Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems

James S. Plank

Wael R. Elwasif

Department of Computer Science
University of Tennessee
Knoxville, TN 37996
[plank,elwasif]@cs.utk.edu

Appearing in *The 28th International Symposium on Fault-tolerant Computing*, Munich, June, 1998.

Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems

James S. Plank* Wael R. Elwasif
Department of Computer Science
University of Tennessee
Knoxville, TN 37996

Abstract

In the past twenty years, there has been a wealth of theoretical research on minimizing the expected running time of a program in the presence of failures by employing checkpointing and rollback recovery. In the same time period, there has been little experimental research to corroborate these results. In this paper, we study three separate projects that monitor failure in workstation networks. Our goals are twofold. The first is to see how these results correlate with the theoretical results, and the second is to assess their impact on strategies for checkpointing long-running computations on workstations and networks of workstations. A significant result of our work is that although the base assumptions of the theoretical research do not hold, many of the results are still applicable.

1 Introduction

The price and performance of desktop workstations has made them a viable platform for scientific computing. Combined with software platforms that allow workstations to cooperate using the paradigms of message-passing [17, 26], and shared memory [1], workstation networks have become powerful computational resources, rivaling supercomputers in their utility for scientific programming.

Traditionally, checkpointing and rollback recovery have been employed to provide fault-tolerance for long-running computations on all computing platforms (e.g. [4, 19, 21, 25, 27, 31]). By storing a checkpoint, a program limits the amount of re-execution necessary following a process or processor failure. In turn, this improves the program's running time in the presence of failures.

How often to checkpoint is a question of paramount practical importance. If one checkpoints too often, then the overhead of checkpointing may slow down the application program too much. However, if one checkpoints too infrequently, then the program may spend too much time re-executing code following failures. The problem of determining how often to checkpoint is called the *optimal checkpoint interval* problem. Its goal is to allow users of checkpointing systems to determine the frequency of checkpointing (the "interval") that minimizes the expected running time of the application in the presence of failures.

Determining the optimal checkpoint interval is a field of research with a rich history. The first papers on the topic appeared in the 1970's in the context of transaction processing systems [3, 11, 12, 33]. Later work has concentrated on real-time systems [10, 23], distributed systems [13, 29, 32] and more general frameworks for analysis [2, 8, 14, 15, 28, 30]. All of these papers derive analytical results concerning the performance of checkpointing systems in the presence of failures.

In relation to scientific computing on workstations and workstation networks, the above research has many implications. If the assumptions underlying the analytical results hold, then they may be used to:

- Predict a program's expected running time in the presence of failures, with or without checkpointing.
- Determine the optimal interval in which to checkpoint. This enables the program to minimize its expected running time in the presence of failures.
- Compare the performance of checkpointing algorithms.

In short, the results may be used to help make important decisions concerning the algorithms and runtime parameters in a checkpointing system.

All of the above papers require that the probability distribution of workstation failures is known. Typically, Poisson failure rates are assumed. In the papers where they are not assumed (e.g. [23, 28]), they are still

*`plank@cs.utk.edu`. This material is based upon work supported by the National Science Foundation under grants CCR-9409496, CDA-9529459 and CCR-9703390.

employed to exemplify the usage of the resulting equations. If workstation failures do not follow a Poisson model, the applicability of these results for scientific computing on workstations is brought into question.

There has been very little research that addresses the underlying assumptions of these results. In [6], the manifestation of software errors was shown *not* to follow Poisson processes. More significantly, in [16], Long *et al* performed a study monitoring the availability of machines on the Internet. In this study, they determine that the probability of machine failures following a Poisson model is extremely small. They do not attempt to characterize the failure model as following any standard probability distribution function.

Thus, the applicability of the theoretical equations is questionable, since their assumptions appear not to hold. The purpose of this paper is to address this problem and assess its practical implications on the users of checkpointing systems. We do this by analyzing machine availability data on three different networks of workstations, including Long's. We simulate the performance of long-running programs with and without checkpointing on these networks, and compare the simulated results to the theoretical predictions.

A significant result of this paper is that although the failure model of all three networks is decidedly not governed by Poisson processes, to a first order approximation many of the theoretical results still hold. Thus, in the absence of more data than the MTTF of a machine, one may determine a checkpoint interval that is not optimal, but reasonably close.

2 Outline

The outline of this paper is as follows. In Section 3, we state significant results from research on the optimal checkpoint interval. If failures follow a Poisson distribution, then the results cited in this section are extremely useful for determining runtime parameters for checkpointing, and for comparing checkpointing algorithms.

In Section 4, we describe the three sets of data that we use in this study. Each set contains longitudinal failure information for a collection of workstations over a period of six months or greater. In each data set, we can state with high confidence that failures do *not* follow a Poisson distribution.

In the remainder of the paper, we use the data from Section 4 to run simulations of checkpointing systems. With these simulations, we may determine the performance of checkpointing with any given parameters (e.g. checkpoint interval, overhead, etc). We use the simulations to assess how well the equations from Section 3 predict actual checkpointing performance. We conclude with recommendations on selecting parameters and algorithms for checkpointing that minimize the expected running time of long-running computations.

3 Results from Research on the Optimal Checkpoint Interval

In the literature cited in Section 1, there are many useful equations concerning the performance of checkpointing systems. We divide them into four categories:

1. **Predicting the performance of a program without checkpointing**. Without checkpointing, one runs a program and hopes that it completes before the machine on which it is running fails. If the machine does fail, then the program must be started anew when the machine becomes functional.

