interval, $V_{OPT} = \frac{7f}{8}$ and $V_A = \frac{6f}{8}$. In Figure 3-2, intervals of saved work are labeled with their length.

Using this execution pair as an example, we now define two kinds of checkpoints.

Definition 3 *A* critical checkpoint *is any checkpoint that is used for recovery at least once.*

Definition 4 *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 are *wasted checkpoints*. Skipping a wasted

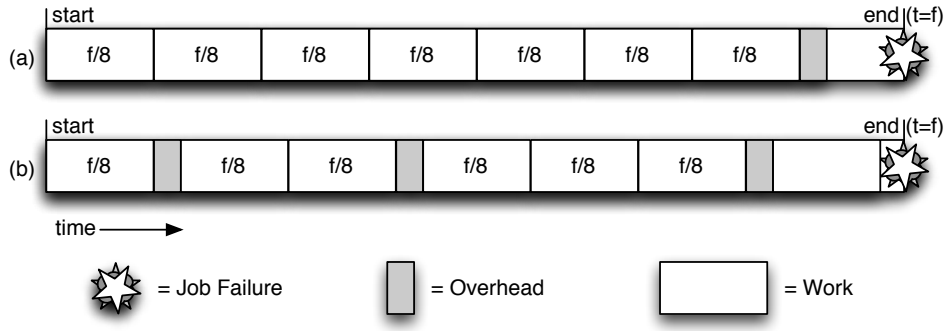

Figure 3-2: An example of the execution of OPT (a) executing alongside a sample algorithm A (b). OPT performs the latest checkpoint that could finish before the failure, and no others. Algorithm A performs the first, third, and sixth checkpoints. A could have performed another checkpoint before failing, but instead saved less work than OPT.

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 Figure 3-2, note that A could have completed the 7^{th} checkpoint before the failure ended the interval. Had A performed that checkpoint, it would have saved $\frac{7f}{8}$ units of work, just like OPT. In order for an algorithm to be made incrementally more like the optimal, two things can be done:

1. Skip wasted checkpoints.

2. Perform a critical checkpoint that is closer to the end of the FF interval.

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. For most executions in which A cannot make progress, neither can OPT.

Often, the competitive ratio of an algorithm is governed by the relationship among various parameters in the system. Of those introduced into the model so far, the two key parameters are I and C. Although R (recovery time) and D (downtime) 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.

Lemma 1 *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, $V_{OPT} \geq V_A + I\lfloor\frac{kC}{I}\rfloor$. (Skipping Lemma)*

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. Consider the standard equation for the optimal periodic checkpointing interval, where $\frac{1}{\lambda}$ is the MTBF:

$$I_{OPT} = \sqrt{\frac{2C}{\lambda}} \tag{3.1}$$

In a real system, such as IBM's BlueGene/L, the projected upper bound for C is 12 minutes (720 seconds). Simultaneously, the mean time between failures for the full machine will likely be rather small, perhaps on the order of minutes. Indeed, if the MTBF is any less than 6 minutes (360 seconds), then $C > I$:

$$I_{OPT} = \sqrt{\frac{2C}{\lambda}} = \sqrt{2 \cdot 720 \cdot 360} = 720 \texttt{ seconds} = C$$

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.

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.*

Proof We must show both directions.

- Given that A is α-competitive, we show that there cannot exist an f such that the amount of work saved by OPT is more than α times the amount of work saved by A; we do this by contradiction. Assume there is such an interval length, f. Construct Q as a sequence of $n > 0$ elements with value f. This is now the worst-case input, because, in every FF interval, OPT saves more than α times the amount of work saved by A. In other words, A is not α-competitive, which is a contradiction.

- Given that, $\forall f$, the amount of work saved by OPT is at most α times the amount of work saved by A, we show that A must be α-competitive. By the given characteristics of the FF interval lengths, there does not exist any f such that OPT saves more than α times as much as A. Furthermore, there exists some f where $V_{OPT} = \alpha V_A$. Within the bounds of the given information, we can construct Q as in the previous bullet using this worst-case f. As before, by Definition 1, A is α-competitive. □

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.*

Remark There may be some f for which the ratio of V_{OPT} to V_A is $\alpha' < \alpha$. The worst-case analysis gives an α that is an upper bound.

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

Proof By Corollary 1, it is only necessary to find the interval length for which OPT does the best relative to A. In this case, let $f = dI + C$ for some $d > 1$. Thus, if A fails just before it completes the first checkpoint; it makes no progress. Meanwhile, OPT will perform the latest of the first $d - 1$ checkpoint requests such that the checkpoint finishes before the failure. Because OPT makes progress, but A does not, the ratio is infinite and 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.*

Proof Consider some deterministic cooperative checkpointing algorithm A and the worst-case FF interval length, f. Let k be the number of wasted checkpoints in that interval, n be the number of computation intervals executed, and m be the number of intervals of work saved by A.

It must be the case that $k \geq 1$, otherwise either A is OPT or f was not the worst-case interval length. Because OPT is not deterministic, and A is, there must be at least one wasted checkpoint in the worst-case interval. Furthermore, $m \geq k$ because every saved interval must have at least one associated checkpoint. Finally, $n \geq m + 1$ because the worst-case FF interval will be extended such that at least one computation interval is performed and then lost.

The competitive ratio (α) is a combination of two factors, alluded to by the steps toward optimality mentioned in Section 3.1.1: the wasted checkpoints and the distance of the critical checkpoint to the end of the FF interval.

$$\alpha = \frac{n}{m} + \lfloor \frac{kC}{I} \rfloor$$

To show that $2 + \lfloor \frac{C}{I} \rfloor$ is a lower bound on deterministic competitiveness, we attempt to make α as small as possible within the constraints determined above. The second term

can be bounded from below by setting $k = 1$. There is exactly one wasted checkpoint. By Theorem 2, A must perform the first checkpoint in order to be competitive. There is one wasted checkpoint, so the program fails before the end of the next checkpoint at time $2I + 2C$. Only the first interval is saved and $m = 1$ as well. Using $n \geq m + 1$, we can lower bound the first term:

$$\frac{n}{m} \geq \frac{m+1}{m} \geq \frac{1+1}{1} \geq 2.$$

Thus, we have a lower bound on α, and no deterministic cooperative checkpointing algorithm can be better than $(2 + \lfloor \frac{C}{I} \rfloor)$-competitive. □

3.1.2 Periodic Checkpointing

This section contains competitive analyses of periodic checkpointing algorithms. Specifically, it describes how cooperative checkpointing can be used to simulate periodic checkpointing, and proves 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 $I_p \bmod I = 0$, then exact simulation is possible. When $I_p \bmod I \neq 0$, 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 $A_{n,1}$. This algorithm is optimally competitive.

Theorem 5 *A_{all} is $(2 + \lfloor \frac{C}{I} \rfloor)$-competitive.*

Proof Set $f = |FFI| = 2I + 2C$. A_{all} takes the first checkpoint, and fails just before finishing the second one; $V_{all} = I$. The behavior of OPT is deduced by working backwards from f. OPT will perform exactly one checkpoint, but after how many requests? First, subtract C from f for the required overhead of OPT's checkpoint. The remaining $2I + C$ can include at least 2 intervals, but the exact number depends on the relationship between C and I. If the overhead is especially large, skipping that checkpoint may allow OPT to execute more intervals of computation before performing a checkpoint. Specifically, $V_{OPT} = 2I + \lfloor \frac{C}{I} \rfloor I$. Therefore, the value of α for this interval is $2 + \lfloor \frac{C}{I} \rfloor$.

Is it possible for some other f_j to give a larger ratio? Consider the growth of V_{OPT} and V_{all} as f_j grows beyond $2I + 2C$. V_{OPT} increases by I for every I that f_j increases. V_{all}, meanwhile, increases by I only after f_j increases by $I + C$. In other words, the optimal pays the checkpoint overhead once for all the work in the interval, while A_{all} must pay the overhead for each checkpoint individually. Asymptotically, as f_j goes to infinity, $\frac{V_{OPT}}{V_{all}}$ goes to $\frac{I+C}{I} = 1 + \frac{C}{I}$. This can never exceed $2 + \lfloor \frac{C}{I} \rfloor$, so the original f was the worst-case interval length.

Therefore, $f = 2I + 2C$ is a worst-case interval length; by Corollary 1, A_{all} is $2 + \lfloor \frac{C}{I} \rfloor$-competitive. □

Corollary 2 *Asymptotically, V_{OPT} grows in proportion to $|FFI|$.*

Remark By Theorem 3, A_{all} is competitively optimal for a deterministic algorithm.

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 Corollary 2.

Let $A_{p,d}$ 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 6 *$A_{p,d}$ 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. □

Remark As expected, the competitive ratio of $A_{p,1}$ is identical to that of A_{all}; they are the same algorithm.

3.1.3 Exponential-Backoff 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 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:

$$\begin{aligned} A_{all} &= \{1,1,1,1,\ldots\} \\ A_{n,2} &= \{0,1,0,1,0,1,\ldots\} \\ A_{p,3} &= \{1,0,0,1,0,0,1,\ldots\} \end{aligned}$$

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:

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

The intention in evaluating 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.

Theorem 7 *A_{2x} is $(2 + \lfloor \frac{C}{I} \rfloor)$-competitive.*

Proof Let $f = |FFI| = 2I + 2C$. For this interval length, A_{2x} has the same behavior as A_{all}, so $\alpha = 2 + \lfloor \frac{C}{I} \rfloor$.

Asymptotically, V_{2x} doubles every time f increases by $V_{2x} + C$. To similarly increase V_{OPT} by V_{2x}, f must be increased by V_{2x}. Thus, the asymptotic ratio is $\frac{V_{2x}+C}{V_{2x}} = 1 + \frac{C}{V_{2x}}$, which goes to 1 for constant C.

$f = 2I + 2C$ was the worst-case interval, so A_{2x} is $(2 + \lfloor \frac{C}{I} \rfloor)$-competitive. $\square$

3.1.4 Comments

If the request overhead is ignored, a smaller I should typically allow the online algorithm to do better, because it has more choices regarding when to perform checkpoints. Instead, this competitive analysis shows that many algorithms have a competitiveness that is inversely related with I. This is a consequence of comparing with OPT; requests that are spaced far apart cause OPT to do more poorly, which makes the cooperative algorithm look better. In practice, the value of C tends to be sensitive to where in the code checkpoints are performed. More frequent checkpoints may result in a smaller C, which improves competitiveness. Furthermore, choosing I such that checkpoint requests can be placed in the code where the required overhead is minimal may prove more valuable than increasing I to artificially improve α.

3.2 Expected Competitive Analysis

This section proposes a more relevant form of competitive analysis in which the algorithm considers the actual failure distribution. The algorithms remain deterministic, but, with this information and the refined model, the analysis is significantly different. This refined model addresses a number of weaknesses with the worst-case analysis, and includes the power to talk about failure distributions.

3.2.1 Failure Probability Density

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,

$$P(a \leq F \leq b) = \int_a^b \chi(t)dt \tag{3.2}$$

and assume that:

$$\begin{aligned} F &\geq 0 \\ \chi(t) &\geq 0 \qquad \forall t \end{aligned}$$

$$\int_0^\infty \chi(t)dt \;=\; 1$$

Properties of this probability distribution, like mean ($\mu = E(F)$), variance (σ), and standard deviation, are calculated in the usual way:

$$\begin{aligned} E(F) &= \int_0^\infty t\chi(t)dt \\ E(F^2) &= \int_0^\infty t^2\chi(t)dt \\ \sigma &= E(F^2) - [E(F)]^2 \\ SD(F) &= \sqrt{\sigma} \end{aligned}$$

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

$$\chi(t) = \delta(t - f) \tag{3.3}$$

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[1]. Let Q be an observationally-collected list of numbers $(f_1, f_2, \ldots, f_n)$, where f_i is the length of the i^{th} FF interval for all jobs across the system (or, perhaps, those from the same node partition). The empirical distribution would then be defined on $[0, \infty]$ by

$$P_n(a, b) = \#i : 1 \leq i \leq n, a < f_i < b/n$$

This can be made into a piecewise continuous function by representing the function as a histogram, where the domain is divided into bins. Rectangles are drawn over the bins such that the height is the proportion of observations per unit length in that bin.

[1]For more information on empirical distributions, and a more complete explanation of this notation, please see Pitman's text *Probability* [21].

This can then be used as $\chi(t)$. By its nature, this function changes over time. For example, replacing a failed node with a new one changes its probability of failure. The ability to adjust the cooperative checkpointing algorithm A to accommodate these changes is powerful.

3.2.2 Work Function

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 $\chi(t)$ in Section 3.2.3 to calculate the expected competitive ratio of the algorithm.

Work functions are nondecreasing, irregularly-spaced staircase functions. Some properties of the work function:

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

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

Proof First, we prove the lower bound. Let s be the number of intervals of saved work ($\frac{W_A(t)}{I}$). Each completed checkpoint must save at least one interval, so $s \geq k$. Thus, $W_A(t) \geq kI$.

