Cluster Fault-Tolerance: An Experimental Evaluation of Checkpointing and MapReduce through Simulation

Thomas C. Bressoud [#1], Michael A. Kozuch [*2]

[#] *Department of Mathematics and Computer Science, Denison University*
P.O. Box 810, Granville, Ohio, USA
[1] bressoud@denison.edu

[*] *Intel Research Pittsburgh, Intel Corporation*
4720 Forbes Avenue, Pittsburgh, Pennsylvania, USA
[2] michael.a.kozuch@intel.com

***Abstract*—Traditionally, cluster computing has employed checkpointing to address fault tolerance. Recently, new models for parallel applications have grown in popularity—namely MapReduce and Dryad, with runtime systems providing their own reexecute-based fault-tolerance mechanisms, but with no analysis of their failure characteristics. Another development is the availability of failure data spanning years for systems of significant size at Los Alamos National Labs (LANL), but the Time Between Failure (TBF) for these systems is a poor fit to the exponential distribution assumed by optimization work in checkpointing, bringing these results into question. The work in this paper describes a discrete event simulation driven by the LANL data and by models of parallel checkpointing and MapReduce tasks. The simulation allows us to then evaluate and assess the fault tolerance characteristics of these tasks with the goal of minimizing the expected running time of a parallel program in a cluster in the presence of faults for both fault tolerance models.**

I. Introduction

The number and size of parallel applications in cluster-computing environments continues to grow at a rapid rate. Areas such as *data-rich computing* [1] are moving problems with an overwhelming scale of data such as video-scene understanding, natural language translation and data mining applications from rare to everyday, driven by the availability of vast new datasets. Further, clusters are no longer constrained to be built in-house. The advent of cloud computing [2] enables the creation of virtual clusters in the cloud to suit the application, enabling both transient and permanent cluster applications. With larger input datasets and the implied longer processing times, the number of processing elements, n, applied to the problems is increasing as well.

Accompanying this growth is an increasing probability of failure of some of the components involved in the computation. Fault-tolerance techniques, which allow a computation to make progress in the presence of individual component failures, are required. Checkpointing with rollback-recovery [3] is one fault-tolerance technique that has long been employed [4], [5], [6], [7], [8] in cluster systems to address this requirement.

In rollback-recovery, a failed process is rerun from an application state saved earlier to stable storage, with periodic checkpoints over the application execution used to reduce the amount of lost computation. Checkpointing with rollback-recovery may be employed for single processing element applications ($n = 1$) as well as for parallel applications. When $n > 1$ and the processing elements communicate, such interaction creates causal dependencies between the processes which must be handled by the checkpointing scheme, either through *coordinated checkpointing*[1] or by risking cascading rollbacks of the processes. If $n = 1$, or if $n > 1$ and the application is embarrassingly parallel, with no causal communication between processes, then *uncoordinated* (independent) *checkpointing* may be used without risk of cascading rollback. The parallel programming models in which checkpointing may be employed include message passing systems (e.g. MPI [9]) as well as custom crafted and independently scheduled sets of processes. The performance issue in parallel applications using checkpointing is to determine the interval at which to initiate checkpoints so that the expected running time of the application in the presence of failures is minimized.

In efforts to simplify programming for parallel applications, particularly ones with a clear data decomposition, new paradigms have been introduced and have grown in popularity—namely MapReduce [10] and Dryad [11]. These systems comprise both a programming model *and* a runtime system. The runtime system partitions and manages the data distribution and automatically defines and schedules instances of the execution elements on processors in the system, managing the data in-from and data out-of the execution elements. In the model, the programmers job is simplified and centers around creating one or more sequential program stages, called *tasks*, that implement a sequential program on their "slice" of the data. The output of these tasks can be directed (by the

[1] *Communication-induced* checkpointing is an alternative technique that avoids rollback propagation.

runtime) to other tasks or to the final output. In MapReduce there are two stages of computation (the map stage and the reduce stage), while Dryad generalizes this "dataflow" view of parallel computation to an arbitrary directed acyclic graph of computation with the data flowing along the edges of the graph. These new parallel programming paradigms are being applied to both existent problems and to the data-rich types of problems mentioned above.

The runtime system in both of these paradigms provides a model-specific technique for fault tolerance. First, the runtime must support the detection of the failure of processing nodes. Because it is also responsible for scheduling, the runtime knows what tasks have failed as a result of a node failure and can redistribute and schedule redundant instances of failed tasks on non-failed nodes. The runtime must also ensure the data in and out of the restarted tasks are directed appropriately. The correctness of the technique depends on an underlying assumption of deterministic execution of the tasks—given a particular input, instances of the same task will, even if executed repeatedly, generate the same output. The model provides the required independence of tasks within a stage and forces all causal dependencies into the dataflow between stages.

With the growth in scale of parallel applications, the fundamental issue of performance is more important than ever. The performance of fault-tolerant parallel applications is determined by both (i) the distribution of failures in our cluster systems and by (ii) the fault tolerance techniques we employ. Application designers are faced with real choices. Given their application characteristics, they must decide the appropriate programming paradigm and the appropriate fault tolerance technique that minimizes the expected run time of their application.

Checkpointing has a long history of research on the performance question. Focusing on $n = 1$ and assuming node failures are governed by a Poisson process with a failure rate of λ, a mathematical analysis can derive an approximation to the optimal checkpoint interval in terms of λ and the checkpoint overhead [12], [13]. Essentially, this analysis balances the tradeoff between the overhead of taking checkpoints with the expected amount of lost work when failures occur. These theoretical results have been brought into question as a number of studies of real failure data (primarily from networks of workstations) [14], [15], [16], [17], [18] have failure distributions that are fit poorly by the exponential distribution that is a necessary condition of the Poisson process assumption of the checkpointing work.

As a newer programming paradigm and associated fault tolerance mechanism, the MapReduce and Dryad systems have a significant absence of research work on their performance in the presence of failures. There is no equivalent mathematical analysis that starts with a failure distribution and derives expected run time in the presence of failures nor optimization of the parameters of the system to minimize expected run time of the parallel applications under execution.