2. **Predicting the performance of a program with checkpointing**. Here the program periodically stores checkpoints of its execution state. If the machine fails, then the program recovers to the state of the last stored checkpoint.

3. **Predicting the optimal checkpoint interval**. This is the frequency of checkpointing that minimizes the program's expected running time in the presence of failures.

4. **Predicting the failure rate of parallel systems**. Equations in the above three categories have all been derived for uniprocessor systems. In certain cases, one can treat a parallel checkpointing system like a uniprocessor system with a slightly different failure model. In order to use the equations in the above three categories, the failure rate of the parallel system must be predicted from the uniprocessor failure rate. This prediction is the subject of this category.

In the equations that follow, we employ the following nomenclature (mostly borrowed from Vaidya [30]):

Failure-free running time (F). This is the running time of an application with no checkpointing on a machine that does not fail.

Average checkpoint overhead (C). Checkpoint overhead is the amount of time added to the application in a failure-free run as the result of checkpointing. C represents the average overhead per checkpoint.

Checkpoint latency (L). Latency is defined to be the time between when a checkpoint is initiated, and when it may be used to recover from a failure. If the application is halted while checkpointing, then the latency typically equals the overhead. However, certain optimizations such as *forked* checkpointing decrease overhead drastically while slightly increasing latency (for a discussion of this, see [30]).

Recovery time (R). This is the time that it takes the system to restore a checkpointed state following the detection of a failure and reconfiguration. Note that R does not take into account the down time of a system or the re-execution time of the application. It is simply a measure of how long it takes the system to restore itself from a checkpoint. Typically, R and L have similar values.

Down time (D). This is the average time following a failure before the system becomes functional.

Checkpoint interval (I, T). When an application is checkpointing periodically, the frequency of checkpointing is governed by the checkpoint interval. There are two ways to specify the checkpoint interval. The first, I, is the duration between the start of one checkpoint and the start of the next checkpoint. The second, T, is defined to be $I - C$. If latency is equal to overhead, then T is the time between the end of one checkpoint and the beginning of the next checkpoint.

Some checkpointing systems (e.g. [21, 19]) require the user to specify I, while others (e.g. [31]) require the user to specify T. When optimizations such as *forked* checkpointing are used, and $L \gg C$, I is the more natural specification. However, all theoretical research on the optimal checkpoint interval assumes that T is specified.

The difference between specifying I and T has a subtle impact on performance. If I is specified, then the interval between the beginning of the program and the start of the first checkpoint is I. If T is specified, then it is T. Similarly, if I is specified, then the interval between the restoration of a checkpoint and the beginning of the next checkpoint is I. If T is specified, then it is T. Thus, if one checkpointing system requires the user to specify I, and another requires the user to specify T, then even if all other parameters (e.g. C, L, R, etc) are the same, and even if $T = I - C$, performance of the two systems may differ. If $I \gg C$, this difference will be very slight, but if I is close to C and the failure rate is high, the difference may be significant. In practice it is true that $I \gg C$, and therefore we assume that conclusions that hold for I will also hold for $T = I - C$.

In all checkpointing systems, I must be greater than L. Otherwise, one checkpoint will not complete before the next one begins.

Expected running time (E_I, E_T). E_I/E_T is the expected running time of the application with checkpointing in the presence of failures. The checkpoint interval is either I, or T, depending on which interval specification method the checkpointing system requires.

Expected running time with no checkpointing (E_F). If the checkpoint interval is F (by either specification method), then the program will never checkpoint. If a failure occurs, then the application must restart from the beginning. Therefore E_F is the expected running time presence of failures when there is no checkpointing.

Optimal checkpoint interval (I_{opt}, T_{opt}). I_{opt} is the value of I that minimizes E_I. T_{opt} is the value of T that minimizes E_T. If $I \gg C$, then $I_{opt} \approx T_{opt} + C$.

Failure rate (λ). This is the average number of failures per unit uptime per machine [30]. We view machines has having alternating uptimes and downtimes. Each uptime interval is called a TTF interval, and each downtime interval is called a TTR interval. If the mean TTF interval in a collection of machines is *MTTF*, then $\lambda = 1/MTTF$.

If λ is a random variable following a Poisson distribution (i.e. λ is governed by a Poisson process), then the following results hold.

3.1 Predicting the performance of a program without checkpointing

The equation for predicting E_F was first specified by Duda [8]:

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

In this equation, it is assumed that the down time is zero. To include non-zero down times, this equation must be multiplied by $e^{\lambda D}$:

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

This assumes that $1/D$ is also governed by a Poisson process.

3.2 Predicting the performance of a program with checkpointing

To calculate E_T, Vaidya provides the following equation [30]:

$$E_T = \left(\frac{F}{T}\right)\frac{e^{\lambda(D+L-C+R)}\left(e^{\lambda(T+C)} - 1\right)}{\lambda} \tag{3}$$

Note that E_F may be derived from this when $T = F$ and $C = R = L = 0$.

3.3 Predicting the optimal checkpoint interval

An approximation to T_{opt} was first derived by Young [33]:

$$T_{opt} = \sqrt{2C/\lambda} \tag{4}$$

We refer to this as *Young's approximation.* In this approximation, it is assumed that $C = L = R$. Later papers provide refinements to Young's approximation. Recently, Vaidya has provided the following equation for T_{opt} [30]:

$$e^{\lambda(T_{opt}+C)}(1 - \lambda T_{opt}) \quad = \quad 1, \quad T_{opt} \neq 0 \qquad (5)$$

We refer to this as *Vaidya's approximation.* It is important to note that although Vaidya includes overhead, latency and recovery time in his model, the equation for T_{opt} depends only on the overhead and the failure rate.