The upper bound is determined by considering the total amount of computation time, t, and maximizing the amount of saved work. If k checkpoints were performed, kC time was spent checkpointing. This leaves $t - kC$ time for useful work. Accounting for the possibility of an incomplete interval, the maximum number of intervals of work completed by t is $I\lfloor\frac{t-kC}{I}\rfloor$. □

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:

$$W_{OPT}(t) \geq W_A(t) \qquad \forall t$$

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 3.3 and waits until the latest possible time before performing a single checkpoint.

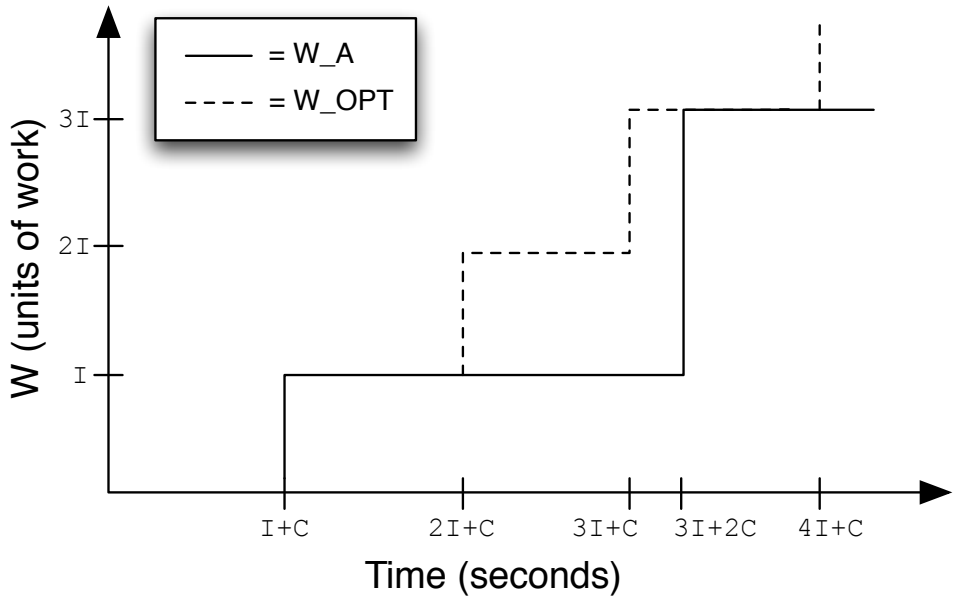

Figure 3-3: The initial portion of two work functions: $W_A(t)$ and $W_{OPT}(t)$. The behavior of A is to skip the second checkpoint. A high probability of failure in the interval $[3I+C, 3I+2C]$ would be bad for A, because the difference between $W_{OPT}(t)$ and $W_A(t)$ is high. The plot is not to scale.

As an example, the work function for A_{all} is

$$W_{all}(t) = I\lfloor\frac{t}{I+C}\rfloor$$

because a checkpoint is completed every $I + C$ seconds, and I seconds of work are saved each time. To help visualize work functions, consider an algorithm A that skips the second checkpoint but performs all others. The work function of A, as well as W_{OPT}, are plotted together in Figure 3-3.

3.2.3 Competitive Ratio

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 5 *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, $E[V_A] \leq \omega E[V_{OPT}]$.*

By the definition of V_A, if a failure happens at time $t = h$,

$$V_A = W_A(h)$$

Therefore, calculating $E[V_A]$ can be thought of as an infinite sum of the product of the work function and the probability of failing at each time over the possible failure times:

$$\begin{aligned} E[V_A] &= \int_0^\infty W_A(t)\chi(t)dt \\ E[V_{OPT}] &= \int_0^\infty I\lfloor\frac{t-C}{I}\rfloor\chi(t)dt \end{aligned}$$

By the definition of ω-competitiveness,

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

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 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 $\chi(t)$.

$W_A(t)$ would be piecewise continuous if there was some way to place an upper bound on t, and thereby make the number of discontinuities finite. Bounding t has value, anyway, because $\int_0^\infty W_A(t)dt = \infty$ for any A that takes at least one checkpoint. There are two reasons why taking this infinite integral is unnecessary:

1. $\lim_{t\to\infty} P(F > t) = 0$ because the system will eventually fail. This means that the length of the FF interval is unlikely to grow very large, and will not be infinite.

2. Programs do not have infinite running time. The end of a program's execution can be modeled as a zero-overhead, required checkpoint. Just as an FF interval does not extend beyond the first failure, the first forced checkpoint effectively ends the FF interval.

Therefore, 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:

$$\omega = \frac{E[V_{OPT}]}{E[V_A]} = \frac{\int_0^T I\lfloor\frac{t-C}{I}\rfloor\chi(t)dt}{\int_0^T W_A(t)\chi(t)dt} \tag{3.4}$$

3.2.4 Example Analyses

This section contains the analysis of several distributions and algorithms in order to illustrate the use of expected competitive analysis for checkpointing.

Uniform Distribution

Let $\chi(t)$ be uniformly distributed from 0 to T. Because the probability density is a constant, it comes out of the integrals and cancels. ω is now a ratio of the area under the curves of OPT and A. The integral of $W_{OPT}(t)$ can be calculated exactly. Use $D = \lfloor \frac{T-C}{I} \rfloor$:

$$I \int_0^T \lfloor \frac{t-C}{I} \rfloor dt = I \sum_{k=1}^{D-1} (kI) + DI(T - DI - C)$$

The only piece needed to calculate ω is $\int_0^T W_A(t)dt$. What valid work function maximizes this? That depends on the relationship between I and C. When $I \gg C$, A_{all} is optimal; Theorem 8 proves that A_{all} is optimal under a uniformly distributed $\chi(t)$ when $I \gg C$. This makes sense, because checkpointing is cheap and we want to maximize the amount of saved work at any given time. Furthermore, recall that W_{OPT} behaved (generally) as though checkpoints were free and took every one of them; a similar strategy is appropriate for A in this case. On the other hand, $C \gg I$ requires a different strategy: skip the early checkpoints and perform the later ones. Because checkpoints are so expensive, the algorithm should wait to perform them until there is work saved up already. Spending time on a long checkpoint when there is little work already saved is risky. This may seem counterintuitive, because skipping checkpoints is typically considered risky. In the case where checkpoints are costly, however, performing them is what holds the most risk.

Theorem 8 *Let $\chi(t)$ be a uniform distribution. If $I \gg C$, A_{all} is 1-competitive (optimal) in the expected case.*

Proof Under the uniform distribution,

$$\omega = \frac{I \int_0^T \lfloor \frac{t-C}{I} \rfloor dt}{\int_0^T W_A(t)dt}$$

because $\chi(t)$ comes out of the integrals as a constant and cancels (as will the factors of I in the work functions). Take $W_{all} = I\lfloor \frac{t}{I+C} \rfloor$ and consider the limit as $C \to 0$:

$$\lim_{C \to 0} (\omega) = \lim_{C \to 0} \left(\frac{\int_0^T \lfloor \frac{t-C}{I} \rfloor dt}{\int_0^T \lfloor \frac{t}{I+C} \rfloor dt} \right) = 1$$

Therefore, for $I \gg C$, A_{all} is expected case 1-competitive (optimal). □

Of course, practical situations will probably be somewhere in between, in which case more careful selection is required. For small T, the space of algorithms is searchable. The space grows exponentially ($O(2^T)$), but the search can be guided by heuristics inspired by the observations for the extreme cases above. Specifically, skipping checkpoints (especially early on) is generally good when C is large, and performing many of them is good when I is large. Recall that I is not the checkpointing interval, but the request interval. Having $I < C$ or even $I \ll C$ is not as bizarre as in the situation where checkpoints are actually taken every I seconds.

If the uniform distribution is constrained to some interval $[a, b]$ smaller than $[0, T]$, with $0 \leq a < b \leq T$, such that

$$\chi(t) = \begin{cases} \frac{1}{b-a}, & \text{if } t \in [a, b] \\ 0, & \text{otherwise} \end{cases}$$

then the strategy is similar. Inside $[a, b]$, the analysis proceeds as before. After that interval, performing checkpoints has no value. Before the interval, exactly one checkpoint should be performed: the latest checkpoint that would complete before a. As $a \to b$, this analysis and the resulting algorithm become increasingly like OPT from the worst-case analysis. That is, if A knows the exact time of the failure, it now has all the power that OPT had. Realistically, this will rarely be the case. Still, one can imagine the scheduled shutdown of an otherwise reliable system might approximate to this situation.

Exponential Distribution

The reliability of components is often modeled as following an exponential distribution. Because it has nice properties, we use

$$\chi(t) = \lambda e^{-\lambda t} \qquad t \geq 0$$

If we think about F as being the lifetime of an FF interval, then the probability of the interval lasting longer than s seconds is

$$P(F > s) = \int_s^\infty \chi(t)dt = e^{-\lambda t}$$

Furthermore this *survival function* can be used to calculate the probability of F falling within a particular range, which, for an exponential $\chi(t)$, is:

$$P(a < F \leq b) = P(F > a) - P(F > b) = e^{-\lambda a} - e^{-\lambda b}$$

Among the nice properties of this distribution are the mean and standard deviation:

$$E(F) = SD(F) = \frac{1}{\lambda}$$

The distribution of a positive random variable is memoryless iff it is an exponential of the form $e^{-\lambda t}$ for some $\lambda > 0$. Memoryless means that the probability that $F > a+b$ given that $F > a$ is the same as the probability that $F > b$ in the first place. Formally,

$$P(F > a + b | F > a) = P(F > b) \qquad (a \geq 0, \quad b \geq 0)$$

A great deal of research in systems reliability treats the lifetime of components as being exponentially distributed. That is, there is no gradual degradation or aging of components; they fail suddenly and without warning. Most distributions (any that are not exponential) are *not* memoryless. 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, as above, 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 9 *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: $I_{OPT} = \sqrt{\frac{2C}{\lambda}}$. All that remains is to set it up so that A_λ will *not* make progress, even though it is the optimal periodic checkpointing algorithm for an exponential distribution with that mean.

The way to do this is to 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:

$$a(I + C) + (1 - a)t_2 \geq \frac{2I^2}{C} \tag{3.5}$$

Using Equation 3.5 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 3 *By Theorem 9, a periodic checkpointing algorithm that assumes an exponential failure distribution may be arbitrarily bad compared to a cooperative checkpointing algorithm that permits arbitrary probability density functions as failure distributions.*

In Section 3.1.1 there was some discussion about real reliability numbers in the context of IBM's BlueGene/L. In light of Theorem 9, it is worth asking how badly BG/L 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?

First, set $C = 6$ minutes (360 seconds); the upper bound 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 selecting realistic numbers for the checkpointing interval and distribution mean, an interesting paradox occurs. Consider these two approaches to setting the variables in the absence of empirical data, and how and why they fail:

- Assume the mean time to failure for an individual node is 5 years. This may even be too high. With $N = 65,536$ nodes, the mean time to failure for them collectively comes from the survival function:

$$P(F_1 > t \ \& \ F_2 > t \ \& \ldots \& \ F_N > t) = (P(F > t))^N = e^{-N\lambda t} = e^{-65536\lambda t}$$

The expected failure time, then, is decreased by a factor of 65,536. That is, $E[F] = \frac{2^{-16}}{\lambda}$. If $\frac{1}{\lambda}$ was 5 years, by assumption, then the expected lifetime of the entire machine is around 40 minutes (2406 seconds). The optimal periodic checkpointing interval for this distribution is 21.9 minutes (1316 seconds). With a checkpoint overhead of 6 minutes, this means spending 21% of the machine time on performing checkpoints. In the worst case (C = 12 minutes), this percentage is 28%. No user or system administrator would be willing to spend this much time checkpointing, but anything less would increase the application's expected running time.

- Instead, assume that no more than 10% of the machine time should be spent performing checkpoints. If the overhead is 6 minutes (360 seconds), this means checkpointing every 54 minutes (3240 seconds). For what exponential distribution is this actually optimal?

$$\begin{aligned} 3600 &= \sqrt{2(360)E[F]} \\ E[F] &= \frac{3240^2}{2(360)} \\ &= 14580 \texttt{ seconds} \\ &= 243 \texttt{ minutes} \end{aligned}$$

This requires that each individual component have a lifetime 65,536 times greater than that; the mean time to failure for a single node must be 30.3 years. This is not realistic, but anything less would require more frequent checkpoints.

In practice, the mean time between failures for a 4,096-node BG/L prototype was 12.45 minutes (747 seconds). Many of these failures were correlated, however, and shared a root cause. In the end, Sahoo et al [15] estimate that the system saw 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. The reason application programmers have an expectation of progress in these situations is that they have implicitly accepted that failure dis-

tributions are not exponentials, are not identically distributed, or not independent. Cooperative checkpointing makes these implicit assumptions into explicit considerations and reaps a benefit as a result.

Returning to the original question, how would a periodic checkpointing algorithm perform if it really did assume an IID exponential distribution? On the full BG/L machine, it would checkpoint every 17 minutes (1,025 seconds). With an overhead of 6 minutes, it would spend 26% of the machine time performing checkpoints; this may be as high as 33% when $C = 12$ minutes.