Relative to our objective of understanding the performance of these varied fault tolerance mechanisms for cluster systems as we scale n and under real-world failure rates, we can summarize the deficiencies of prior work:

1) Theoretical work in checkpointing relies on a failure rate distribution that is not borne out by real failure data.
2) Checkpointing analysis has focused on the $n = 1$ case and, when $n > 1$ checkpointing systems are examined, the failure rate is assumed to follow $\lambda_n = n \cdot \lambda$.
3) Studies of failures in real systems have almost exclusively been targeted at networks of workstations, the important exception being the Schroeder and Gibson work [18]. One would expect the closely managed and administered nodes of a cluster would have different failure characteristics than a heterogeneous network of workstations.
4) The new programming paradigms and their associated reexecute-based fault-tolerance mechanism lack any fault-tolerance based analysis other than the basic demonstration of their correct operation. Specifically, there is no analysis of the failure-related performance of these systems as we scale n.

The real-world data provided by Los Alamos National Labs (LANL) in 2005 [19] is central to improving the current situation. This failure data encompasses 22 systems in operation at LANL from 1996 through 2005, including both NUMA (Non-Uniform-Memory-Access) and 2- and 4-way SMP node clusters. These are the systems whose characteristics are studied in [18].

Our solution approach is to use the real-world failure data provided by a targeted set of cluster system from the LANL datasets and to build a discrete event simulation that can model both checkpoint-based fault-tolerance and the independent reexecution fault-tolerance mechanism for MapReduce style programs. We can then vary the factors that impact the overhead and performance of these system as they execute in a simulated cluster and can evaluate the performance of both mechanisms. Our end objective is to give tools and data to inform the choice of programming paradigm and fault-tolerance mechanism by the application designer.

The research work described in this paper fills a vital gap in understanding the performance of cluster systems in the presence of failures and as highlighted by the deficiencies enumerated above. By using the LANL data, we obtain a realistic failure rate distribution that is applicable to the types of cluster systems we are interested in and does not suffer from the mismatch of the distribution assumed by a mathematical analysis. It also gives us a feasible approach for the new paradigms, which currently have no mathematical analysis. By developing models of parallel MapReduce and Checkpointing applications used in the simulation, we overcome the dominance of the $n = 1$ analysis of most of the checkpointing work, and can vary n as we execute scenarios in the simulation. In addition, we may easily model variations to perform "what-if" performance analysis on different models.

The remainder of this paper is organized as follows. In Section II we look at related work that is the conceptual

starting point for the current investigation. Section III presents an analysis of the failure data for the specific systems under study. We then describe our simulator and its models of Checkpointing and MapReduce in Section IV. Given an understanding of the simulator and the source data, we present a failure rate analysis in Section V. Sections VI and VII present our results in the simulation of checkpointing and MapReduce parallel applications. Finally, in Section VIII, we summarize our results and discuss future directions.

II. Related Work

The availability of node failure data in real systems has long been an issue. When collected by vendors, the data is held close and not disseminated. The early notable exception is the work by Gray [20], [21]. The majority of the work since then has focused on gathering failure data from a loose distribution of heterogeneous machines on a network using means such as probing the set of target machines, recording reboot actions, and using event logs/problem reports to infer time-to-failure and time-to-recover [14], [15], [16], [22], [23], [24][2]. Several of these works report the same conclusion that the failure distribution is poorly fit by an exponential. Unfortunately the data sets in these studies are typically small and/or the data collection interval is also small. As noted, these systems may not be representative of clusters.

More recent work has focused failure analysis on more tightly coupled systems of nodes for parallel applications. Work by Sahoo et al. [17] studies a production environment of nearly 400 heterogeneous server machines whose dominant workloads were parallel scientific codes using MPI. Data was collected using log files on the servers over a year and a half period. Another study by Oliner and Stearly [25] used raw system logs to explore five supercomputers, all ranked on the Top500 Supercomputers List as of June 2006 [26]. Logs span periods from 104 days up to 558 days. Although the datasets in both these studies match more closely the cluster systems that we are interested in, the focus of these studies is on simply understanding the failure rates and causes of events in the logs, so there is no attempt to use the data in then understanding performance for any fault tolerance mechanism.

In similar fashion, the studies by Schroeder and Gibson [18], [27] analyze the 22 LANL high performance systems. The first of these studies was coincident with the public release of the LANL failure data in 2006. The study examines failure root causes and failure correlation along with failure arrival rates and repair times, and comes to the conclusion that the failure rates are poorly fit by an exponential distribution. These authors have continued efforts to make real-world failure data available (including the LANL data) for use by other researcers [28]. While these more recent works are more applicable to cluster systems, they do not take the next step of applying these realistic failure rates to fault tolerant mechanisms to evaluate performance.

[2]Both [15] and [24] use the dataset of [14] for their own analysis in addition to providing additional data sets.

TABLE I
LANL Systems in Study

System	Nodes	Cores	Dates	Failures
18	1024	4096	May-02 to Aug-05	3997
19	1024	4096	Oct-02 to Aug-05	3284
20	256	1024	Dec-01 to Sep-05	2124

Of the papers examining failure rates, the landmark work by Plank and Elwasif [15] goes beyond analyzing the distribution and explores the effects on the performance of the checkpoint restart-recovery fault tolerance mechanism. In many ways, this work is the most comparable to ours in that it uses real failure data and also uses the technique of simulation to assess performance. Failure data from three data sets drove the simulation, with all three based on a heterogeneous network of machines. The bulk of the work focuses on the $n = 1$ case, with some effort to ask the question whether the assumption of $\lambda_n = n \cdot \lambda$ is valid for the given datasets. Unfortunately, these data sets were too disparate to answer that question, giving inconsistent and inconclusive results.

Our own work builds on the these last two works by using select clusters from the LANL data to obtain quality input data to our simulation that is representative of the clusters we are interested in. We focus on the parallel case and, by employing models for different fault tolerance mechanisms, are able to evaluate both checkpointing and map-reduce techniques.

III. Source Failure Data

While the failure data included from LANL includes 22 systems, only the three largest clusters of SMPs were selected as input datasets for the current work. These were Systems 18 (S18), 19 (S19), and 20 (S20)[3], and their basic characteristics are listed in Table I. We use the term *node* to refer to an independently bootable machine and *core* to refer to an execution core within a node. Within all three of these clusters, the processor/memory chip model are the same, but individual nodes can vary in the amount of memory installed.

The LANL failure data, obtained from a Remedy database, recorded each time interval that a node was down; for each of the three systems, the failure data was processed into a file consisting of a set of "up"-intervals, (t_i, t_j) for each node, with time measured relative to the first up time in the system's dataset. These up-intervals then give an explicit time-between-failure (TBF) for each node and a time-to-recovery implicit in the time from the end of one up-interval until the time to the start of the next up-interval. The relative frequency of the TBF for the three systems is plotted in Figure 1. While S18 and S19 exhibit very similar distributions, with Mean TBF for each at 251 days and 255 days, S18 has a MTBF of 149.25 days along with a larger coefficient of variation (1.48 for S20 compared with 1.07 and 1.03 for S18 and S19).