3.4 The failure rate of parallel systems

There is less theoretical research on checkpointing performance in parallel systems. A straightforward approach is to assume that N processors are cooperating to run a parallel application, and periodically they coordinate to take checkpoints of the global system state. These are called *coordinated checkpoints.* If any processor fails, then the rest of the processors stop the computation, and wait for the failing processor to become functional. When that happens, all processors roll back to the stored checkpoint. For a thorough discussion of coordinated checkpointing, see the survey paper by Elnozahy *et at* [9].

In this scenario, we can view the parallel system as being in one of two states: *up* if all N processors are functional, and *down* if less than N processors are functional. We define λ_N to be the reciprocal of the average uptime. If we then use λ_N in place of λ, the above equations may be employed to predict the performance of checkpointing.

If processor failures are independent and they all follow Poisson distributions, then:

$$\lambda_N \quad = \quad N\lambda \qquad (6)$$

This equation is employed in all performance predictions of coordinated checkpointing (e.g. [13, 29, 32]).

4 Data Collection

To assess the applicability of the above equations, we obtained collections of failure data for three separate workstation networks. In each collection, a set of workstations was monitored for a period of at least six months. For each workstation, the data records the periods when the machine was functional. We assume that between functional periods, the machine is in a failure state. The granularity of the data is seconds, although the accuracy, as described below, is on the order of minutes to hours. We describe each data set below.

4.1 LONG – Machines on the Internet

The first set of data was collected by Long *et al* between July, 1994 and May, 1995. They wrote a program called "the tattler," which periodically queries a set of workstations on the Internet to determine their uptime intervals. To remain unaffected by network partitions, the tattler is replicated at many sites, and the data is merged to provide a unified view of the workstations in question. A description of the tattler and an assessment of the failure data appears in [16]. In that paper, they report data from 1139 hosts distributed throughout the world.

We obtained their data, and culled the number of hosts to 993, removing machines that reported erroneous (e.g. negative) uptimes, and machines that had less than 50 percent availability, since it is unlikely that such machines would be used for scientific programming. It is for this reason that our TTF/TTR data looks slightly different than in [16].

It is obvious that this data collection method may not tell the whole story concerning a machine's failures. For example, the tattler reports whether a machine is up, and not necessarily if it is usable. However, as a collection of a wide variety of geographically distributed machines, the **LONG** collection is extremely useful.

4.2 PRINCETON – A Local Network

The **PRINCETON** data set contains failure information for a collection of sixteen DEC Alpha workstations in the Department of Computer Science at Princeton University. Some of machines are owned by individuals in the department, and are sitting on their desks. The others are general-purpose workstations for any member of the department who needs them. Although the size of this network is much smaller than **LONG**, it represents a typical local cluster of homogeneous processors that is often used for parallel computation. The **PRINCETON** machines are not rebooted or brought down for backups on any regular schedule. When they fail, it is typically not planned.

The failure data for **PRINCETON** was collected between January and July, 1996. The method of collection differed from the **LONG** method. Instead, we used a program called **ltest** whose job was to "live" on each target machine and to recognize failures. **Ltest** runs in the background on each machine, probing the machine every ten minutes to ensure that it is alive. It makes use of the **cron** daemon on each machine to spawn additional copies of itself every hour, four hours, eight hours, twelve hours, and 24 hours, so that if the main **ltest** program dies, due to machine or process failure, it gets restarted by the **cron** daemon.

Network	**LONG**	**PRINCETON**	**CETUS**
# of machines	993	16	31
# of TTF intervals	10958	79	1898
# of TTR intervals	9965	63	1867
MTTF (days)	13.306	32.715	3.207
MTTR (days)	1.497	1.303	0.245
Availability	0.899	0.962	0.929

Table 1: Basic characteristics of the data sets

Ltest therefore measures a slightly different class of failures than the tattler. For example, if the system administrator or the owner of the machine decides to kill all processes in the system, **ltest** will detect this as a failure, while it goes unnoticed by the tattler. However, **ltest** only detects that a machine is functional when it (**ltest**) is running. Therefore, the time between a machine's restoration from a failure and when **ltest** gets initiated by the **cron** daemon will be considered downtime by **ltest**, but uptime by the tattler.

4.3 CETUS – A Local Network

The **CETUS** data set contains failure information, collected by **ltest**, for the "Cetus lab" in the Department of Computer Science at the University of Tennessee. The Cetus lab is a collection of thirty-one Sun Sparc IPX workstations connected by a local-area network. The machines are general-purpose workstations for use by any member of the department who needs them. They are a popular computing platform for running parallel scientific applications. One big difference between the **CETUS** machines and the **PRINCETON** machines is that the **CETUS** machines may be reserved at night for exclusive use by researchers conducting timing tests. When the reservation begins, the machines are rebooted, their **cron** daemons are disabled, and logins are refused for any user but the one with the reservation. Thus, while the tattler would report uptimes for reserved times, **ltest** classifies them as down, since the machines are unavailable at that time. The **CETUS** data was collected from December, 1995 to July, 1996.