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 [15], 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 Theorem 9; 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 1459 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, $\chi(t)$ causes the application to fail at time $I + C$ 99.88% of the time, a situation in which A_λ saves no work, but OPT saves I. The remaining 0.22% of the time,

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

The expected ratio for this example case is,

$$\omega = \frac{V_{OPT}}{V_\lambda} = \frac{0.9988(512) + 0.0022(503296)}{0.0022(186368)} = \frac{1618.6368}{410.0096} = 3.95$$

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. The realism of $\chi(t)$ or the ability to actually achieve OPT are beside the point. The intention of this example was to illustrate that cooperative checkpointing is not merely of theoretical value; it can result in tangible improvements in reliability and performance over periodic checkpointing.

3.3 Dynamic Competitive Analysis

This section introduces dynamic-case analysis. This model differs from the expected-case model in the following ways:

1. The gatekeeper selects a deterministic cooperative checkpointing algorithm at each checkpoint request based on what is optimal at the time; the selection only affects the current checkpoint request, because the algorithm will be re-considered at the next request. This is different from the previous models, in which the algorithm was selected at the beginning of the FF interval and was not reconsidered until after a failure.

2. $\chi(t)$ can change between checkpoint requests, not just between FF intervals. As $\chi(t)$ evolves over the course of an execution, the gatekeeper may dynamically change the checkpointing algorithm, as per change #1.

3. The checkpoint overhead is no longer constant, but may be different for every checkpoint. The overhead of the i^{th} checkpoint in an FF interval is denoted C_i.

4. The model introduces another parameter in addition to expected value for the application: the *system value function*, S. This allows the gatekeeper to consider such factors as the scheduling queue status, deadlines and QoS guarantees, and network traffic considerations.

The checkpointing choices of a job can affect the system by tying up network resources, restricting the job scheduler, or impacting system-level performance guarantees. The system value function, S, captures these considerations and is a measure of how desirable a particular checkpointing algorithm is to the system, as opposed to the application. S is the analog of V, so it also has units of `work`. The system value of an algorithm A is denoted by S_A.

Definition 6 *The* cumulative value *of an algorithm A, denoted Z_A, is the sum of V_A and S_A.*

The equation for S will differ depending on the resource restrictions and capabilities of the system. For an example of a system value function, consider a system that checkpoints over a shared, packet-based pipe. A large application P_1 starts a checkpoint with a projected overhead of C seconds; simultaneously, P_2 requests a checkpoint, also with a projected overhead of C seconds. The system knows that P_1 is checkpointing, and that starting a second checkpoint would effectively double the overhead for both checkpoints. The extra C of overhead for P_2 is captured in V. The system value function must capture the notion that choosing to perform the current checkpoint also *costs* P_1 an extra C of work. Let A be any algorithm that would take the checkpoint:

$$S_A = -C$$

This leads into the notion of projecting the checkpoint overhead, which is no longer set to be a constant. There are many potential techniques for doing this estimation, including a network traffic analysis (as used with P_1 and P_2), software techniques for tracking the amount of application state that has changed, and application profiling.

Definition 7 *An online checkpointing algorithm A has* dynamic competitive ratio ϖ *(A is ϖ-competitive) if the expected cumulative value of the optimal offline algorithm (OPT) is at most ϖ times the expected cumulative value of A. That is, $E[Z_A] \leq \varpi E[Z_{OPT}]$.*

The gatekeeper for the dynamic case selects the optimal checkpointing algorithm at each checkpoint request. It is not required to execute the remainder of the algorithm, but it looks ahead in order to estimate risk. Returning to the binary notation for deterministic cooperative checkpointing algorithms, the gatekeeper is, at each checkpoint request, asking what algorithm with a prefix matching what has already been executed is optimal for the latest system data. Imagine that $\chi(t)$ is generated by an event predictor that only generates predicted distributions of the form $\chi(t) = \delta(t - t_1)$ where t_1 changes over the course of the FF interval. Let A_i be the algorithm that the gatekeeper "picks" at request i. One possible beginning to an FF interval might be as follows, where the current checkpoint is underlined:

$$\begin{aligned}
A_1 &= \{\underline{0}, 0, 0, 1, 0, 0, 0, 0, \ldots\} \\
A_2 &= \{0, \underline{1}, 0, 0, 0, 0, 0, 0, \ldots\} \\
A_3 &= \{0, 1, \underline{1}, 0, 0, 0, 0, 0 \ldots\} \\
A_4 &= \{0, 1, 1, \underline{0}, 1, 0, 0, 0, \ldots\} \\
A_5 &= \{0, 1, 1, 0, \underline{1}, 0, 0, 0, \ldots\} \\
A_6 &= \ldots
\end{aligned}$$

The execution might be interpreted as follows. At the first request, $\chi(t)$ predicts that the failure will not occur until after the 4^{th} checkpoint request can be performed successfully. $A_1 = OPT$ given this information. At the second request, however, $\chi(t)$ has been updated to indicate that the failure will happen before the 3^{rd} checkpoint can be completed, so A_2 is optimal. The failure does not occur, and at the third request the predictor has revised $\chi(t)$ to suggest that the failure will happen before the 4^{th} checkpoint request can be performed successfully. A_3 is not optimal, because it would have been better to skip the second request. Given the choices the gatekeeper already made, A_3 is the best it can do. This continues until a failure actually occurs, thereby ending the FF interval.

3.3.1 Example Analysis

Consider a system in which the job scheduler must adhere to deadlines set by users. From the system's perspective, jobs that are not completed by the deadline have no value. The following analysis shows how cooperative checkpointing can help to ensure that deadlines are met, and demonstrates how dynamic-case analysis works. Oliner et al [17] investigated a similar system in which probabilistic deadlines were promised, and suggested that a mechanism similar to cooperative checkpointing might be used to make and keep those promises.

If a job misses its deadline, it is not useful and counts toward wasted work. Let d_A be an indicator random variable such that $d_A = 1$ iff algorithm A is guaranteed to cause the application to miss its deadline. An appropriate S_A (and Z_A) in this case might be

$$\begin{aligned} S_A &= -d_A V_A \\ Z_A &= V_A - d_A V_A \end{aligned}$$

The cumulative value function is zero if A would cause the job miss its deadline. Note that S may be negative, unlike V.

Let $t = 0$ be the beginning of the FF interval, and $t = t_1$ be the time of the 5^{th} checkpoint request. This example application has running time $9I$ and makes only 8 checkpoint requests. Without explanation as to why certain checkpoints were taken or not, say the execution so far restricts the gatekeeper's choices at time t_1 to algorithms in the family

$$A_? = \{0, 1, 1, 0, \underline{?}, ?, ?, ?\}$$

Let the projected checkpoint overhead for all future checkpoints be C and let the application deadline be $t_1 + 5I + C$. A can only afford to perform one more checkpoint and to recompute at most I work. The gatekeeper must pick A to maximize the probability that a failure (if one occurs) will happen less than I seconds after the

completion of the next checkpoint. Let

$$\chi(t) = \frac{1}{2}\delta(t - (t_1 + I + C)) + \frac{1}{2}\delta(t - (t_1 + 2I + C))$$

This means that the predictor thinks it is equally likely that the gatekeeper should either perform the current checkpoint (5^{th}) or perform the next checkpoint (6^{th}). What is the dynamic competitive ratio in this situation?

There are only two algorithms that result in a non-zero Z:

$$A_x = \{0, 1, 1, 0, 0, 1, 0, 0\}$$
$$A_y = \{0, 1, 1, 0, 1, 0, 0, 0\}$$

Calculate the competitive ratio for each:

$$\frac{Z_{OPT}}{Z_{A_x}} = \frac{E[V_{OPT} + S_{OPT}]}{E[V_{A_x} + S_{A_x}]} = \frac{9I + 0}{\frac{9I}{2} + \frac{9I}{2} - \frac{9I}{2}} = \frac{9I}{\frac{9I}{2}} = 2$$

A_x is 2-competitive in the dynamic case; this is intuitive because with probability 0.5 A_x misses the deadline and with probability 0.5 A_x is OPT. The situation is symmetrical for A_y, meaning that the gatekeeper could pick either one without preference or prejudice.

As is the case with deadline-based systems, the dynamic case analysis considers factors, when checkpointing, that are critical to the system as a whole. The following chapter considers what kind of infrastructure might be needed to support cooperative checkpointing with the ability to consider the dynamic case.

Chapter 4

Embodiment

This chapter discusses a possible embodiment for cooperative checkpointing, so that it might be implemented in practice. An explanation of the general system model is followed by a description of what infrastructure might be used to support cooperative checkpointing.

There are two implementations in progress. One is in the form of a Java simulator, and the other is being coded for IBM's BlueGene/L. For BG/L, cooperative checkpointing will initially be used to dynamically adjust the checkpoint interval based on the size of the job's partition. The intention is for BlueGene to eventually include an implementation of cooperative checkpointing of the kind described in this chapter, complete with health monitoring and event prediction. The simulator implemented an earlier rendition of cooperative checkpointing. That simulator and the results of a number of experiments are presented in Section 4.3.

4.1 System Description

The system is modeled as N computational nodes of equal processing power, connected to stable storage via a shared network connection with limited bandwidth. The cluster is composed of homogeneous, dedicated nodes, where homogeneous means only that the hardware specifications are uniform but not identical across the cluster. Unlike most previous work that assumed identical processors, we allow nodes to vary

in reliability and fail in correlation with the behavior of other nodes. In that respect, no two nodes are identical. The assumption of homogeneity simply means that any two non-failing nodes can accomplish the same amount of useful work in a certain amount of time.

In general, the connection to stable storage (I/O) is the bottleneck with regard to saving a checkpoint to disk. Let this connection, or *pipe*, have bandwidth B Gb/s. While the bandwidth of the disk itself may be the true bottleneck, the pipe is modeled as being the primary determinant of checkpoint overhead time. A job j that must save K Gb to disk as part of a particular checkpoint requires $\frac{K}{B}$ seconds of dedicated time on the pipe. When $n > 1$ jobs are competing for the pipe, the required overhead increases as a function of n. This model allows the system to estimate the checkpoint overhead before agreeing to perform the checkpoint, while still simulating the effect of the I/O bottleneck on multiple checkpoints. The system is illustrated in Figure 4-1.

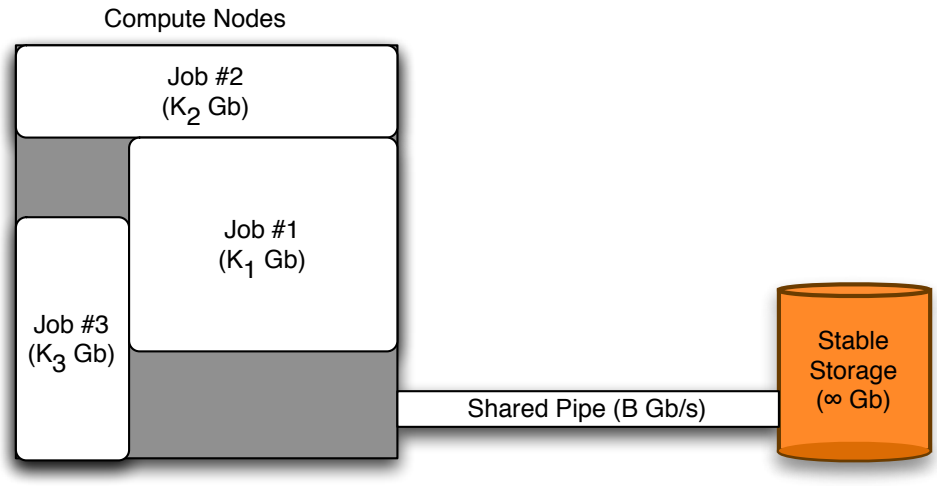

Figure 4-1: Overview of the system and the relationship among its parts. The cluster is running three jobs with state sizes K_1, K_2, and K_3. During a checkpoint, these jobs must all save their state over the Shared Pipe.

4.2 Infrastructure

In order to perform cooperative checkpointing, the system must be equipped with a gatekeeper and the tools necessary for the gatekeeper to make informed decisions. Such tools may include a Reliability, Availability, and Serviceability (*RAS*) Database for monitoring, and an event prediction mechanism (*predictor*) for modeling and forecasting. An overview of a system that is equipped to perform cooperative checkpointing is illustrated in Figure 4-2.

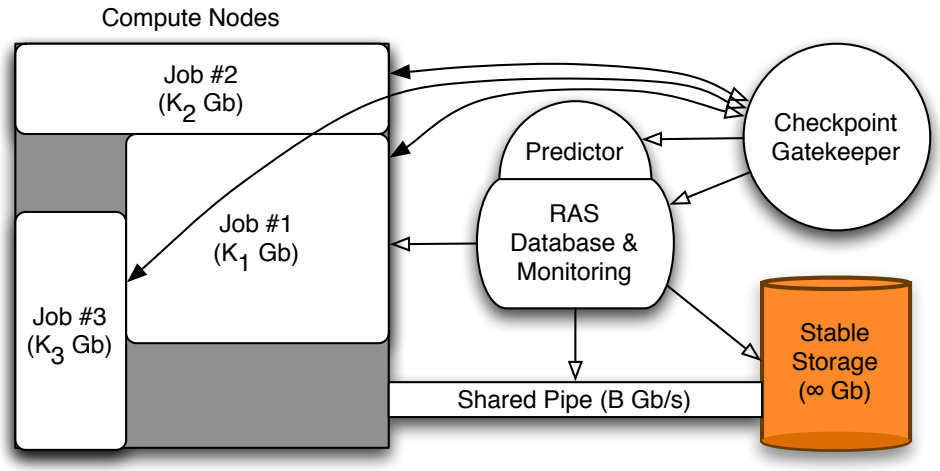

Figure 4-2: The system, including the tools used to perform cooperative checkpointing. The RAS Database monitors the state of the system, logging various events and messages that are pertinent to system health. The Predictor, in turn, uses this data and internal models to forecast critical events. The Checkpointing Gatekeeper receives checkpoint requests from the jobs, and uses RAS and Predictor information to make a decision regarding whether or not to skip the checkpoint.