To assess these empirical distributions against the exponential and other standard distributions, we performed the

[3]In [18], these are given identifiers 7, 8, and 5, respectively.

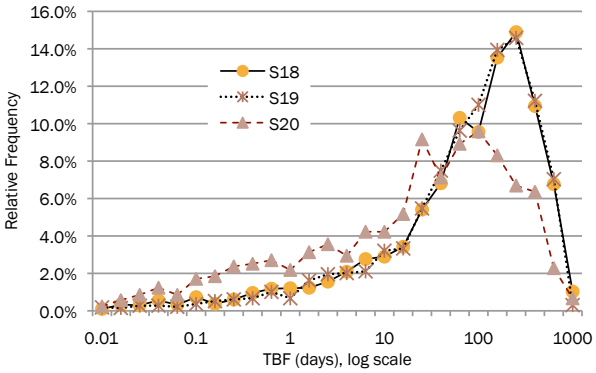

Fig. 1. Relative Frequency of Time Between Failures for LANL Clusters

TABLE II
KOLMOGOROV-SMIRNOV STATISTIC ANALYSIS

Distribution	S18	S19	S20
Beta (t)	0.020	0.022	0.038
Weibull (t)	0.070	0.067	0.049
Gamma (t)	0.083	0.078	0.062
Exponential (t)	0.105	0.090	0.242
Lognormal (t)	0.135	0.127	0.117
critical value (v)	0.021	0.023	0.030

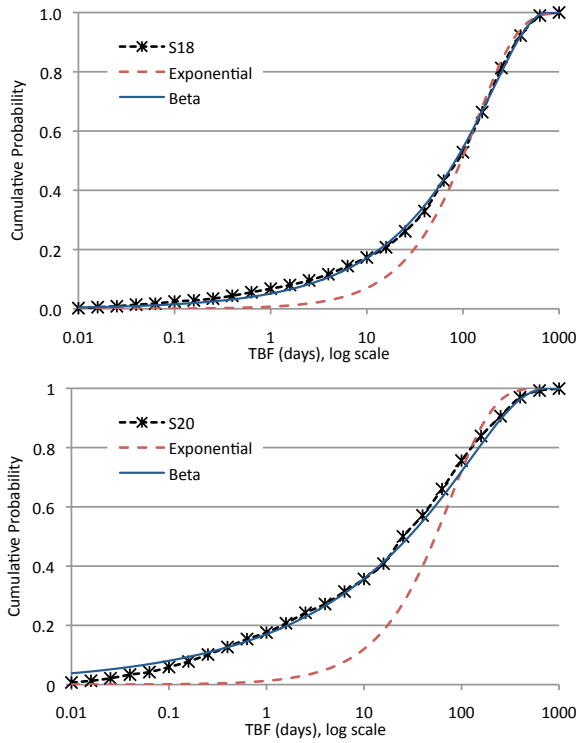

Fig. 2. Cumulative Distribution of Time Between Failures for S18 and S20

Kolmogorov-Smirnov (K-S) goodness of fit test. We started by parameter fitting five distributions to the empirical data for the three systems. The distributions chosen included the exponential, the lognormal, the gamma, the Weibull, and the beta distributions. We then calculated the K-S statistic, t, for each of these, as well as the critical value for the K-S statistic (v) given $\alpha = 0.05$, and present the results in Table II.

When $t < v$, we cannot reject the null hypothesis that this empirical data is from the given distribution. Thus the beta distribution passes this test for S18 and S19 and is a better fit than the other distributions for S20.

In Figure 2, we present the Cumulative Distribution Function (CDF) for the empirical TBF distribution, the exponential distribution, and the beta distribution for S18 and S20 (the S19 CDFs appear qualitatively identical to S18 and are omitted here). These graphs visually confirm the K-S analysis indicating the superior fit of the beta distribution over the exponential for the failure rate of these cluster systems.

IV. SIMULATOR

The simulator we designed and built for this work, called CFTsim (**C**luster **F**ault **T**olerance **sim**ulator), is written in Python and makes use of the the *SimPy* (Simulation in Python) discrete-event simulation package [29] . The package is object-oriented and process-based and fit our needs for a flexible system in which we could introduce new models for the behavior of fault tolerance mechanisms in an easily extensible framework.

Using a set of up-intervals as input, the simulator takes (i) one of our datasets (S18, S19, or S20), (ii) a scale parameter, n, that specifies the number of cores to employ for each application instance A, and (iii) a set of parameters specific to a fault tolerance model, and steps through the events of nodes/cores going up and down and interacting with the parallel application and fault-tolerance model. The simulator also has a configuration parameter specifying the number of cores per node. The goal is to obtain a set of simulated execution times (E_i) for A over the time interval defined by the dataset. One execution of CFTsim over the dataset time interval defines a *run*.

We define a *task instance* (or simply *task*) as a unit of work of A assigned to a single core. During a run, the *scheduler* component of CFTsim is responsible for selecting a random starting time for A and for randomly selecting a set of n cores to be used for the tasks of A. This set of cores is fixed and dedicated for use by this instance of A until its completion. Note that the size of the set of tasks of A may be $> n$, depending on the programming and fault tolerance mechanism being modeled. Depending on N, the total number of cores in the cluster, along with n and the start times, multiple instances of A may be simulated concurrently (although on different cores) in a given run.

In addition to the scheduler, the simulation of the cluster involves modeling application instances, the task instances making up a given application instance, and the cores upon which the task instances execute. Each of these elements in CFTsim is implemented as a Python/SimPy process object. The behavior of the application instances and task instances is defined by the fault-tolerance model and is described below. The design of the operation of the cores must be agnostic

TABLE III
CHECKPOINTING MODEL PARAMETERS

Param	Description
n	Number of cores for an application instance.
F	Failure free execution time for a single task instance (days).
I	Checkpoint interval (min.).
C	Checkpoint overhead during which no task work is accomplished (min.).
L	Checkpoint latency – interval of time from start of checkpoint until it is committed (min.).
R	Recovery time – interval from a core coming back up until a prior checkpoint is loaded and work may be resumed (min.).
Coord	Boolean indicating if checkpointing is coordinated or independent.