The **CETUS** and **PRINCETON** data sets are included as opposite ends of a spectrum. Both are homogeneous, local-area networks of workstations, but the **PRINCETON** machines fail infrequently, while the **CETUS** machines fail frequently and at quasi-regular intervals. We have no knowledge about the usage of the machines in the **LONG** data set. In looking at the failure data, it is clear that some follow a daily rebooting schedule, while others fail at unpredictable times.

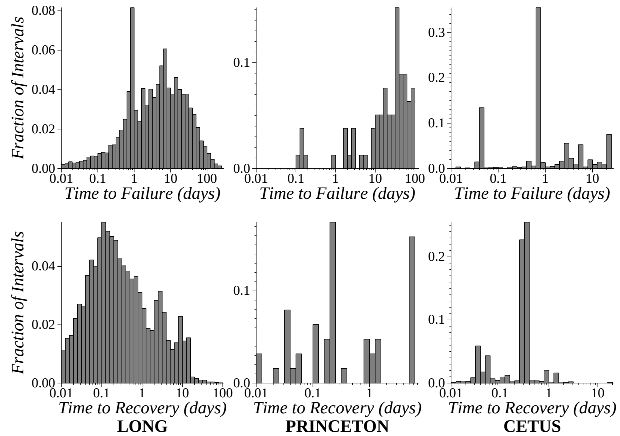

Figure 1: TTF and TTR distributions of the data sets

5 Basic Characteristics of the Data

The basic characteristics of the three data sets are in Table 1 and Figure 1. In this table, we use the same definition of availability $\left(\frac{\text{MTTF}}{\text{MTTF+MTTR}}\right)$ as Long [16].

As expected, the **PRINCETON** machines have the longest mean time to failure, and the **CETUS** machines the shortest. As the TTF/TTR interval distribution graphs show, the three networks have greatly varying distributions. The **LONG** data shows a variety of TTF intervals; however there is a distinct spike at just under one day. Long guesses that this is due to a number of machine owners who reboot their machines at the end of the day [16].

Nearly half of the **PRINCETON** TTF intervals are longer than a month, and 75% are longer than ten days. 80% of the TTR intervals are under one day.

In contrast, 70% of the **CETUS** TTF intervals are less than a day, with one value, roughly 15.5 hours, accounting for over 30% of the intervals. Similarly, over 40% of the TTR intervals are between 8 and 9 hours. Since the lab reservations are for 8.5-hour periods, the TTF and TTR distributions seem quite reasonable.

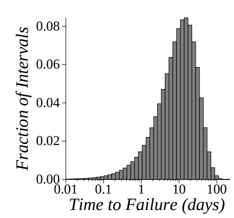

Figure 2: An exponential TTF distribution with a MTTF of 13.306 days

If workstation failures are governed by a Poisson process, then the TTF intervals should be distributed exponentially. This is a necessary but not sufficient condition. For reference, an exponential distribution with a MTTF of 13.306 days is shown in Figure 2. While one cannot rule out a set of observed data being governed by en exponential distribution, there are standard tests by which one may state with high or low confidence whether a set of data is exponentially distributed. Long performs one such test on his data to determine that the probability that the intervals are distributed exponentially is vanishingly small. Likewise, we performed Q-Q tests [7] on all three sets of data using the SPSS software package [24], and reached the same conclusion.

6 Simulation of Checkpointing and Rollback Recovery

To assess the applicability of the equations presented in section 3, we wrote a simulator to calculate expected performance of long-running programs with periodic checkpointing and rollback recovery. As input, the simulator takes one of the above data sets and the running time parameters of a long-running program X. These parameters are F, C, L, R and I. The simulator then picks a machine M in the data set, and a starting time t when the machine is first up. Now the simulator simulates running program X on machine M starting at time t, with checkpointing and rollback recovery.

Program X takes a checkpoint every I seconds while M is up. Each checkpoint requires C seconds of processing, and is not committed until L seconds after it starts. During the first I seconds, the simulator assumes that I seconds of X's computation has completed. Between checkpoints, $I - C$ seconds have completed. Whenever a failure occurs, the program cannot resume until the machine is up again. At that time, the simulator waits R seconds to roll back to the most recently committed checkpoint. Note that failures can occur during checkpointing and recovery because the rollback point is always to the last *committed* checkpoint. The program then resumes, and checkpoints are taken every I seconds again. When program X gets F seconds of running time, it finishes, giving us a data point for its running time, $E_{(I,M,t)}$.

The simulator calculates $E_{(I,M,t)}$ with a starting time t for every hour that M is up. In other words, if the simulator has just calculated $E_{(I,M,t)}$, then it next calculates $E_{(I,M,t+1h)}$. If M is in a failed state at $t + 1h$, then it calculates $E_{(I,M,t')}$ where t' is the first up time after $t + 1h$. At some point, t becomes too large and program X will not finish before we run out of data for machine M. At that point, we are finished calculating data points for machine M. We calculate

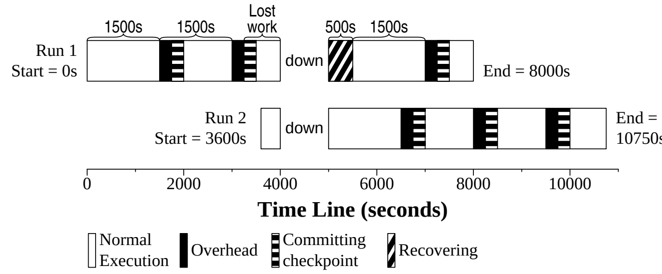

Figure 3: Example program simulation,

data points for each machine M in the data set, and then average all the values of $E_{(I,M,t)}$ to get E_I, the expected running time in the presence of failures with checkpointing and rollback recovery.