4.2.1 System Health Monitor and Modeler

In order to make informed checkpointing decisions, the system must have a way of monitoring and modeling its health. This mechanism has access to both physical and logical data about the state of the machine, including information such as node temperatures, power consumption, error messages, problem flags, and maintenance schedules. The more information the health monitor is able to access, the more use-

ful the derived models can be. Most large-scale machines, like BG/L, are already equipped with a RAS database that logs everything from failed communication messages to ECC memory errors.

This modeling mechanism is able to answer questions like, "Is there a problem with node x? Which nodes are frequently exceeding their maximum tolerated temperature? What communication errors have been passed between nodes y and z?" Such a system could be used without event prediction to improve other system features, such as job scheduling. This thesis will mostly use the health monitoring and modeling system as a tool for checkpointing.

4.2.2 Event Predictor

The supercomputer should possess an event prediction mechanism, which makes heavy use of the health modeling system. This event prediction may use a set of algorithms similar to those presented elsewhere [26], or any other algorithm that provides reasonable prediction accuracy. The prediction system [19] is given a set (partition) of nodes and a time window and either:

1. Returns the estimated probability of failure or

2. Returns a boolean indicating whether a failure is anticipated or not.

The quality of the predictor is governed by the accuracy with which it estimates these probabilities (Type #1) or correctly identifies whether a failure was going to occur (Type #2). One way to think about the predictor is as a tool for estimating the failure probability distribution, $\chi(t)$.

What events, exactly, are we interested in predicting? For a job running on a given partition, we are looking for any critical events in the system that would cause that job to fail. For example, the predictor could be used to predict when a troublesome memory component will fail completely, or will result in data corruption so severe that the application will have to be rolled back.

How far ahead must we predict? The choice of whether or not to skip a checkpoint becomes moot once the subsequent checkpoint is completed. Thus, we typically want

to predict critical failures that will happen prior to the end of the next checkpoint. Intervals exceeding several hours are uncommon, meaning we will rarely need to predict further ahead than that.

4.2.3 Gatekeeper

The gatekeeper is the online decision-making component. As illustrated in Figure 4-2, it is in communication with nearly all other system components. When a checkpoint request is made by an application, the request is passed to the gatekeeper. This component has been actively considering everything it knows about $\chi(t)$, network traffic, the estimated checkpoint overhead, and so on; when the request comes in, it is ready with an answer. This minimizes the request/response latency of cooperative checkpointing. In a large-scale system of the kind targeted by this thesis, I is large enough, and the number of running jobs is small enough, that having an answer ready for each job at any given time is not computationally expensive.

Section 3 analyzed online cooperative checkpointing algorithms. These algorithms are chosen and executed by the gatekeeper. To put it another way, the dynamic component of cooperative checkpointing resides in the gatekeeper, while the static component is implemented inside the applications.

As an example, imagine that the predictor, using data gathered from the RAS database, estimates that $\chi(t) = \lambda e^{-\lambda t}$ for some λ. With only this information, the gatekeeper will choose to perform checkpoints roughly periodically. At some point, however, the predictor may notice that the pipe is being saturated by traffic from some other application. It predicts that the checkpoint overhead will be very high and shares this information with the gatekeeper. The gatekeeper may then increase the checkpointing period in order to perform fewer checkpoints (until this network traffic subsides).

4.3 Simulations

This section presents results from simulations of a rudimentary implementation of cooperative checkpointing. They demonstrate that using cooperative checkpointing, even with very simple heuristics, can translate to measurable performance improvements. An event-driven simulator is used to process actual supercomputer job logs, and failure data from a large-scale cluster [29].

The simulations consider gatekeepers that use one of two heuristics, called *work-based* and *risk-based* checkpointing. Let d be the number of intervals of computation that have been performed since the last checkpoint, C_i be the projected overhead for the current checkpoint, p_f be the predicted probability that a failure will occur before the completion of the subsequent checkpoint, and I be the request interval. The heuristic for risk-based checkpointing is

$$p_f dI \geq C_i \tag{4.1}$$

At each checkpoint request, the gatekeeper checks Equation 4.1. If the inequality holds, the gatekeeper performs the checkpoint. Work-based checkpointing is the special case of risk-based checkpointing where $p_f = 1$. This simplified conception of cooperative checkpointing is run in simulation.

4.3.1 Simulation Environment

The event-driven simulator models a 128-node system either in flat cluster or in three dimensional $4 \times 4 \times 8$ torus configuration. The simulator is provided with a job log, a failure log, and other parameters (for example: checkpoint overhead, checkpoint interval). The events include: (1) *arrival events*, (2) *start events*, and (3) *finish events*, similar to other job scheduling simulators [12]. Additionally, the simulator supports (4) *failure events*, which occur when a node fails, (5) *recovery events*, which correspond to a failed node becoming available again, (6) *checkpoint start events*, indicating the start of a job checkpoint, and (7) *checkpoint finish events*,

which correspond to the completion of a checkpoint.

Compared to earlier work [19], the following changes were made to the simulation environment.

- Jobs may be checkpointed, and these checkpoints have an overhead. The interval and overhead cost are parameters of the simulation.

- The downtime of a failed node is set at a constant 120 seconds, which is estimated to be a modest restart time for nodes in any large-scale computer system. While down, no jobs may be run on the node.

- The job scheduler is equivalent to the scheduler in previous work [12] with only backfilling.

The simulation produces values for the last start time (s_j) and finish time (u_j) of each job, which are used to calculate wait time (w_j), response time (r_j), and bounded slowdown (bs_j). While utilization measurements can often be misleading, we still calculated system capacity *utilized* and *work lost* based on the following formulations. If $M = (\max_{\forall j}(u_j) - \min_{\forall j}(a_j))$ denotes the time span of the simulation, then the capacity utilized (K_{util}) is the ratio of work accomplished to computational power available.

$$K_{\text{util}} = \sum_{\forall j} \frac{s_j e_j}{MN}.$$

Let t_x be the time of failure x, and jx be the job that fails as a result of x, which may be *null*. If c_{jx} is the time at which the last successful checkpoint for jx started, then the amount of work lost as a result of failure x is $(t_x - c_{jx})n_{jx}$ (this equals 0 for $jx = null$). Hence, the total work lost (K_{lost}) is

$$K_{\text{lost}} = \sum_{\forall x} (t_x - c_{jx})n_{jx}.$$

This thesis ignores K_{unused}, which relates to capacity that is unused because of a lack of jobs requesting nodes or other non-failure reasons.

This thesis also considers metrics similar to those in Krevat's scheduler [12]. The actual job execution time is calculated based on start time s_j and actual finish time u_j of each job; when measuring utilization, we use execution time excluding checkpoints. Similarly, s_j, u_j, and job arrival time (a_j) can be used to calculate wait time $w_j = s_j - a_j$, response time $r_j = u_j - a_j$, and bounded slowdown $bs_j = \frac{max(r_j,\Gamma)}{min(e_j,\Gamma)}$, where $\Gamma = 10$ seconds. Therefore, we consider the following metrics when evaluating overall system performance: (1) $\{Average[w_j]\}$, (2) $\{Average[r_j]\}$ and (3) $\{Average[bs_j]\}$.

Calculations used the so-called "last start time" of each job, which is the latest time at which the job was started in the cluster. There may be many start times, because a failed job returns to the wait queue. It would be misleading to use the first start time, because a job may fail many times, spend time checkpointing in the cluster, and sit waiting in the queue, all after the initial start time. Due to the choice of start time, w_j tends to be similar to r_j.

In order to be consistent, and because this thesis proposes that checkpoints should be optional, checkpointing overhead is treated as being unnecessary work. That is, e_j is the execution time of the job *without* checkpoints. Therefore, values for bounded slowdown, for example, may seem unusually high. In fact, this is a more accurate representation of the performance of the cluster; if the checkpoints could be skipped, the baseline optimal may be improved.

4.3.2 Workload and Failure Traces

The simulations used job logs from the parallel workload archive [11] to induce the workload on the system. The parallel job logs include a log from NASA Ames's 128-node iPSC/860 machine collected in 1993 (referred to as NASA log henceforth), San Diego Supercomputer Center's 128-node IBM RS/6000 SP job log from 1998-2000 (SDSC log), and Lawrence Livermore National Laboratory's 256 node Cray T3D job log from 1996 (LLNL log). For space considerations, this thesis focuses on the results using the SDSC log, and includes some results using the NASA logs. Each log contained $10,000$ jobs. Some characteristics are shown in Table 4.1, where runtimes do not include checkpoints.

Job Log	Avg Size	Avg Runtime (s)	Max Runtime (hr)
NASA	6.3	381	12
SDSC	9.7	7722	132
LLNL	10.2	1024	41

Table 4.1: Basic characteristics of the job logs used to generate workloads in the simulations.

To generate failure behavior, failure logs were used from filtered traces collected for a year from a set of 350 AIX machines for a previous study on failure analysis and event prediction [29, 27]. Failures from the first 128 such machines were used, resulting in $1,021$ failures, an average of 2.8 failures per day. The MTBF on any node in the cluster was 8.5 hours. Therefore, the timing and distribution of failures used in these experiments reflect the behavior of actual hardware and software in a large cluster.

4.3.3 Results

Results are presented for the NASA and SDSC job logs on a flat cluster, as well as for a toroidal communication architecture using the SDSC log. The simulations, in all, represent more than 600,000 days of cluster time, and involve the scheduling of more than 30 million jobs. The results presented here are necessarily a subset of these simulations. A particular graph was included either because it was representative of our results or it accentuated an important feature. Exceptional results are noted as such.

System Performance

This section investigates the effects of work-based and risk-based checkpointing on system-level metrics such as average bounded slowdown. Figure 4-3 plots checkpointing interval against average bounded slowdown for the SDSC log, on a flat cluster, with a checkpoint overhead of 12 minutes (720 seconds). The same runs for the NASA log are shown in Figure 4-4, and runs for the SDSC log on a toroidal interconnect architecture are shown in Figure 4-5. The five curves represent periodic, work-based,

and risk-based checkpointing for three accuracy levels. Periodic checkpointing means every checkpoint is performed at intervals defined by the x-axis value. Similarly, work-based checkpointing is performed according to the definition in Equation 4.1 with $p_f = 1$.

Risk a indicates risk-based checkpointing with a false negative rate of $1 - a$. Referring to the definition of risk-based checkpointing in Equation 4.1, $a = 0$ implies that the predictor will always return $p_f = 0$ or no checkpointing is performed. Despite the fact that no checkpoints are performed, the metric for Risk 0 varies with the checkpoint interval. This is because the job scheduler must estimate the completion time of the job. In other words, the scheduler considers the runtime (r), checkpoint interval (I), and checkpoint overhead C, and estimates the total running time (R) as if all checkpoints are to be performed based on $R = r + C \cdot \lfloor (r/I) \rfloor$. Therefore, R for each job is estimated to be greater for a smaller I, making performance enhancing techniques, like backfilling, less likely. As a result, performance of Risk 0 is marginally worse at smaller I values.

For a checkpoint overhead of 720 seconds, $I < C$, so work-based checkpointing results in the same curve as periodic checkpointing. As the checkpointing interval is decreased, bounded slowdown for the periodic checkpointing scheme increases exponentially. In general, risk-based checkpointing, at any accuracy, results in a lower bounded slowdown compared to either work-based or periodic checkpointing. This is because the bounded slowdown is dominated by the checkpointing overhead. Risk-based checkpointing will never perform more checkpoints than work-based checkpointing, and work-based checkpointing will never perform more checkpoints than periodic checkpointing. Zooms of Figures 4-3 to 4-5, with all accuracy levels included, are shown in Figures 4-6 to 4-8, respectively.