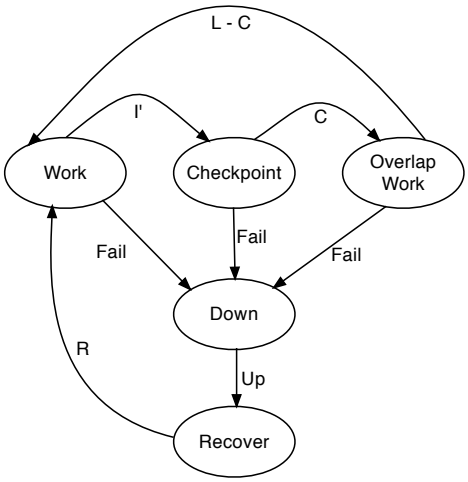

Fig. 3. Checkpointing Finite State Machine

to the fault-tolerance mechanism and works as follows. A core is associated with a particular node, and is given the set of up-intervals specific to that node. A core also has an incoming task queue, whose elements are task instances. The task queue is filled, as appropriate, by an application instance. When available, the core activates at most one task instance from its queue. Beyond activation and recording task instance completion, a core simply alternates between UP and DOWN states, notifying any interested elements (the scheduler and/or the application instance and/or the task instances themselves) of these state transitions.

A. Checkpointing Model

The checkpointing model is defined by the parameters given in Table III. A checkpointing application instance, A, is given n cores by the scheduler and creates a single task per core. Each task executes its own finite state machine (FSM) as given in Figure 3. A task begins in the *Work* state and initializes the amount of work remaining, Z, to F. After I', the task begins taking a checkpoint, where $I' = I$ initially or after a recovery, and I' is $I - L$ the rest of the time[4]. To allow for different checkpointing designs, C is distinguishable from L in that C is the overhead during which no work is accomplished, and L is the total time before a checkpoint is committed. This allows a checkpoint to be initiated, but then to continue concurrently with additional work and then to be committed at a later point in time. Once a checkpoint is committed, a rollback recovery need not re-execute any work implied by the checkpoint, and Z is decreased by the work accomplished in the cycle. This cycle can be seen in the FSM through the *Checkpoint* and *Overlap Work* states back to the *Work* state.

At any time, a failure may occur and is signaled by the core on which the task is executing. If *Coord* is true, the failure of a core is conveyed to all the task instances for the app. This is denoted by the **Fail** transition to the *Down* state. When the failed core comes back up, the task (or all tasks if *Coord* is true) transition through *Recover* and return to the *Work* state after R time has elapsed.

[4]This allows I to define a consistent period for the initiation of the checkpointing process.

TABLE IV
MAPREDUCE MODEL PARAMETERS

Param	Description
n	Number of cores allocated to a map-reduce application.
M	Number of map tasks.
R	Number of reduce tasks.
F_M	Failure free execution time of a single map task (days.).
F_R	Failure free execution time of a single reduce task (days.).

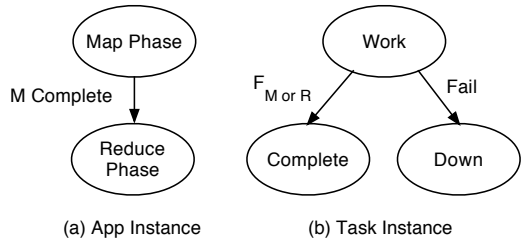

Fig. 4. MapReduce Finite State Machines

B. MapReduce Model

The map-reduce model is defined by the parameters given in Table IV. A map-reduce application instance A is given n cores with which to complete its work. A is responsible for shepherding the set of M map tasks through the map phase of the computation followed by the shepherding of the R reduce tasks through the reduce phase (see Figure 4(a)). Only when all map tasks have completed does A transition to the reduce phase, and once this occurs, the work of the map phase is committed and need not be re-executed even with subsequent core failures. For each phase, A maintains sets tracking the tasks – *Outstanding* for a task which has not yet performed its work, *Inprogress* for the set of tasks currently operating on some core, and, for each core, a set of *Completed* tasks for that core. In the absence of failures, the management of a task proceeds from *Outstanding* to *Inprogress* (when the task is scheduled) and from *Inprogress* to *Completed.*

The FSM for an individual map or reduce task is much simpler than the checkpointing model, as seen in Figure 4(b).

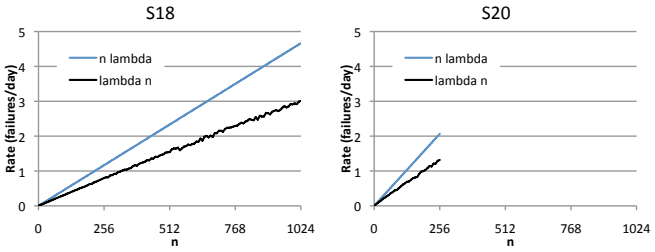

Fig. 5. Failure Rate as n Scales

A task starts in the *Work* state and either transitions to *Complete* (after time equal to F_M or F_R), if no failure occurs, or transitions to *Down* if the core that it is executing on fails.

When a failure occurs, A takes the task that was *Inprogress* along with all of the *Complete* tasks for the affected core and places them back in the *Outstanding* set. This allows us to model the reallocation of these tasks to non-failed cores as implemented in the MapReduce runtime.

V. Failure Rate Analysis

Given the MTBF for a set of components, the rate, λ, associated with the failure interval is given by $1/\text{MTBF}$. In a parallel system, the overall system is considered *up* if all n elements are up, and the overall system is considered *down* if *any* of the n elements have failed. We use λ_n to denote the rate of failure for the n-collection and λ_n is given by $1/\text{MTBF}_n$. As noted in Section I, work in assessing fault tolerance in parallel systems has assumed failure independence and failure arrival following a Poisson process. This implies that

$$\lambda_n = n \cdot \lambda \tag{1}$$

This has been used in predicting the performance of coordinated checkpointing in parallel systems [30], [31], [32].

However, the availability of the LANL failure data provides us with a new opportunity to evaluate whether or not the simplifying assumptions that give us Equation (1) are valid. In particular, we can employ CFTsim with a specialized application model to experimentally determine MTBF_n as we vary n for our input datasets of S18, S19, and S20, and we can then assess whether or not Equation (1) holds for these systems. For this application model, we let the CFTsim scheduler select a random set of n cores and initiate the app instance. The app instance then simply waits until all cores are up and measures, as its expected execution time E_i, the time interval until some core in the set goes down. This completes the app instance with an expected execution time equivalent to the TBF for the given set of n cores and given start time. By iteratively repeating the process with randomly generated start timesover a large sampling of n cores from the overall set of N, we obtain our experimental value of MTBF_n, and may then calculate λ_n.