For example, suppose our data set consists of one machine with two uptime intervals, $(0s, 4000s)$ and $(5000s, 11000s)$, and we wish to simulate a long-running program X with $F = 5000s$, $C = 250s$, $L = R = 500s$, and $I = 1500s$. Figure 3 shows how the simulation proceeds, including the checkpoints, failures and recoveries. First, $E_{(I,M,0)}$ is calculated. Two checkpoints complete before M fails. When M comes back up, program X is restored from the second checkpoint, and takes one more checkpoint before completing. The running time for this program is $8000s$ (5000 execution, 750 checkpoint overhead, 500 recovery overhead, 750 lost work, 1000 downtime). Next, $E_{(I,M,3600)}$ is calculated. The program executes for $400s$ before the M fails. When it comes back up, program X must be restarted from the beginning. It takes three checkpoints before completing, and the total running time is $7150s$ (5000 execution, 750 checkpoint overhead, 0 recovery overhead, 400 lost work, 1000 downtime). Data point $E_{(I,M,7200)}$ is not calculated because program X cannot complete if it starts at $7200s$. Therefore, the expected time to completion of this program, E_I, is 7575 seconds.

7 Simulation Results

We ran many simulations on the three data sets, and compared the results to predictions using the analytic equations of Section 3. We present the results in the four prediction categories presented in Section 3.

7.1 Predicting the performance of a program without checkpointing

To predict E_F for a program, we use Eq. 2 from Section 3. Figure 4 displays the results of simulating programs with no checkpointing and varying F on each of the networks. Unsurprisingly, the **PRINCETON** network gives the fastest running times, and the **CETUS** network gives the slowest. Figure 4 also plots the

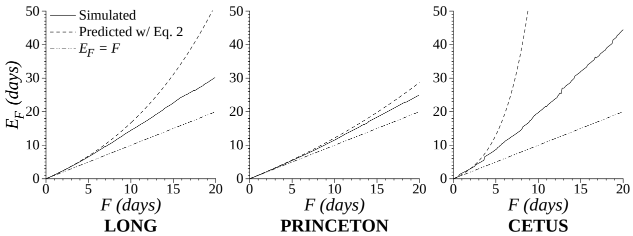

Figure 4: Expected running times in the presence of failures, E_F, as a function of failure-free running time.

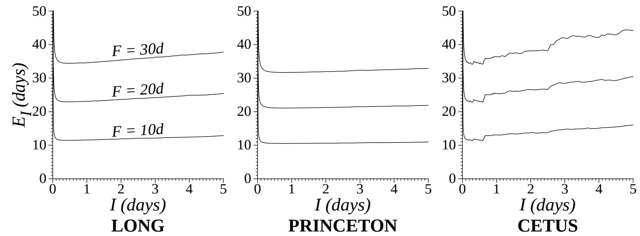

Figure 5: Effect of modifying I on the simulated value of E_I with $C = L = R = 10m$, $F = 10d, 20d, 30d$

predicted values of E_F using Eq. 2 from Section 3, and a line at $E_F = F$ for reference. In all networks, the predicted and simulated values show little difference for small values of F, but as F grows larger, the predicted values become much larger than the simulated values. The effect is more marked in the networks with higher failure rates.

7.2 Predicting the performance of a program with checkpointing

To predict the performance of a program with periodic checkpointing, we use Eq. 3 from Section 3. To measure how well Eq. 3 predicts the running time of programs that employ checkpointing, we ran simulations of programs that ran for 10, 20, and 30 days, had overheads between 1 and 60 minutes, and varied I from just greater than the overhead to ten days. Overheads as high as 3.5 minutes have been reported in real-life checkpointing applications [31]. We include overheads that are higher than this as a recognition of trends. It is not uncommon to find workstations with over 500 *MB* of memory that access network file systems at a bandwidth of approximately 0.2 *MB/s* (the same rate as [19, 31]). Therefore, although checkpoint overheads in the tens of minutes have not been seen heretofore, they are not out of the realm of possibility. Moreover, by including large checkpoint overheads, we may see certain trends in the data that might go unnoticed otherwise.

In these tests, we assume that $C = L = R$.

In Figure 5, we display the results for 10-minute overheads, and I varying from 15 minutes to 5 days. This Figure displays many features that are typical of all the tests. First, as anticipated, when I is too small, the overhead of checkpointing dominates the running time of the program. However, this effect decreases rapidly as I increases until the expected running time reaches some minimal value. This is where $I = I_{opt}$. As I takes on values greater than I_{opt}, E_I increases with a small slope, due to lost work resulting from failures.

The second feature is that in each graph, the shape of the curves is similar for each value of F. Thus, we can say that to a first degree approximation, the effect of varying I on the running time of the program is independent of F. Third, in all cases, I_{opt} is well less than 5 days: approximately 10 hours for **LONG**, 19 hours for **PRINCETON**, and 7 hours for **CETUS**.

The shape of the curve for the **CETUS** network is less smooth than for the others. This is a direct result of the TTF distribution. As displayed in Figurst 1, more than a third of the TTF intervals occur at roughly $15.5h$ (0.645 days). Therefore, it makes sense for there to be a sharp increase in E_I as I grows past $15.5h$. Similarly, one expects to see a smaller increase as I grows past $7.7h$. This effect is more pronounced in subsequent graphs.