As a representative case of checkpointing results for higher overheads (say $C =$ 3600 seconds), Figure 4-9 plots bounded slowdown for the SDSC log on a flat cluster. Between the intervals of $I = 3500$ seconds and $I = 4000$ seconds, work-based checkpointing diverges suddenly and dramatically from periodic checkpointing. The checkpoint overhead is 3600 seconds, so $I > 3600$ seconds means that every check-

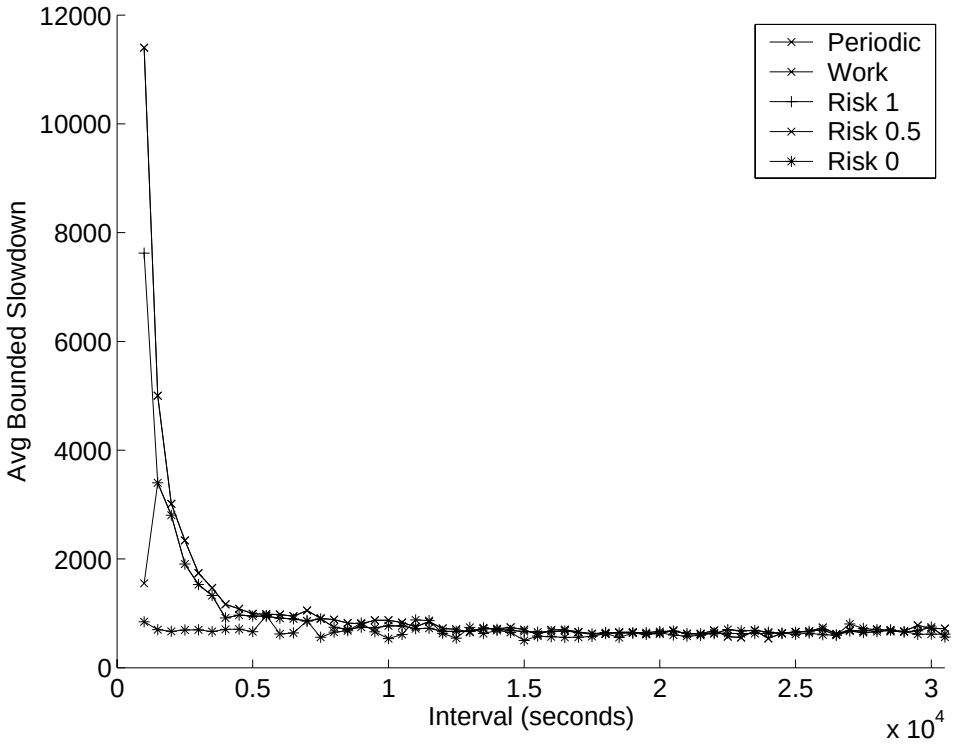

Figure 4-3: Bounded slowdown vs. checkpoint interval in seconds for the SDSC job log, using a checkpoint overhead of 720 seconds.

point will be performed. Below 3600 seconds, the work-based heuristic takes into effect. At $I = 3500$ seconds, for example, every other checkpoint is performed, starting with the second one. Applications that only checkpointed once do not checkpoint at all. This immediately results in a 7-fold decrease in average bounded slowdown. Again from Figure 4-9, there is nearly a 50% gap in performance between the *Work* and *Risk 1* maximum values. A similar gap can be seen in all Figures for $C = 720$ seconds. Work-based checkpointing will perform every checkpoint such that $dI > C$, whether or not the event predictor indicates that a failure is likely. On the other hand, risk-based checkpointing, with no false positives, will only perform a checkpoint when a failure is predicted to occur before the end of the subsequent checkpoint. Consequently, there are many checkpoints to be performed by work-based checkpointing compared to risk-based checkpointing. In the case of $a = 1$, all failures are correctly predicted (ideal case). Our predictor does not have any false positives. If, however, the false positive rate was set at 1 (always predicts a failure) and the predictor, there-

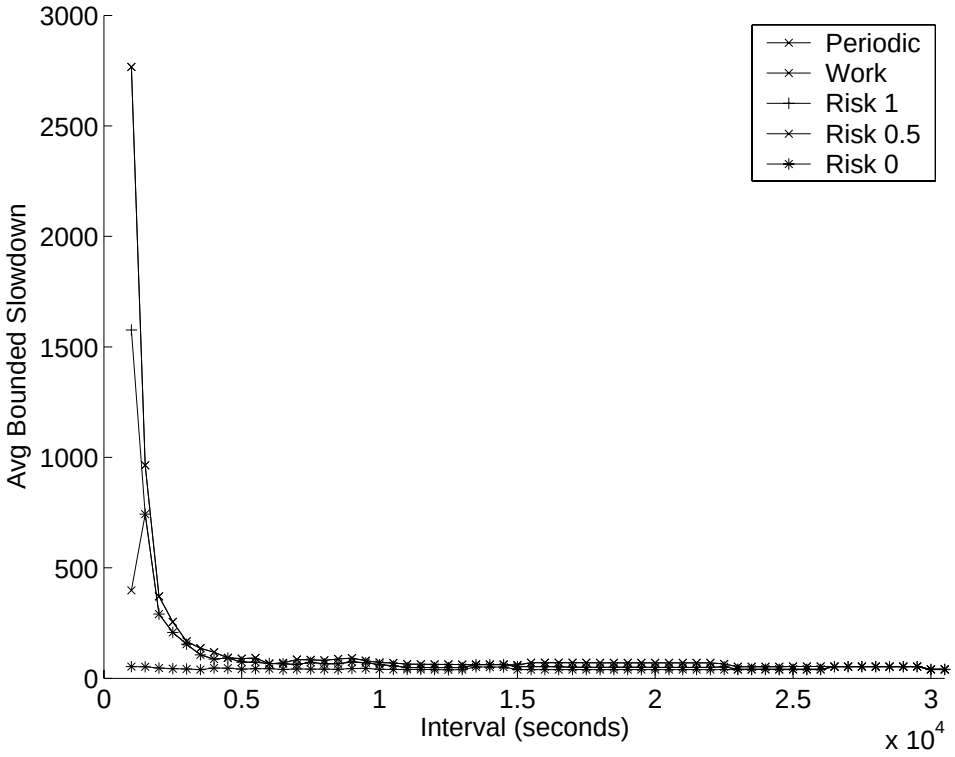

Figure 4-4: Bounded slowdown vs. checkpoint interval in seconds for the NASA job log, using a checkpoint overhead of 720 seconds.

fore, always returned $p_f = 1$, then the Risk 1 and Work curves would be identical. We therefore call the gap between these curves the *false positive gap*.

Another prominent characteristic visible in Figures 4-6 to 4-9 is the stratification of the drop in metric under risk-based checkpointing at different accuracies. For risk-based checkpointing at higher accuracies (and work-based checkpointing for Figure 4-9), this drop occurs below $I = C$ seconds, since $p_f I < C$, in general. For lower accuracies, however, this drop occurs at higher intervals. At these lower accuracies, a greater proportion of failures will be predicted with lower probabilities. For instance, at $a = 0.9$, all failures will have $0 \leq p_f \leq 0.9$, while at $a = 0.2$, $0 \leq p_f \leq 0.2$, $\forall p_f$. Additionally, the number of failures predicted is nondecreasing as a increases, so the average p_f tends to increase with a. Referring back to the heuristic for risk-based checkpointing, lower accuracies will be more likely to satisfy $p_f dI < C$, thus more checkpoints are skipped.

Because of our choice of job start time, response time and wait time tend to be

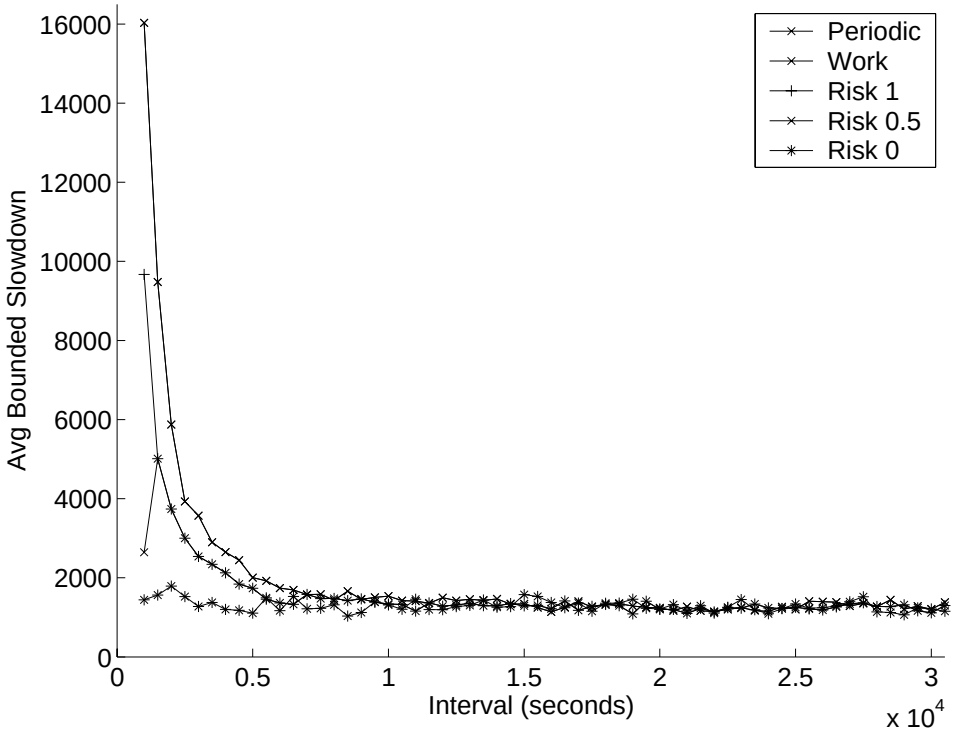

Figure 4-5: Bounded slowdown vs. checkpoint interval in seconds for the SDSC job log on a toroidal topology, using a checkpoint overhead of 720 seconds.

similar to each other. This is clear from a comparison of Figures 4-10 and 4-11, which use $C = 720$ seconds. In general, bounded slowdown, response time and wait time curves for the same input parameters are similar in nature. Bounded slowdown is used because the curves more strongly exhibit important characteristics.

To summarize, periodic checkpointing suffers from an exponential decrease in performance as the checkpoint interval decreases. Work-based checkpointing effectively trims off the most devastating part of this curve by applying a simple run-time heuristic, for those situations in which $C > I$. Further benefit can be achieved by using risk-based checkpointing, where all event predication accuracies fare better than either periodic or work-based checkpointing.

System Utilization

A common measure of performance is average system utilization. Particularly for dynamically-arriving workloads, as jobs are encountered in most production circum-

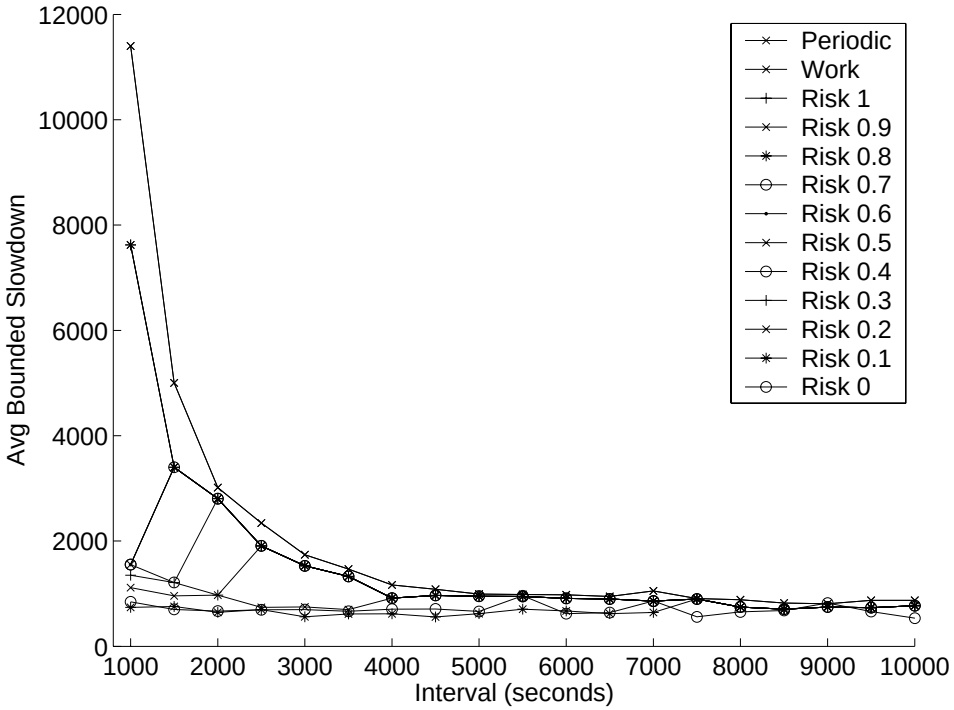

Figure 4-6: A zoom of Figure 4-3, with all prediction accuracy levels.

stances, this is a misleading metric. A long-running job arriving late in the simulation can dominate the running time (M), which solely determines the utilization for a given workload.

Results for the SDSC log on a torus ($C = 720$ seconds) are presented in Figure 4-12. A closer view of the figure, between checkpointing interval 1000-10000 seconds for all accuracies, is shown in Figure 4-13. We see that naïve checkpointing can reduce effective utilization from $\sim 74\%$ to $\sim 55\%$; when $C = 3600$ seconds, utilization dropped from $\sim 67\%$ to $\sim 20\%$. Simple work-based checkpointing increases utilization, in the latter case, by more than 25%. In general, for these parameters, not checkpointing decreases effective utilization. For smaller intervals, however, Risk 0 is optimal for this metric. Once again, stratification is clearly visible in Figure 4-13.