In Figure 5 we show the results of executing CFTsim with this λ_n app model for each of the systems under study. For each system, we vary n up to the number of nodes in the system, and operate in a configuration with one core per node[5]. The empirical determination of λ_n yields a lower failure rate than the $n\lambda$ assumption, and we see that the difference can be substantial: theory overpredicts the failure rate consistently by 50%, even across the different systems. This discrepancy with theory would be explained by (i) a lack of failure independence and/or (ii) a violation of the Poisson assumptions. The latter was substantiated previously by the poor fit of the exponential to the empirical CDF in Figure 2.

VI. Checkpointing

The availability of the LANL failure data provides us with the opportunity to evaluate how closely existing theoretical results match experience with real systems. In particular, we examine (a) the expected execution time when checkpointing is not employed and (b) the optimal checkpointing interval when it is.

The execution time, E_F, of a program (assuming $n = 1$) without checkpointing is predicted by Equation 2 (see Duda [33]).

$$E_F = \frac{(e^{\lambda F} - 1)}{\lambda} \tag{2}$$

Equation 2 can be extended for parallel applications (that are coordinated in the sense that a failure on any core requires all cores to revert to a prior checkpoint) by using either $n\lambda$ (denoted [P1]) or our empircally determined λ_n (denoted [P2]) for the failure rate.

In the absence of checkpointing, ($I = F$), if the application has causal communication, the failure of *any* of the cores involved in the computation would require all of the tasks to restart from the beginning. Figure 6 shows the execution time for applications running in such an environment on S18 and on S20[6] both with $F = 4$ and $F = 8$. On each of the plots, we show the CFTsim calculated execution time as a function of the number of cores employed for the app. In each plot we also show the predicted performance based on Equation 2, for P1 ($n\lambda$) and for P2 (empirical λ_n). Here, we see that the accepted theoretical results (P1) do not match the simulated execution times that are based on empirical data; in some cases ($F = 8$), the discrepancy is quite dramatic. While the P2 calculation is consistently superior to the P1 calculation, it also fails to accurately predict the exection times based on empirical data.

We also consider the no checkpointing case where the application is able to perform its tasks independently. In this case, the failure of one core has no effect on the tasks executing on the other cores, and the total execution time will be determined by the maximum time (relative to the application start time) of any core to attain F continuous uptime. Results for this case are shown in Figure 7 for S18 and S20 and, for each system, we show the $F = 4$ and $F = 8$ cases. As expected, the execution time approaches a maximum that is dependent on the underlying failure rate.

[5]Increasing the number of cores per node would artificially create dependence between the failures of the cores.

[6]Again, the simulation of S19 generated results that are nearly identical to S18 and are not presented.

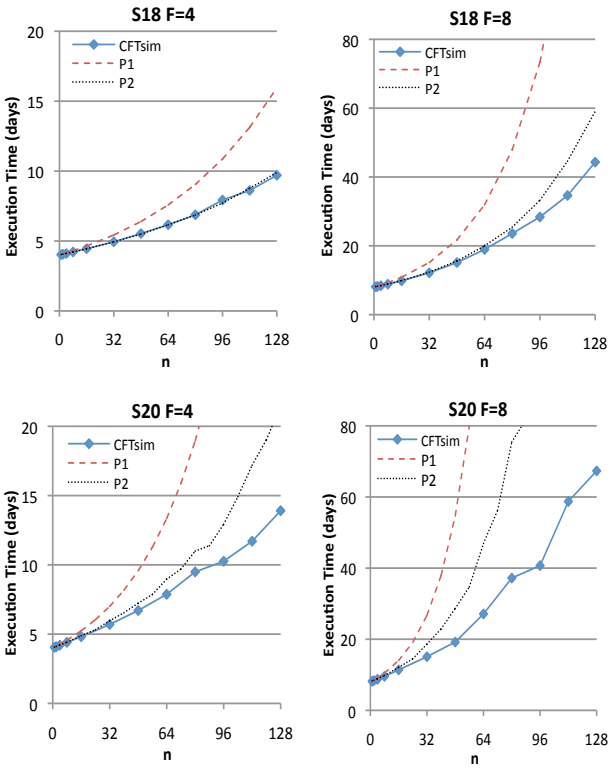

Fig. 6. Execution Time for Coordinated Application (No Checkpointing)

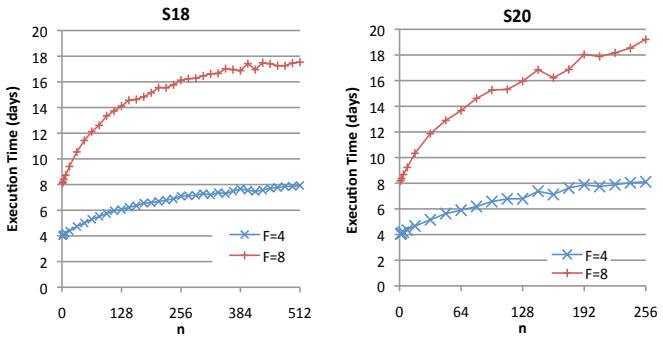

Fig. 7. Execution Time for Uncoordinated Application (No Checkpointing)

When checkpointing is used, a checkpoint interval, I, must be selected, and this interval represents a tradeoff between failure resilience and performance overhead. If checkpoints are taken too infrequently, significant computation may be lost when the program restarts from a checkpoint, but if checkpoints are taken too frequently, the creation of checkpoints can become a significant overhead to execution time.

Equation 3 predicts the optimal checkpoint interval value, denoted I_{opt}, assuming $n = 1$ and $C = L = R$ (result by Young [12]).[7]

$$I_{opt} = \sqrt{2C/\lambda} \tag{3}$$

[7]While our simulator can model the cases where $C \neq L \neq R$ (see extended theory by Vaidya [13]), space constraints prevented us from presenting such results here.