For brevity, in the figures that follow we only plot values for $E = 30$ days, and we restrict the values of I to those that are near I_{opt}.

In Figure 6, we plot both simulated and predicted values of E_I for programs where $C = L = R = 1m$, $10m$, and $60m$. To predict E_I with Eq. 3, we assume $E_I = E_T$, where $T = I - C$. As anticipated, when the overhead of checkpointing is lower, the overall E_I is lowered for all values of I. The value of I_{opt} is also lower. A striking feature of all the graphs in Figure 6 is that the predicted values of E_I are quite close to the simulated values. This is especially true for I near I_{opt}, and even holds for the often-failing **CETUS** machines. From this, we draw the conclusion that for small values of I, Eq. 3 is a reasonable predictor of checkpointing performance.

7.3 The optimal checkpoint interval

Perhaps a more important equation from Section 3 is the determination of the optimal checkpoint interval. In this section we evaluate the utility of both Young's and Vaidya's approximations on the three data sets. For programs with running times of 30 days, we used our simulator to determine the optimal checkpoint interval I_{opt} for values of $C = L = R$ between ten seconds and one hour. We then compared these to values of I_{opt} as calculated by Young and Vaidya's approxi-

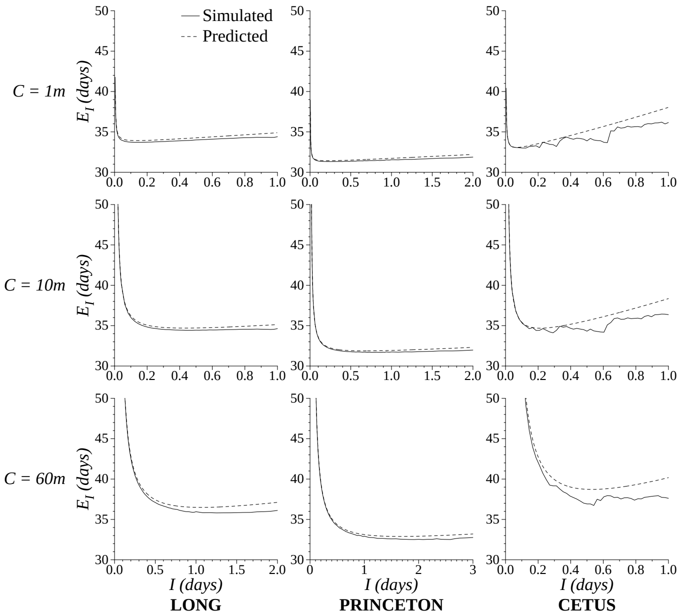

Figure 6: The effect of modifying I on E_I, simulated and predicted, for $F = 30d$, $C = L = R = 1m, 10m, 60m$.

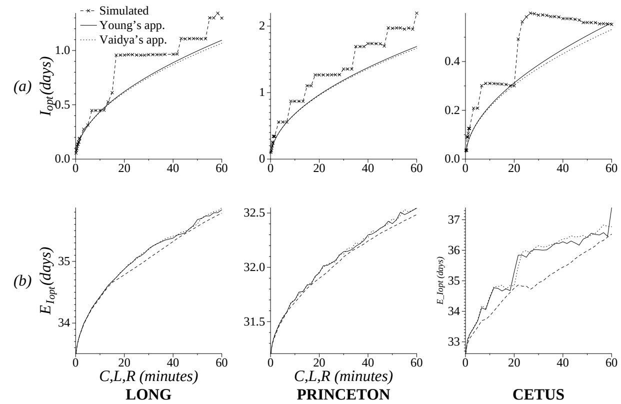

Figure 7: Comparing simulated and predicted values of I_{opt} for C, L, R between $10s$ and $1h$, $F = 30d$.

mations.

Figure 7 shows the results of these tests. In row (a), we plot all three values of I_{opt} as a function of C, L, R. In row (b), we used each value of I_{opt} — simulated, Young's and Vaidya's — and simulated the expected running time of the program with that as the checkpoint interval. These are plotted as a function of C, L, R. By definition, the simulated value of I_{opt} will yield the lowest expected running time.

There are several interesting results in Figure 7. First, the simulator's determination of I_{opt} is almost always greater than Vaidya's approximation. Moreover, it resembles a step function which changes "steps" whenever it approaches Vaidya's approximation. The "steps" are larger in the **LONG** and **CETUS** networks, and appear related to the fact that both of these data sets have a single TTF value which is disproportionately represented (roughly $0.98d$ in **LONG**, and $0.64d$ in the **CETUS** network). Therefore, if Young or Vaidya's approximation says, for example, that I_{opt} should be $0.25d$ in the **CETUS** network, a better choice should be approximately $0.32d$, because the same number of checkpoints (two) will be committed in the most frequent TTF interval, but more computation will be committed before the failure.

Figure 7(b) quantifies the performance penalty of choosing Young or Vaidya's approximation to I_{opt} instead of the simulated value. In the **LONG** and **PRINCETON** networks, the penalty is quite small – just a few hours in each case, which is less than 0.5% of the total running time of the application. The **CETUS** network shows worse performance penalties, with a maximum of 30 hours. The conclusions that we draw from this is that in the **LONG** and **PRINCETON** networks, both Young's and Vaidya's approximations achieve close to the optimal simulated performance on long-running programs. In the **CETUS** network, both approximations have points at which they penalize performance significantly (more than a day) over the optimal selection of the checkpoint interval.