From these results, and the results in Section 4.3.3, it appears that checkpointing does not generally act to improve system-level metrics like utilization and bounded slowdown, for these workloads and failure distributions. Checkpointing increases the effective running time of jobs, and makes efficient scheduling more difficult. In par-

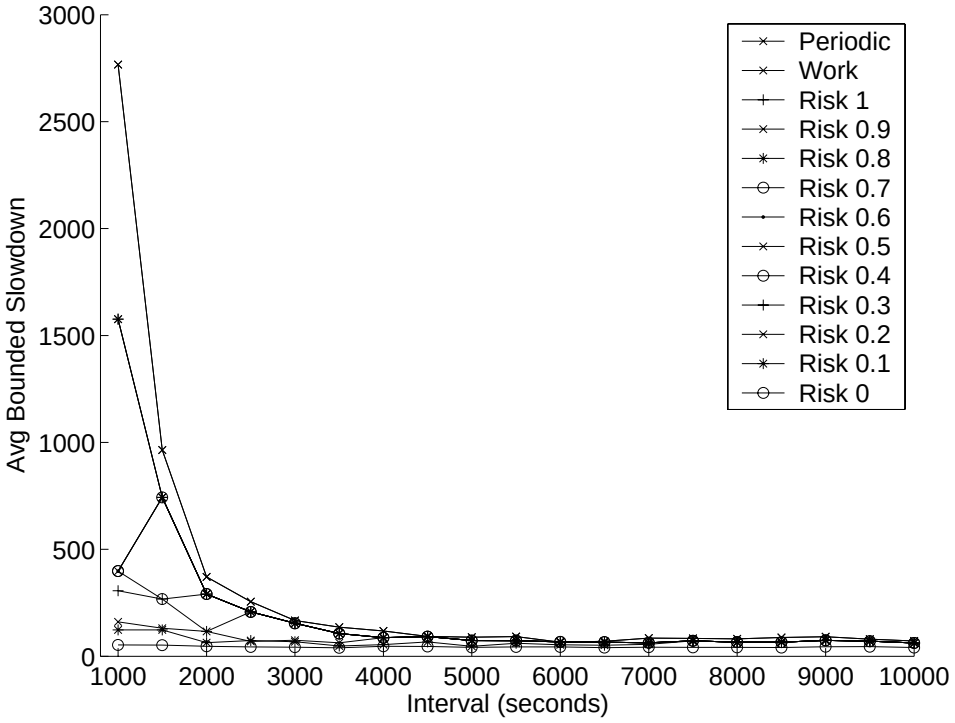

Figure 4-7: A zoom of Figure 4-4, with all prediction accuracy levels.

ticular, it complicates backfilling. Work-based and risk-based checkpointing mitigate this loss of efficiency by skipping checkpoints that the heuristics perceive as being superfluous. In section 4.3.3, we examine the trade-off made for those improvements in performance.

Work Lost

Standard checkpointing is intended to be a selfish act: a job checkpoints in order to minimize the amount of recomputation it will need to perform after a failure. It does not consider the effect of its continued execution on the scheduling of other jobs, or the perceived speed of the cluster. Minimizing the work lost parameter inherently satisfies the goal of checkpointing, and is the basis for the requirements of work-based and risk-based checkpointing. Neither of the algorithms in these simulations have a system value function (S) that would permit consideration of system-level factors that might improve utilization or average slowdown.

Figure 4-14 shows the total amount of work lost due to failures for the SDSC

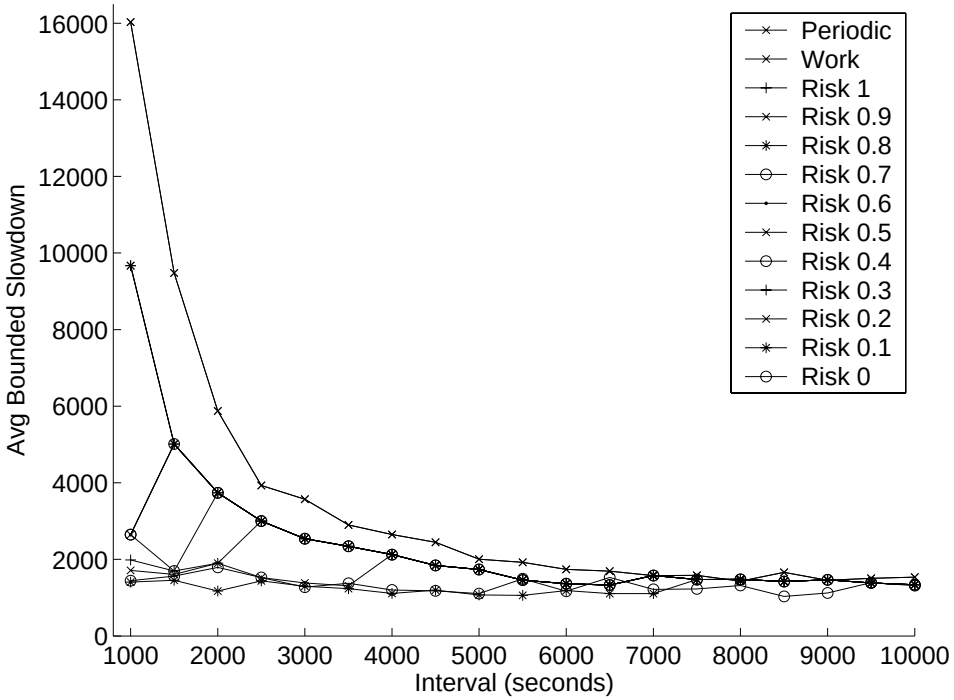

Figure 4-8: A zoom of the torus results in Figure 4-5, with all prediction accuracy levels.

log on a flat cluster. The most outstanding feature is the curve for Risk 0 (no checkpointing), which is distinctly separate from the other curves. The amount of work lost from failures is reduced to nearly the optimal value when the accuracy of the predictor is raised to 50%. Figure 4-15 shows the same results for the torus. Clearly, checkpointing is accomplishing its goal: reducing the amount of work lost due to job failures.

For curves other than Risk 0, where checkpointing is being performed, a higher interval tends to increase the amount of lost work. This is reasonable, because more frequent checkpointing is a common strategy to minimize lost work. The fluctuations in Risk 0 are a consequence of the way in which the jobs happen to be scheduled, and illustrates the variance in the amount of work that may be lost without checkpointing.

A closer look at Figure 4-15, with all prediction accuracies, is shown in Figure 4-16. Compared to no checkpointing, the amount of work lost from failures is reduced by more than 79% when the accuracy of the predictor is raised to 10%, and by 92% at 40% accuracy. In other words, predicting and checkpointing ahead of only 10% of

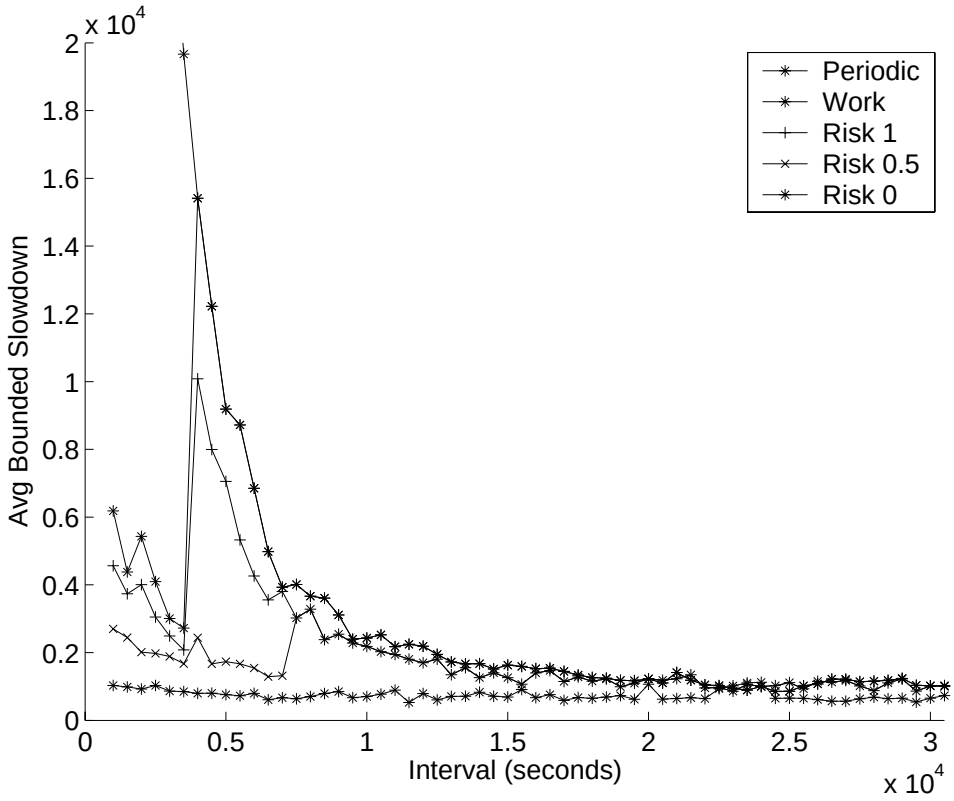

Figure 4-9: Bounded slowdown vs. checkpoint interval in seconds for the SDSC job log, using a checkpoint overhead of 3600 seconds.

all failures makes a huge impact in the amount of lost work. Predicting around half of the failures has the same effect on lost work as checkpointing periodically, whether or not a failure is expected.

Strategy Comparison

This section presents a different view of the results from the previous section, and summarizes the trade-offs offered by these checkpointing heuristics. Figures 4-17 through 4-19 show results for the SDSC log on a flat cluster. The x-axis indicates the type of checkpointing that was used. All plots are for $C = 720$, 3600 seconds and $I = \{1000, 10000\}$ seconds. The results for $C = 720$ seconds represent results when $C < I$. In this case, the work-based heuristic performs every checkpoint; periodic and work-based checkpointing yield the same results.

Consider first the results for $I = 10,000$ seconds. The bounded slowdowns in

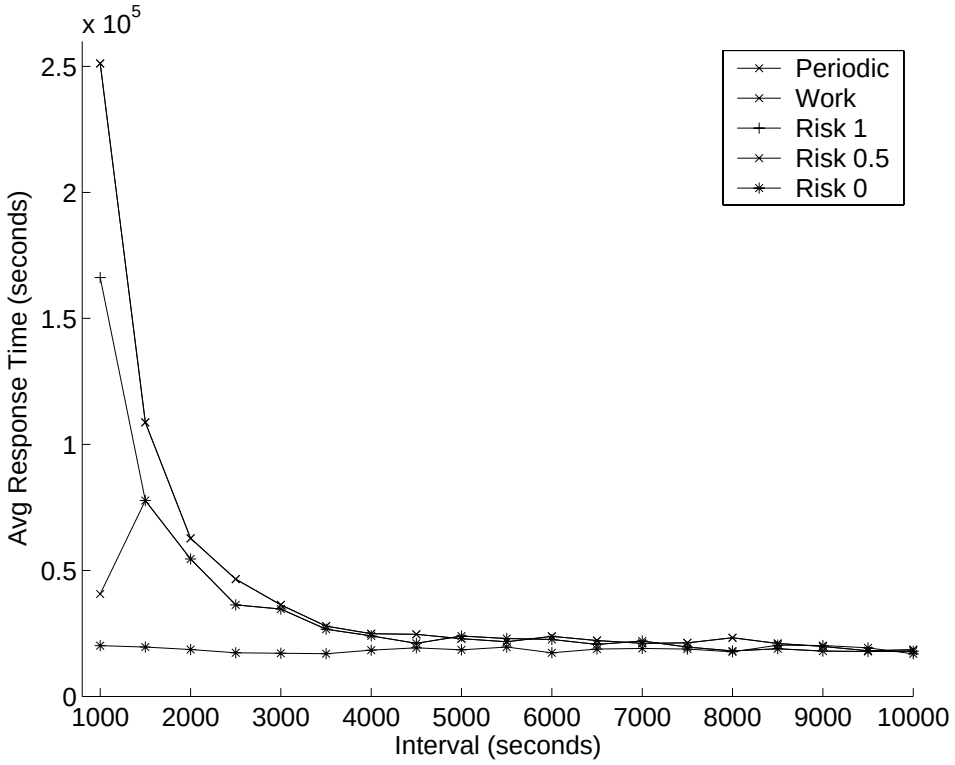

Figure 4-10: Response time vs. checkpoint interval in seconds for the SDSC job log, using a checkpoint overhead of 720 seconds.

Figure 4-17 show a gradual decrease in this metric as fewer checkpoints are performed, for both overheads. While Risk 0 is optimal, note that Risk 0.1 gives nearly the same values. Utilization, graphed in Figure 4-18, shows a similar pattern, with utilization tending to increase as fewer checkpoints are performed. The sole exception is Risk 0 with $C = 720$ seconds, where utilization suffers negligibly. For this interval of $I = 10,000$ seconds, the range of utilization fractions is relatively small.

For $I = 1000$ seconds, periodic checkpointing performs significantly worse than either of our heuristics in both bounded slowdown and utilization. In that case, with $C = 3600$ seconds, work-based checkpointing gave an immediate 25% utilization boost, with an additional 20% being possible if all checkpoints are skipped. Work-based checkpointing reduced bounded slowdown, in this extreme case, by more than a factor of 90. By themselves, these measurements of bounded slowdown and utilization give the impression that checkpointing should be abandoned entirely.