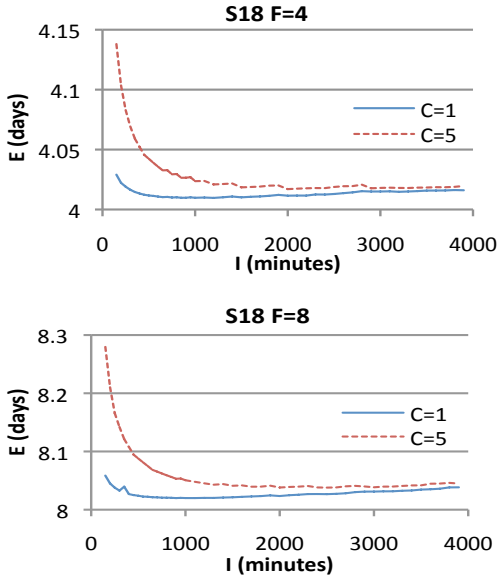

Fig. 8. Coordinated Checkpointing App Execution Time vs. I for $n = 1$

Again, this result can be extended to coordinated checkpointing applications by using either $n\lambda$ (denoted [P1]) or our empirically determined λ_n (denoted [P2]) for the failure rate.

To see the effect of checkpoint interval on execution time, we simulate program execution under different values of I using CFTsim. To enable clear comparison with the theoretical work, we begin with the $n = 1$ case, illustrated for S18 in Figure 8. For each value of F at 4 days and 8 days, we compare two different checkpoint overhead times, $C = 1$ and $C = 5$ minutes[8]. For these runs, $L = C$ and $R = 10$ minutes. In each graph we present simulated execution time as the checkpoint interval, I, varies.

Figure 9 shows the effect of checkpoint interval on execution time for larger values of n (64, 128, and 256), using the empirical failure data for S18 and the same values for C (1 and 5 minutes) and F (4 and 8 days), by presenting the CFTsim-calculated execution times for various values of I. The checkpoint interval tradeoff is obvious from the figure; with small values of I, the induced checkpointing overhead severely affects performance, but as I increases beyond some optimal point, performance degrades as failures cause increasing re-execution.

By inspecting the data from Figure 9, we determined the optimal values for I, and those values are presented in Tables V and VI along with the values for I_{opt} predicted by Equation 3 for (P1) and (P2). Note that while the theory suggests that the optimal interval should be independent of F, we do observe different values when using the empirical data. Moreover, we see that the predicted values differ from the CFTsim derived values, and this difference increases as we scale n, so that for $n = 256$, the difference is approximately a factor of 2. Fortunately for S18, this miscalculation in I_{opt}

[8]These values of C are consistent with [34], which suggests an upper bound on C of 12 minutes for the BlueGene/L system.

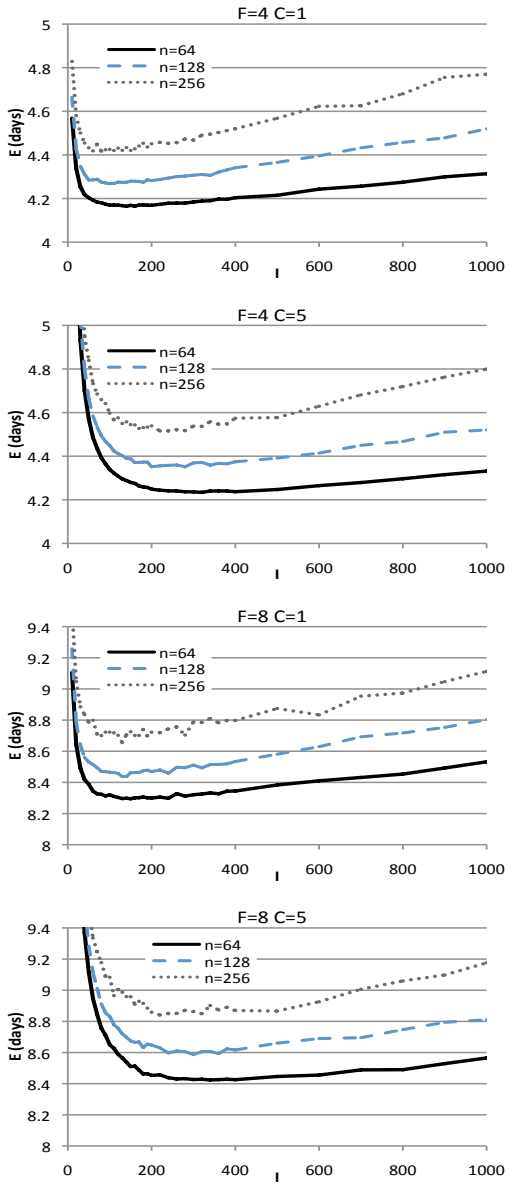

Fig. 9. Coordinated Checkpointing App Execution Time vs. I for n=64,128,256 (S18)

TABLE V
COORDINATED APPLICATION I_{opt} FOR S18, C=1 MIN.

			CFTsim	
n	P1	P2	$F=4$	$F=8$
1	795	795	1200	900
64	99	119	140	150
128	70	85	100	130
256	50	60	80	130

typically causes a performance degredation of less than 1% because, while the difference between the predicted I_{opt} and true I_{opt} may appear to be large, both values lie in the flat range of the curve. Whether the impact of miscalculating I_{opt} remains limited in larger systems is unknown.

In the absence of causal communication, uncoordinated checkpoints may be employed to achieve fault tolerance. Figure 10 shows simulated execution time as we vary the checkpoint interval for this uncoordinated checkpointing case, and we show data series for $n = 1, 64, 128$ and 256 for S18 and for the cases where $F = 4$ and $F = 8$. For both of these graphs, we used a checkpoint overhead, C, of 1 minute. When n is small, we achieve minimum execution times of 4.01 and 8.02 days, respectively, and this is obtained when $I = 1000$.

TABLE VI
COORDINATED APPLICATION I_{opt} FOR S18, C=5 MIN.