Finally, the graphs in Figure 7(b) display how decreasing checkpoint overhead improves the performance of the application. For example, in the **LONG** network, lowering the overhead from one hour to 20 minutes improves the average running time by one day. Lowering the overhead from one hour to one minute improves the running time by two days.

7.4 Latency

Another result from Section 3 is that the optimal checkpoint interval is independent of the checkpoint latency. To test this, we performed three tests which fix the checkpoint overhead at $C = 1m$, $5m$ and $10m$, and vary the latency between C and $60m$. Due to space constraints, we omit the data, which may be found in [20]. The simulations show that to a first order approximation I_{opt} is independent of L. As L increases, the expected running time ($E_{I_{opt}}$) increases as well, although not to the same degree as when C is increased. This agrees with Vaidya's assertion that checkpoint latency has far less impact on the running time of a program than overhead [30].

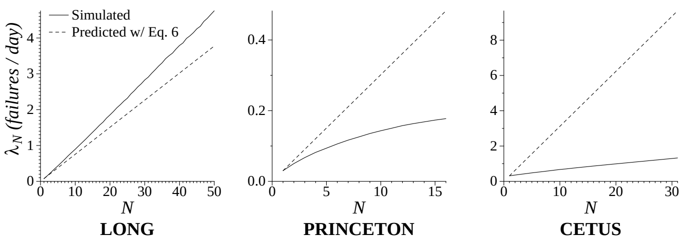

Figure 8: Determination of λ_N for each network.

7.5 The failure rate of parallel systems

We performed one final test to explore the validity of Eq. 6 from Section 3. Here we wrote a program that takes as input a data set (**LONG**, **PRINCETON** or **CETUS**) and a number of processors N, and calculates λ_N for the data set. It does this using a Monte Carlo simulation. The program runs for a given number of iterations, and during each iteration, it chooses a random set of N processors from the data set. It then calculates the TTF intervals for that data set, stipulating that an up state is when all N processors are functional, and a down state is when less than N processors are functional. The TTF intervals are averaged over all iterations, to yeild the MTTF. λ_N is then the inverse of the MTTF.

Figure 8 displays λ_N as calculated by our program (20,000 iterations per value of N). This is plotted as a function of N, and compared to $N\lambda$. The results are quite different for each network. In the **LONG** network, λ_N appears to grow linearly with N, but at a rate of $1.26N\lambda$, rather than $N\lambda$. In the **PRINCETON** and **CETUS** networks, λ_N grows much more slowly than $N\lambda$. This is to be expected because given the proximity of machines to each other within each network, it is unlikely that failures will be independent. For the **CETUS** network, the reservation schedule guarantees that processor failures are not independent, and that λ_N will be closer to λ than to $N\lambda$.

From this data, it is hard to draw general conclusions concerning the failure rate of parallel systems. Certainly assuming that $\lambda_N = N\lambda$ does not seem to be reasonable in any of the networks, with the possible exception of the **LONG** network. It remains to be seen what the implication of this is for predictions of parallel checkpointing performance (e.g. [13, 29, 32]).

8 Conclusions

We have used the results of three workstation monitoring projects to assess the applicability of theoretical equations concerning the performance of checkpointing. Since the equations assume that failure and recovery rates follow Poisson processes, and the actual failure and recovery rates did not follow Poisson processes, we expected the equations to have little applicability. However, there are several cases where the equations provide an excellent barometer of checkpointing performance. To summarize our findings:

- Eq. 2 is a poor predictor of the running time of a program without checkpointing. In particular, when failures play a significant role in the execution of the program, Eq. 2 overestimates the expected running time drastically.
- Eq. 3 provides a good approximation to the expected running time of a program for checkpoint intervals that are near the optimal interval.
- Both Young and Vaidya's approximations to T_{opt}/I_{opt} may be used without a huge performance penalty. In particular, these approximations for the **LONG** and **PRINCETON** networks penalized performance only by a few hours on a program that ran for 30 days.
- The optimal checkpoint interval appears to be independent of checkpoint latency. Moreover, to optimize the performance of checkpointing, decreasing overhead has more impact than decreasing latency.
- Little can be said about the rate to first failure, λ_N, in a N-processor parallel system except that it cannot always be assumed to equal $N\lambda$. In systems where machines are likely to fail in groups, like the **CETUS** network, $\lambda_N \ll N\lambda$.

As an additional note, our simulations show that given a program X, as long as I_{opt} is less than the running time F, then checkpointing is a useful feature. Therefore, even though we only show programs with long running times in our simulations, programs with running times under a day can also benefit from checkpointing.

8.1 Limitations and Future work

There are several limitations to this work. It is a subject of future work to address these limitations so that studies of this manner may have a broader impact.

We make no attempt to quantify the actual failure distribution of the three networks in terms of well known distributions. Nor do we attempt to characterize the mathematical properties of the distributions as they impact the equations of Section 3. Off-hand, we may state that the **PRINCETON** network appears more "Poisson-ish" than the other networks, which is reflected not only in its TTF distribution pattern, but also in the closer matching of simulated values to the equations of Section 3. However, it is a more interesting issue to provide a rigorous statistical analysis to the data.