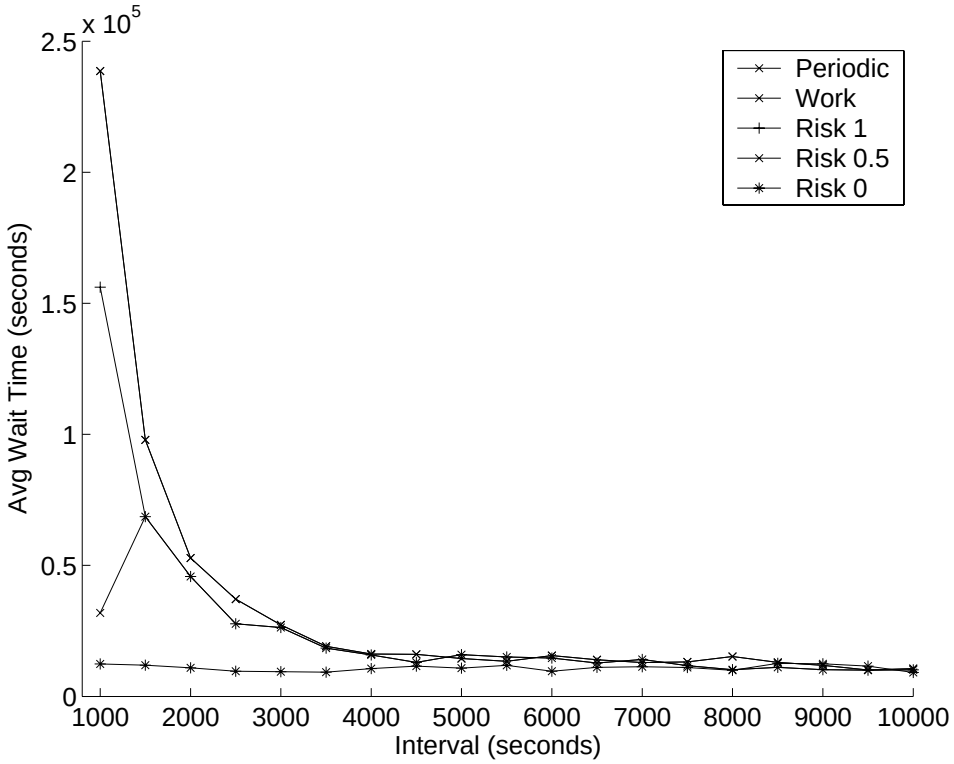

Figure 4-11: Wait time vs. checkpoint interval in seconds for the SDSC job log, using a checkpoint overhead of 720 seconds.

Figure 4-19, however, tells just the opposite. For $I = 10,000$ seconds, the amount of lost work increases as the amount of checkpointing decreases. For a system with $C = 720$ seconds, a prediction accuracy of 10% reduces the amount of lost work as much as periodic checkpointing, while also bringing bounded slowdown and utilization to near optimal values. Recall that event prediction with accuracy as high as 70% has already been achieved [27]. We conclude that an application should spend as little time checkpointing as possible, but no less, and that those important checkpoints can be effectively identified with event prediction.

The results for $I = 1000$ seconds were slightly more complicated: the lost work plot was a U-shaped curve. Too much checkpointing meant that jobs tended to hit more failures, and there was a greater chance of failing during a checkpoint. On the other hand, too little checkpointing was even worse. For such an extreme case, work-based checkpointing is a simple heuristic for getting a considerable boost in

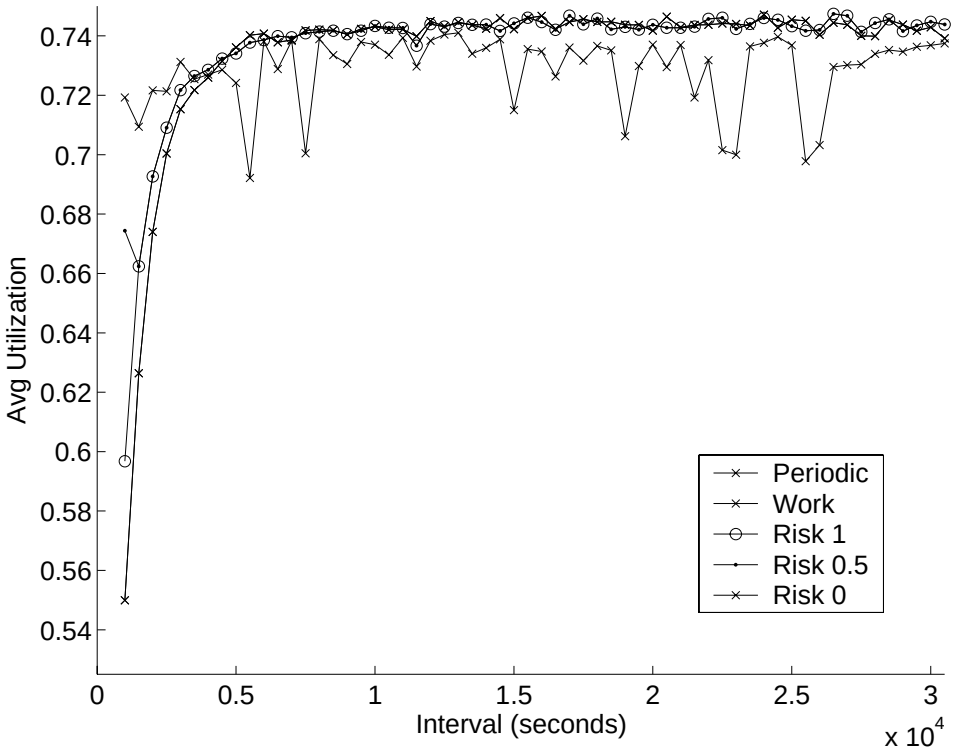

Figure 4-12: System utilization vs. checkpoint interval in seconds for the SDSC job log on a toroidal architecture, using a checkpoint overhead of 720 seconds. This is representative of the results with other inputs.

system-level metrics, while cutting the amount of lost work in half. That is, by intelligently *skipping* checkpoints according to the work-based heuristic, the amount of lost work can be *decreased*. If system-level metrics are most important to the system administrator, risk-based checkpointing may be an appropriate solution. Once again, by using event prediction with a mere 10% accuracy, the amount of lost work can drastically reduced, while simultaneously increasing bounded slowdown and utilization to near-optimal levels.

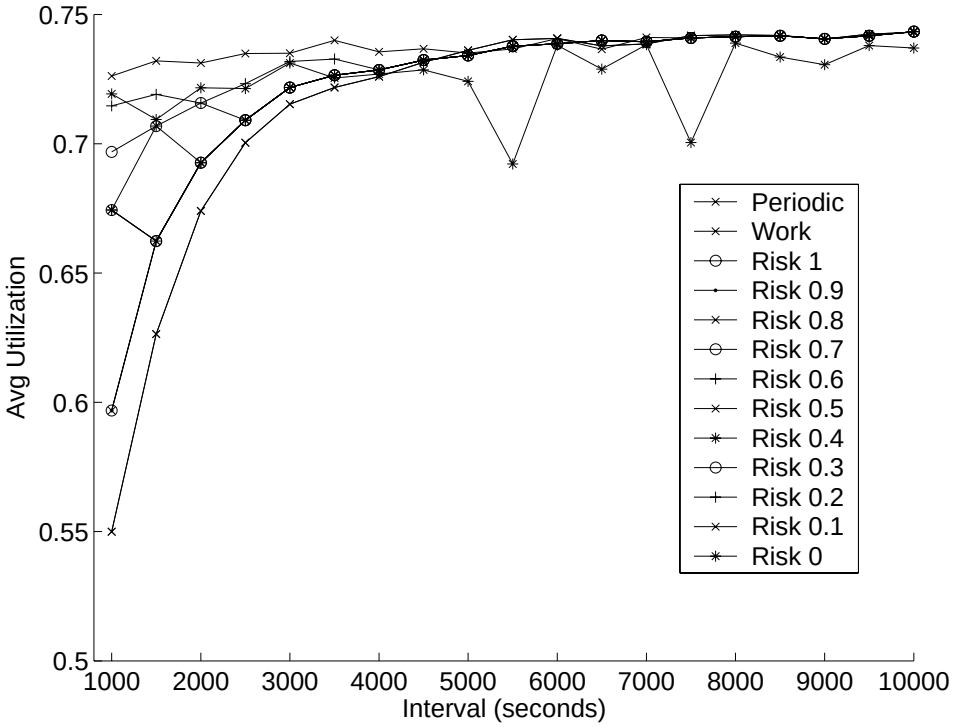

Figure 4-13: Zoom of Figure 4-12 with all accuracies.

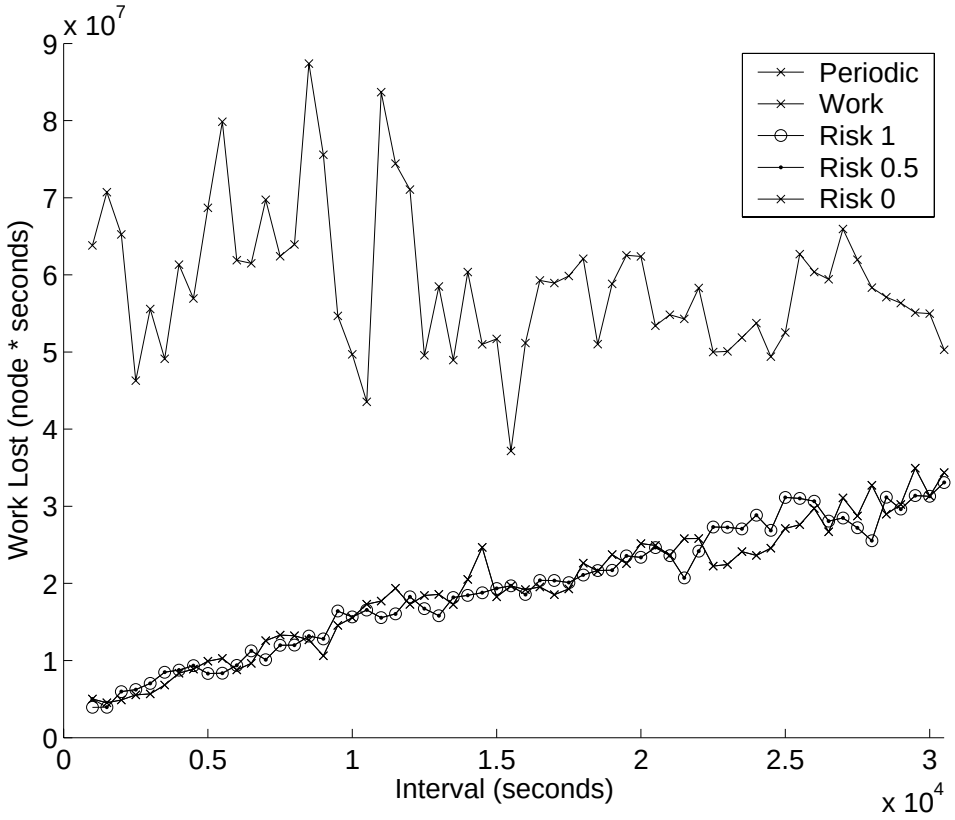

Figure 4-14: Lost work vs. checkpoint interval in seconds for the SDSC job log, using a checkpoint overhead of 720 seconds.

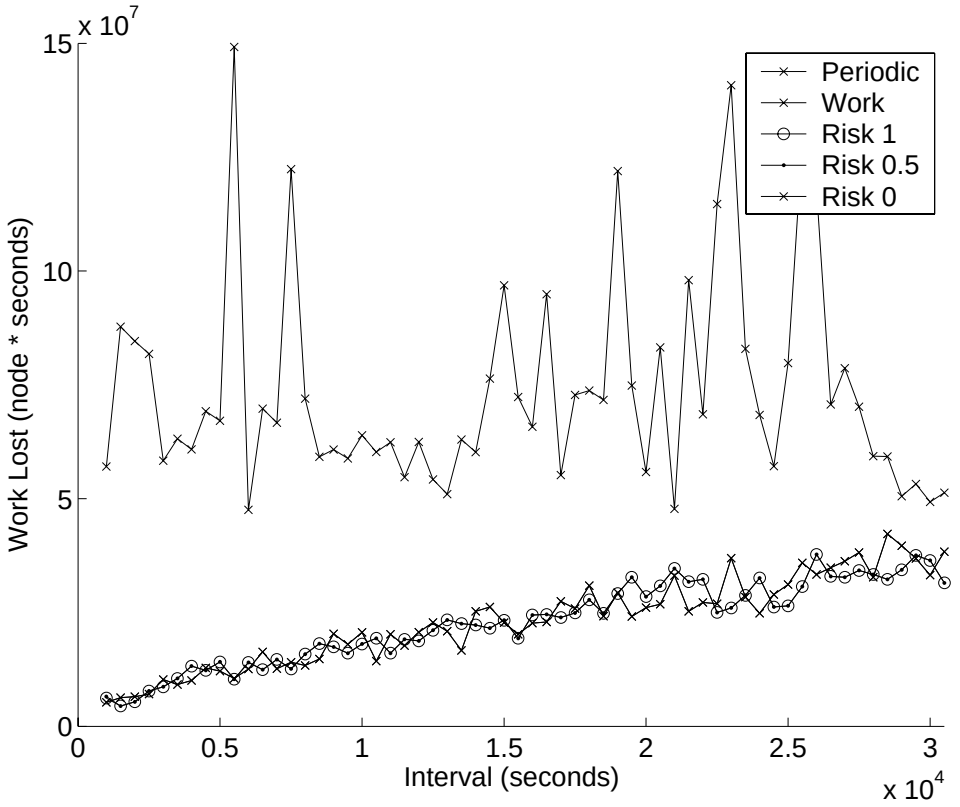

Figure 4-15: Lost work vs. checkpoint interval in seconds for the SDSC job log on a toroidal architecture, using a checkpoint overhead of 720 seconds.