			CFTsim	
n	P1	P2	$F=4$	$F=8$
1	1778	1778	2000	2500
64	222	266	320	340
128	157	190	280	300
256	111	134	220	220

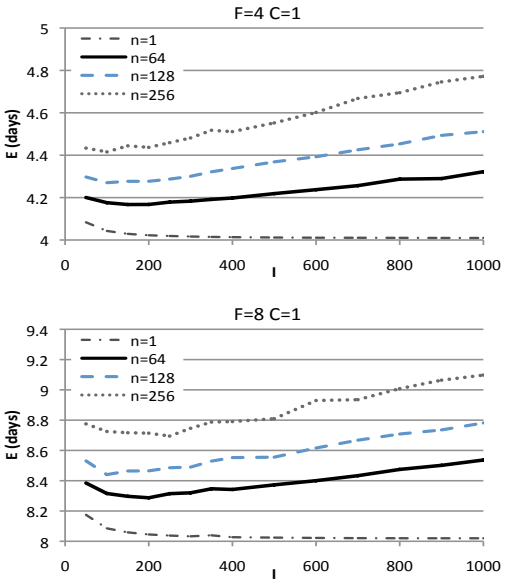

Fig. 10. Uncoordinated Checkpointing App Execution Time vs. I for n=1,64,128,256 (S18)

VII. MAPREDUCE

A significant advantage of the MapReduce application model is its comprehension of fault-tolerance. Each application consists of a phase of independent map tasks followed by a phase of independent reduce tasks. The MapReduce runtime monitors the tasks and restarts those that fail whether due to hardware failure or for other reasons.

The main goal of our MapReduce simulation is to determine the characteristics of this model in the presence of hardware failures. To that end, we use a methodology similar to the one described for checkpointing (Sections VI). However, in this case, the fault tolerance mechanism is one of reexecution, where all map or reduce tasks from a failed core are reallocated dynamically to operational cores whether the tasks had completed or not (i.e. partial results are stored locally), and execution is repeated in its entirety.

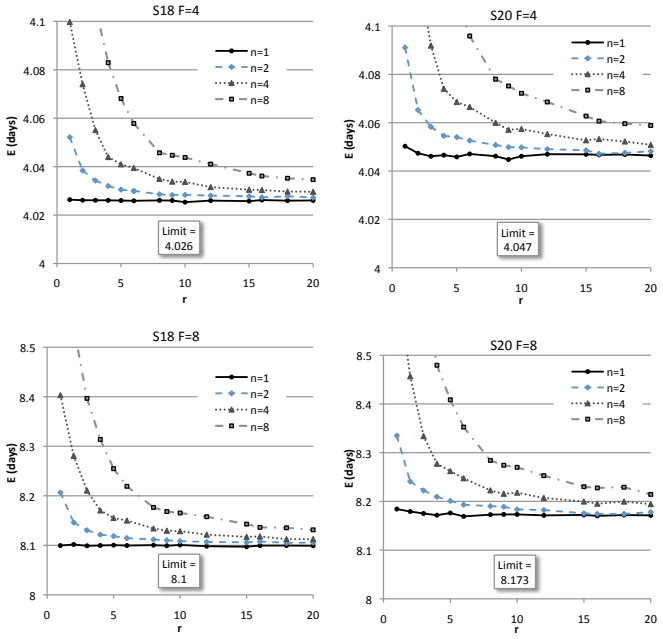

Fig. 11. MapReduce performance for small n

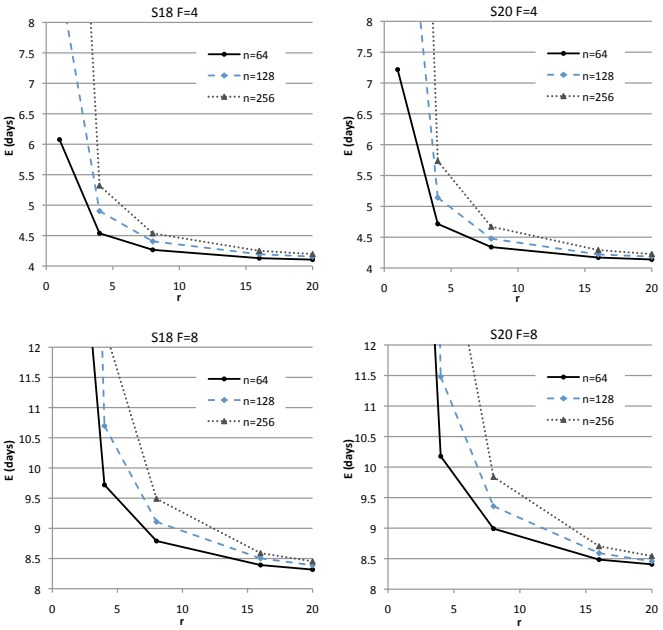

Fig. 12. MapReduce performance for large n

To isolate the reexecution behavior of MapReduce, which is similar to independent checkpointing in that the failure of one core does not have an impact on the work currently underway at the other cores, we model each application as comprising M identical map tasks, which each have a failure-free execution time of F_M, and an empty reduce phase. When we compare this with the checkpointing fault tolerance mechanism, the MapReduce approach elects to pay the failure penalty of reexecution rather than the ongoing overhead of periodic checkpoints.

To understand how MapReduce performs as we scaled our applications, we repeat our simulations for various values of n. The number of map tasks, M, is our selectable parameter; however, it is often more convenient to consider the number of (failure-free) tasks per core, r, where $r = M/n$, and to fix the amount of work executed on a core, denoted F, and from this and r, compute F_M. In the following experiments, we use $F = 4, 8, 12$, $r = 1 \ldots 20$, and $n = 1 \ldots 256$[9].

We begin by looking at small n in order to understand the fault tolerance characteristics of MapReduce. The simulated execution times for a small cluster (assuming failure characteristics similar to either S18 and S20) are shown in Figure 11.

As one would expect, the execution time derived by CFTsim decreases as we increase r. The $n = 1$ case demonstrates the limit, whose value is governed by the probability of failure of a single core (λ_1). When n increases, but we still have a small ratio of map tasks per core, a failure results in a greater amount of re-execution work on the non-failed remaining cores. As r increases, and consequently M, the granularity of the work to be reexecuted (F_M) is reduced, and a failure allows the redistribution of the tasks of the core to be "spread out" over the non-failed cores. When r is small and F_M is correspondingly larger, the redistribution is not as efficient, so there is greater disparity between those cores that have received re-execution work and those that have not. We also expect and observe a diminishing return as M increases. However, this limit does not change as n varies because we are fixing the amount of work per core and the failure of one core does not affect other cores, similar to the independent checkpointing case.

[9]By way of comparison, recent work [35], reported the execution of a MapReduce application with $n = 3800$, $M = 80000$, and $F_M \approx 300$ min. The LANL cluster size did not permit the simulation of n that large, but the total work, $n * F_M$, falls in the range of parameters we evaluated.