Like [22], we have employed a simulator to draw conclusions that are only as good as the simulator's base assumptions. Thus, questions also may be raised concerning the validity of the simulations. For example, all uptimes are considered equivalent by the simulator, whereas in real life, machines undergo fluctuations in load and true availability [5, 18]. Further, the three networks studied may not be a true reflection of the "average" workstation network. Finally, the decision to treat reserved time in the **CETUS** network as downtime may deflate the true availability of the system. For example, a user might checkpoint his or her code just before the reserved time and restore it when the machines become available. We decided to make no assumptions on the users' level of sophistication, and therefore treated the reserved periods as downtime. In future work, we will treat the **CETUS** network in two ways – first as it is treated in this paper, and second as seen by the user who checkpoints before the reserved period starts.

There are several other avenues of future work in this area. The first is to encourage the collection and dissemination of a wider variety of failure data. Second is to consider factors such as CPU load and network performance in the data collection and simulation. Third is to consider parallel systems, and more advanced checkpointing algorithms than simple coordinated checkpointing.

9 Acknowledgments

The authors thank Darrell Long and Richard Golding for sharing their failure data, and Lee Hamner for writing **ltest** and monitoring the **PRINCETON** and **CETUS** networks. We also thank Geoff Abers and Adam Beguelin for letting us monitor their workstation networks, and Nitin Vaidya and Yi-Min Wang for answering questions. Finally, we thank the referees, especially "external referee #1" for their helpful comments and constructive suggestions.

10 References

[1] C. Amza *et al.* TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] B. Baccelli. Analysis of a service facility with periodic checkpointing. *Acta Informatica*, 15:67–81, 1981.

[3] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Trans. Comp.*, 21:546–556, June 1972.

[4] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing*, Nov. 1997.

[5] P. E. Chung *et al.* Checkpointing in CosMiC: a user-level process migration environment. In *Pac. Rim Int. Symp. on Fault-Tol. Systems*, Dec. 1997.

[6] L. H. Crow and N. D. Singpurwalla. An empirically developed fourier series model for describing software failures. *IEEE Trans. on Reliability*, R-33:176–183, June 1984.

[7] M. J. Crowder *et al. Statistical Analysis of Reliability Data*. Chapman & Hall, London, 1991.

[8] A. Duda. The effects of checkpointing on program execution time. *Inf. Proc. Letters*, 16:221–229, 1983.

[9] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Tech. Rep. CMU-CS-96-181, Carnegie Mellon Univ., Oct. 1996.

[10] R. Geist, R. Reynolds, and J. Westall. Selection of a checkpoint interval in a critical-task environment. *IEEE Trans. on Reliability*, 37:395—400, Oct 1988.

[11] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26:259–270, Apr. 1979.

[12] E. Gelenbe and D. Derochette. Performance of rollback recovery systems under intermittant failures. *Comm. of the ACM*, 21(6):493–499, June 1978.

[13] G. P. Kavanaugh and W. H. Sanders. Performance analysis of two time-based coordinated checkpointing protocols. In *1997 Pac. Rim Int. Symp. on Fault-Tol. Systems*, Dec. 1997.

[14] C. M. Krishna *et al.* Optimization criteria for checkpoint placement. *Comm. of the ACM*, 27:1008–1012, Oct. 1984.

[15] P. L'Ecuyer and J. Malenfant. Computing optimal checkpointing strategies for rollback and recovery systems. *IEEE Trans. on Comp.*, 37(4):491–496, Apr. 1988.

[16] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *14th Symp. on Rel. Dist. Sys.*, pages 2–9, Sep. 1995.

[17] Message Passing Interface Forum. MPI: A message-passing interface standard. *Int. Jour. of Supercomp. App.*, 8(3/4), 1994.

[18] M. W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Perf. Evaluation*, Aug. 1991.

[19] J. S. Plank *et al.* **Libckpt**: Transparent checkpointing under unix. In *Usenix Tech. Conf.*, pages 213–223, Jan. 1995.

[20] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. Tech. Rep. CS-97-379, Univ. of Tenn., Dec. 1997.

[21] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Par. & Dist. Tech.*, 2(2):62–67, 1994.

[22] B. Ramamurthy, S. J. Upadhyaya, and R. K. Iyer. An object-oriented testbed for the evaluation of checkpointing and recovery systems. In *27th Int. Symp. on Fault-Tolerant Comp.*, 1997.

[23] K. G. Shin *et al.* Optimal checkpointing of real-time tasks. *IEEE Trans. on Comp.*, 36(11):1328–1341, Nov. 1987.

[24] SPSS, Inc. SPSS for Windows. Release 7.5, see `http://www.spss.com`, 1996.

[25] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th Int. Par. Proc. Symp.*, Apr. 1996.

[26] V. S. Sunderam *et al.* The PVM concurrent computing system: Evolution, experiences, and trends. *Journal of Par. and Dist. Comp.*, 1993.

[27] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, Feb. 1995.

[28] S. Toueg and Ö. Babaoglu. On the optimum checkpoint selection problem. *SIAM Journal on Comp.*, 13:630–649, Aug. 1984.

[29] N. H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Conf.*, May 1995.

[30] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. on Comp.*, 46(8):942–947, Aug. 1997.

[31] Y-M. Wang *et al.* Checkpointing and its applications. In *25th Int. Symp. on Fault-Tolerant Comp.*, pages 22–31, June 1995.

[32] K. Wong and M. Franklin. Checkpointing in distributed systems. *Journal of Par. & Distr. Sys.*, 35(1), May 1996.

[33] J. S. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, Sep. 1974.