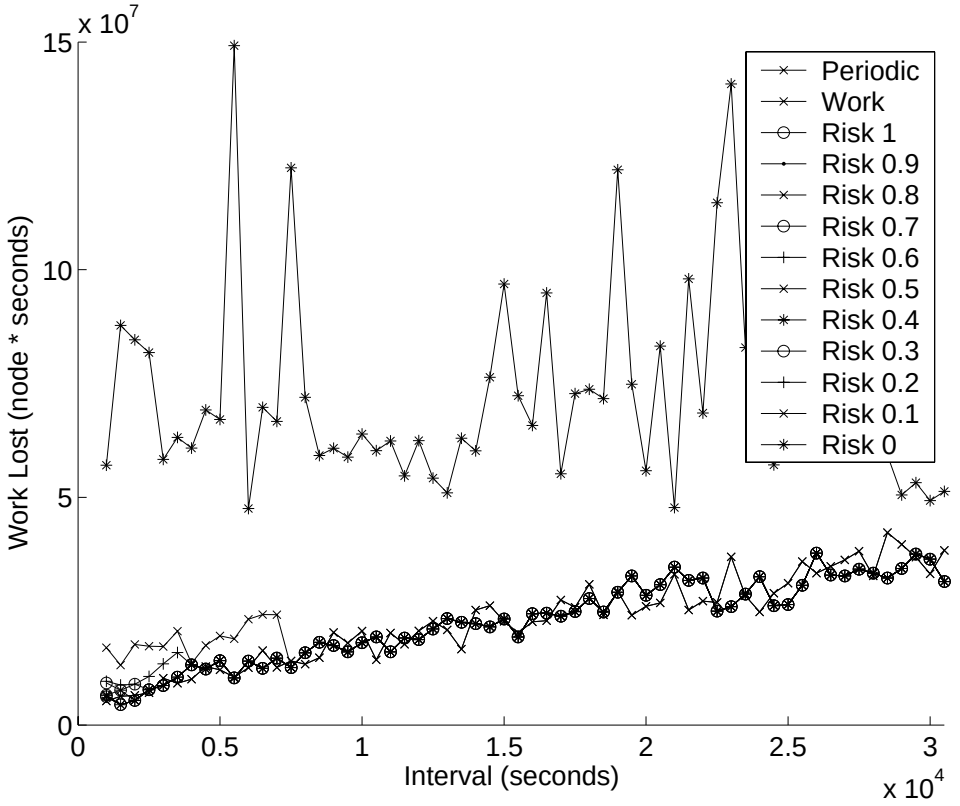

Figure 4-16: A closer look at Figure 4-15, with all prediction accuracies

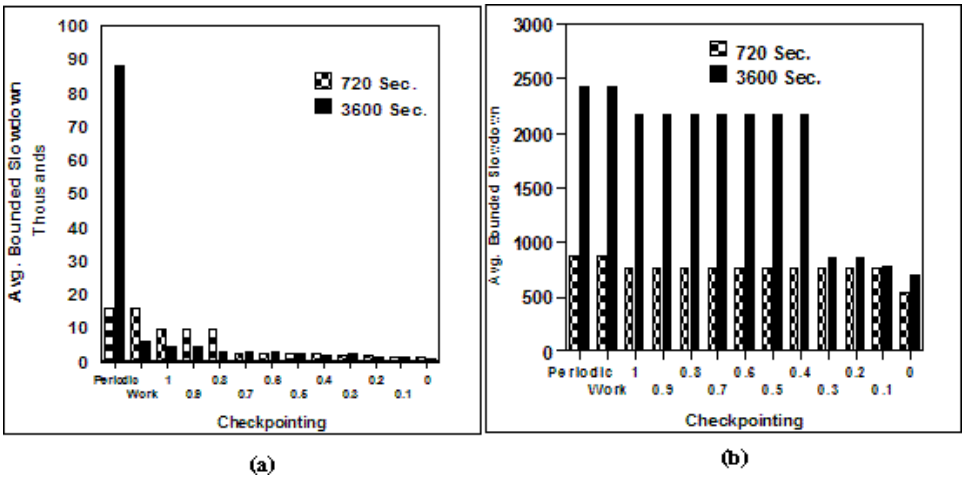

Figure 4-17: Comparison of different checkpointing policies: bounded slowdown, SDSC log. $C = 720$ and 3600 seconds. (a) $I = 1000$ sec., (b) $I = 10,000$ sec.

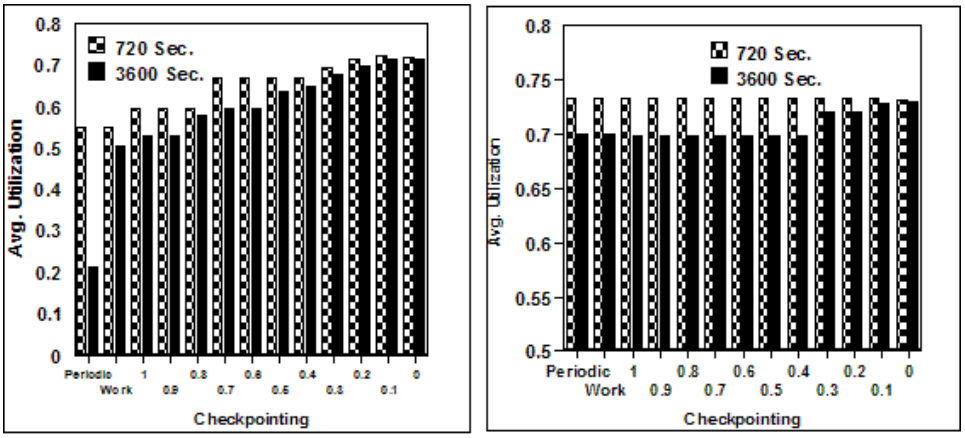

Figure 4-18: Comparison of different checkpointing policies: system utilization, SDSC log. $C = 720$ and 3600 seconds. (a) $I = 1000$ sec., (b) $I = 10,000$ sec.

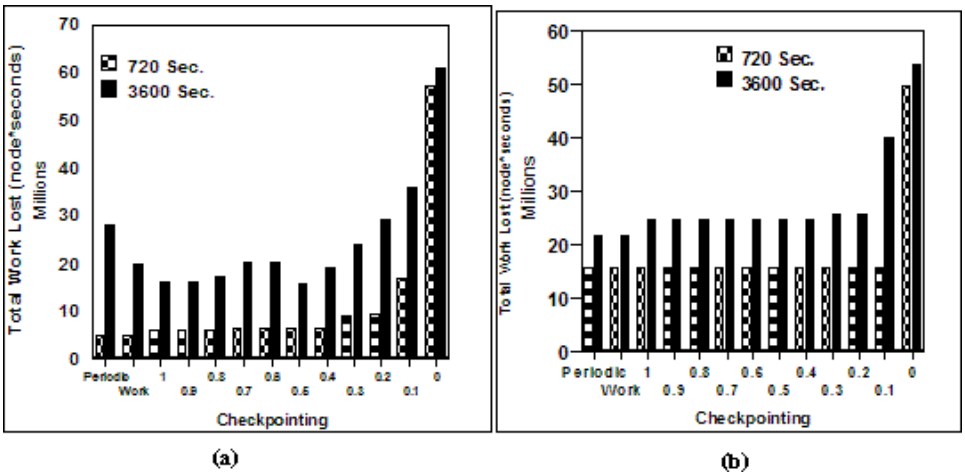

Figure 4-19: Comparison of different checkpointing policies: lost work, SDSC log. $C = 720$ and 3600 seconds. (a) $I = 1000$ sec., (b) $I = 10,000$ sec.

Chapter 5

Contributions

As computing systems continue to grow in scale, new techniques must be developed for ensuring reliable and efficient operation. Periodic checkpointing, at either the application or system level, is quickly becoming an infeasible solution to the challenge of minimizing the cost of failures. This thesis has presented a unique checkpointing scheme called cooperative checkpointing, in which many components of the system work together to save application state to stable storage in an efficient, timely manner.

Specifically, the programmer places checkpoint requests in the application code where the critical state is minimal. These requests are optimized by the compiler. At runtime, checkpoint requests are sent to the gatekeeper, which either performs or skips each checkpoint request. The gatekeeper uses input from a multitude of sources to make informed choices. In this way, cooperative checkpointing combines many of the strengths of previous techniques while negating many weaknesses.

Cooperative checkpointing can be modeled as two parts: the static placement and optimization of checkpoint requests in the code, and the dynamic online decisions made by the gatekeeper. The behavior of the gatekeeper can be modeled as an online algorithm. Where C is the checkpoint overhead and I is the request interval, a worst-case analysis proves a lower bound of $(2 + \lfloor \frac{C}{I} \rfloor)$-competitiveness for deterministic cooperative checkpointing algorithms, and proves that a number of simple algorithms meet this bound. In the expected-case, an optimal periodic checkpointing algorithm that assumes an exponential failure distribution may be arbitrarily bad rel-

ative to an optimal cooperative checkpointing algorithm that permits a general failure distribution. An application using cooperative checkpointing may make progress 4 times faster than one using periodic checkpointing, under realistic conditions. Simulation results show that, in extreme cases, cooperative checkpointing improved system utilization by more than 25%, reduced bounded slowdown by a factor of 9, while simultaneously reducing the amount of work lost due to failures by 30%.

Cooperative checkpointing is a topic of active research. An implementation is currently in progress for BlueGene/L. The next steps include developing more specific cooperative checkpointing algorithms for supercomputing systems, improving health monitoring and event prediction systems, extending the simulator to support the dynamic case, and using that simulator to perform more extensive experiments.

More concisely, this thesis makes the following contributions:

- Formalizes the problem of checkpointing on realistic systems, in which checkpoint overheads are dominated by I/O bottlenecks and where failures may occur in predictable ways.

- Introduces cooperative checkpointing, a novel technique for overcoming these challenges, whereby the application requests checkpoints and the system dynamically decides which to perform and which to skip.

- Analyzes cooperative checkpointing as an online algorithm. The worst-case and expected-case analyses prove that cooperative checkpointing can do significantly better than periodic checkpointing.

- Proposes an embodiment of cooperative checkpointing that takes advantage of its full potential and suggests what kind of supporting infrastructure is required (or desirable) for this scheme.

- Simulates the performance of a rudimentary cooperative checkpointing implementation and presents the results.

Bibliography

[1] N. Adiga and The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *SC2002, Supercomputing, Technical Papers*, November 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] B-Course. A web-based data analysis tool for bayesian modeling. URL: http://b-course.cs.helsinki.fi, 2004.

[4] J. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer-Verlag, New York, 1985.

[5] P. J. Brockwell and R. Davis. *Introduction to Time-Series and Forecasting*. Springer-Verlag, 2002.

[6] M. F. Buckley and D. P. Siewiorek. Vax/vms event monitoring and analysis. In *FTCS-25, Computing Digest of Papers*, pages 414–423, Pasedena, CA, June 1995.

[7] M. F. Buckley and D. P. Siewiorek. A comparative analysis of event tupling schemes. In *FTCS-26, Intl. Symp. on Fault-Tol. Computing*, pages 294–303, June 1996.

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

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

[10] Elmootazbellah N. Elnozahy and James 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.

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

[12] E. Krevat, J. G. Castanos, and J. E. Moreira. Job scheduling for the bluegene/l system. In *JSSPP*, pages 38–54, 2002.

[13] W. Kuo and V. R. Prasad. Annotated overview of system-reliability optimization. *IEEE Transactions on Reliability*, 49(2):176–187, 2000.

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

[15] 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.

[16] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.

[17] 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.

[18] 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*, April 2005.

[19] 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*, April 2004.

[20] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo, California, 1988.

[21] J. Pitman. *Probability*. Springer-Verlag, New York, 1993.

[22] 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.

[23] 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.

[24] James M. Roewe. Fault tolerant dynamic agent systems. Master's thesis, Massachusetts Institute of Technology, 2005.

[25] 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.

[26] 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 *ACM SIGKDD, Intl. Conf. on Knowledge Discovery and Data Mining*, pages 426–435, Washington, DC, August 2003.

[27] 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.

[28] 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*, August 2003.

[29] 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.

[30] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

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

[32] M. M. Tsao. *Trend Analysis and Fault Prediction.* PhD dissertation, Carnegie-Mellon University, May 1983.

[33] R. Vilalta and S. Ma. Predicting rare events in temporal domains. In *Proceedings of IEEE Conf. on Data Mining (ICDM'02)*, Japan, 2002.

[34] G. M. Weiss and H. Hirsh. Learning to predict rare events in event sequences. In *Proceedings* 4^{th} *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 359–363, 1998.

[35] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, and R. K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *10th Workshop on JSSPP, SIGMETRICS*, 2004.