TABLE VII
VALUES OF r FOR MAPREDUCE PERFORMANCE WITHIN 1% OF LIMIT

	n					
	2	4	8	64	128	256
$F = 4$	1/1/2	3/3/4	6/6/8	36/36/36	64/64/48	64/64/64
$F = 8$	2/2/2	4/4/6	8/8/12	48/48/48	64/64/64	64/64/64
$F = 12$	2/2/2	6/5/8	15/12/16	48/48/48	64/64/64	80/64/64

Figure 12 presents MapReduce for S18 and S20 with larger values of n. We observe the same trend toward the limit. The natural question is: How large does M need to be to get "close" to the limit? To gain some understanding into this question, we defined "close" to be within 1% of the limit and used CFTsim on all three systems with F values of 4, 8, and 12 to determine the value of r at which the expected running time was within 1% of the limit. Table VII presents this information, where each entry in the table is of the form $x/y/z$ where x represents the value of r for S18, y represents the value of r for S19, and z for S20.

We see that the required value of our ratio of map tasks to cores increases with n. For the lower values of n, r appears

to match well with n and then level off. Note, however, that since $r = M/n$, each of these entries must be multiplied by n to get the total number of map tasks M. Consequently, this levelling-off property is critical as cluster systems scale; otherwise, M would grow with n^2— placing significant strain on the MapReduce task scheduler.

VIII. Summary and Conclusion

As parallel systems become increasingly prevalent, understanding the real-world fault behavior of these systems is becoming increasingly important. This paper evaluated the conventional wisdom for fault-tolerance in clusters in the context of multi-year empirical failure data.

Using a simulation-based methodology, we showed that the emprical data exhibits the properties of a beta distribution rather than the expected exponential distribution. We also showed that the empirical data does not exhibit the independent failures typically assumed in theory.

Given these observations, we demonstrated that several theoretical results, which assumed properties not exhibited by the real-world data, could not be expected to hold in practice. In particular, the empirical data indicated that applications with coordinated tasks may exhibit better performance in the absence of checkpointing than predicted. Further, the predicted optimal checkpointing interval may deviate from the true optimal by a factor two.

Finally, we evaluated the performance of MapReduce in the context of real-world failure data and identified the pressure to decrease the size of individual map tasks as the cluster size increases.

The continuance of this research will focus on the MapReduce model and its fidelity. The simple reexecution performance cost focus of the current work must be extended to address overheads in scheduling and managing large numbers of map and reduce tasks, as well as the cost of data redistribution that must occur in the presence of failures to supply reexecution tasks their input. We also wish to analyze real applications and systems and thus to provide CFTsim with targeted model parameters.

References

[1] Intel Research, "Data-Rich Computing," http://techresearch.intel.com/articles/Exploratory/1594.htm.

[2] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, 2008.

[3] M. Elnozahy, L. Alvisi, Y. min Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, Sept. 1996.

[4] J. Plank and K. Li, "Ickp: a consistent checkpointer for multicomputers," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 2, no. 2, Summer 1994.

[5] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Usenix Technical Conf.*, 1995.

[6] G. Stellner, "Cocheck: Checkpointing and process migration for mpi," in *Proc. of the Int'l Parallel Processing Symp. (IPPS)*, 1996.

[7] Y. Wang, Y. Huang, K. phong Vo, P. yu Chung, and R. Kintala, "Checkpointing and its applications," in *Proc. IEEE Fault-Tolerant Computing Symp*, 1995.

[8] M. Treaster, "A survey of fault-tolerance and fault-recovery techniques in parallel systems," *ACM Computing Research Repository (CoRR)*, vol. 501002, 2005.

[9] Message Passing Interface Forum, "Message Passing Interface (MPI) Standards," http://www.mpi-forum.org/.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. of the Symp. on Operating Systems Design and Implementation (OSDI)*, 2004.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems (EuroSys)*, 2007.

[12] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, 1974.

[13] N. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *Computers, IEEE Transactions on*, vol. 46, no. 8, Aug 1997.

[14] D. Long, A. Muir, and R. Golding, "A longitudinal survey of internet host reliability," in *Symp. on Reliable Distributed Systems*, 1995.

[15] J. S. Plank and W. R. Elwasif, "Experimental assessment of workstation failures and their impact on checkpointing systems," in *Int'l Symp. on Fault-Tolerant Computing*, 1998.

[16] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a LAN of Windows NT based computers," in *Symp. on Reliable Distributed Systems*, 1999.

[17] R. K. Sahoo and M. S. Squillante, "Failure data analysis of a large-scale heterogeneous server environment," in *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, 2004.

[18] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, 2006.

[19] Los Alamos National Lab, "Operational data to support and enable computer science research," http://institutes.lanl.gov/data/fdata/.

[20] J. Gray, "Why do computers stop and what can be done about it," in *Proc. of the Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[21] ——, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. on Reliability*, vol. 39, no. 4, 1990.

[22] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Networked Windows NT system field failure data analysis," in *Proc. of the Pacific Rim Int. Symp. on Dependable Computing*, 1999.

[23] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do Internet services fail and what can be done about it?" in *Proc. of the USENIX Symp. on Internet Technologies and Systems (USITS)*, 2003.

[24] D. Nurmi, J. Brevik, and R. Wolski, "Modeling machine availability in enterprise and wide-area distributed computing environments," in *Lecture Notes in Computer Science: Euro-Par 2005*.

[25] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, 2007.

[26] ClusterResources, "Top500 Supercomputing Sites," June 2006, http://www.top500.org.

[27] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conf. Series; Scientific Discovery through Advanced Computing (SciDAC)*, vol. 78 (012022), 2007.

[28] USENIX, "The Computer Failure Data Repository (CFDR)," http://cfdr.usenix.org.

[29] SimPy, "Simulation in Python (SimPy) Homepage," http://simpy.sourceforge.net.

[30] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *Proc. of the ACM SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems*, 1995.

[31] K. F. Wong and M. Franklin, "Checkpointing in distributed systems," *Journal of Parallel & Distributed Systems*, vol. 35, no. 1, May 1996.

[32] G. P. Kavanaugh and W. H. Sanders, "Performance analysis of two time-based coordinated checkpointing protocols," in *Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, 1997.

[33] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, vol. 16, 1983.

[34] A. Oliner, R. Sahoo, J. Moreira, and M. Gupta, "Performance implications of periodic checkpointing on large-scale cluster systems," in *Parallel and Distributed Processing Symp.*, April 2005.

[35] O. O'Malley and A. C. Murthy, "Winning a 60 second dash with a yellow elephant," 2009. [Online]. Available: http://sortbenchmark.org/Yahoo2009.pdf