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

December 12, 1997

Technical Report UT-CS-97-379
University of Tennessee

Available via ftp to cs.utk.edu in pub/plank/papers/CS-97-379.ps.Z
Or on the web at http://www.cs.utk.edu/~plank/plank/papers/CS-97-379.html

Submitted for publication. See the web page for futher information.

Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems

James S. Plank*   Wael R. Elwasif

December 12, 1997

University of Tennessee Technical Report UT-CS-97-379.
Submitted for publication. For publication status of this work, please see:
`http://www.cs.utk.edu/~plank/plank/papers/CS-97-379.html`

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 the results of 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 surprising 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 [SGDM93, Mes94], and shared memory [ACD$^+$96], 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. [LS92, PBKL95, HKW95, Ste96, EJZ92, CPL97]). 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.

*`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.

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 [CR72, You74, GD78, Gel79]. Later work has concentrated on real-time systems [SLL87, GRW88], distributed systems [Vai95, WF96, KS97] and more general frameworks for analysis [Bac81, Dud83, TB84, KSL84, LM88, Vai97]. 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 interval 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. [TB84, SLL87]), they are still 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 [CS84], the manifestation of software errors was shown *not* to follow Poisson processes. More significantly, in [LMG95], 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, there is a contradiction between the theoretical results and experimental observations. The purpose of this paper is to address this contradiction 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 programs with and without checkpointing on these networks, and compare the simulated results to the theoretical projections.

A surprising result of this paper is that although the failure model of all three networks is decidedly not governed by Poisson processes, to a first 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 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**. With checkpointing, 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 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 [Vai97]):

C – **Average checkpoint overhead**. 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.

L – **Checkpoint latency**. 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, please see [Vai97]).

R – **Recovery time**. This is the time that it takes the system to restore itself to a checkpointed state once it has become functional. 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.

D – **Down time**. This is the average time following a failure before the system becomes functional.

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

I, T – **Checkpoint interval**. When an application is checkpointing periodically, the frequency of checkpointing is governed by the checkpoint interval. Unfortunately, there are two ways to specify the checkpoint interval. The first, I, is the 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. [PL94, PBKL95]) require the user to specify I, while others (e.g. [WHV$^+$95]) 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 subtle implications 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 recovery from 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 $I = T + C$, performance of the two systems may differ. If $I \gg C$, this difference should be very slight, but if I is close to C and the failure rate is high, the difference may be significant.

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

E_I, E_T – **Expected running time**. 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.

E_F – **Expected running time with no checkpointing**. 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.

I_{opt}, T_{opt} – **Optimal checkpoint interval**. 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 $I_{opt} \approx T_{opt} + C$.

λ – **Failure rate**. This is the average number of failures per unit time. If the mean time before failure of a system 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 [Dud83]:

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

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 [Vai97]:

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

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 [You74]:

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

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} [Vai97]:

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

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 Predicting 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.* For a thorough discussion of coordinated checkpointing, please see the survey paper by Elnozahy *et at* [EJW96]. If any processor fails, then all the processors halt. When N processors again become functional, they all roll back to the stored checkpoint. In this scenario, the above equations may be employed to predict the performance of checkpointing, as long as we use $\lambda = \lambda_N$, which is the rate of the *first* processor in the collection failing.

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

$$\lambda_N \quad = \quad N\lambda_1. \tag{6}$$

This fact is employed in all performance predictions of parallel checkpointing systems (e.g. [Vai95, WF96, KS97]).

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 – Random 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 up-time 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 [LMG95]. 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 times vastly out of the July 1994 to May 1995 interval, 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 [LMG95].

It is obvious that this data collection method may not tell the whole story concerning a machine's failures. For example, if a machine fails twice between two queries of the tattler, then the corresponding interval between the failures will be lost. Further, this data 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 – Network of DEC Alpha Workstations

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.

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 down time by **ltest**, but up time by the tattler.

4.3 CETUS – Network of Sun Sparc Workstations

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 and on no set schedule, 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.

5 Basic Characteristics of the Data Sets

The basic characteristics of the three data sets are in Table 1. 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 and TTR 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 [LMG95].

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 a third of the intervals. Similarly, over 40% of the TTR intervals are between eight and nine hours. Since the lab reservations are for 8.5-hour periods, the TTF and TTR distributions seem quite reasonable.

In order for workstation failures to be governed by a Poisson process, the TTF intervals should be distributed exponentially. For reference, an exponential distribution with a MTTF of 13.306 days is shown in Figure 1. 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

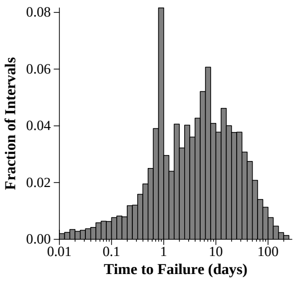
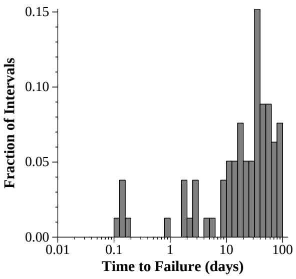
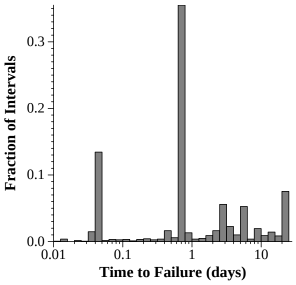
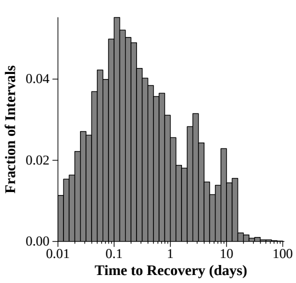
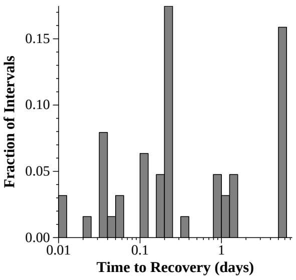
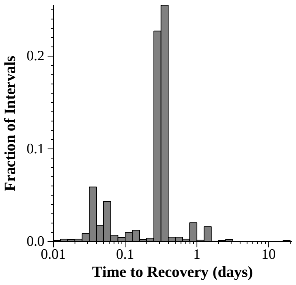
Network	**LONG**	**PRINCETON**	**CETUS**
Number of machines	993	16	31
Number of TTF intervals	10958	79	1898
Number of TTR intervals	9965	63	1867
Mean TTF interval (days)	13.306	32.715	3.207
Mean TTR interval (days)	1.497	1.303	0.245
Availability	0.899	0.962	0.929
TTF interval distribution			
TTR interval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

Table 1: Basic characteristics of the data sets

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 [CKSS91] on all three sets of data using the SPSS software package [SPS96], and reached the same conclusion.

6 Simulation of Checkpointing and Rollback Recovery

To assess the applicability of the theoretical results presented in section 3, we wrote a program to simulate the expected performance of long-running programs with periodic checkpointing and rollback recovery. As input, the simulator takes one of the above data sets, plus F, C, L, R, and I. It then does the following for each machine in the data set. It picks a starting time when the machine is up, and simulates running the program on that machine at that starting time. Every I seconds, the program takes a checkpoint, which requires C seconds of processing,

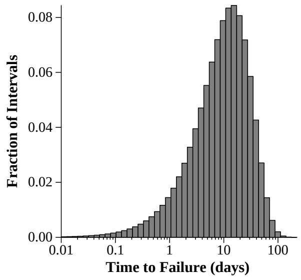


Figure 1: An exponential TTF distribution with a MTTF of 13.306 days

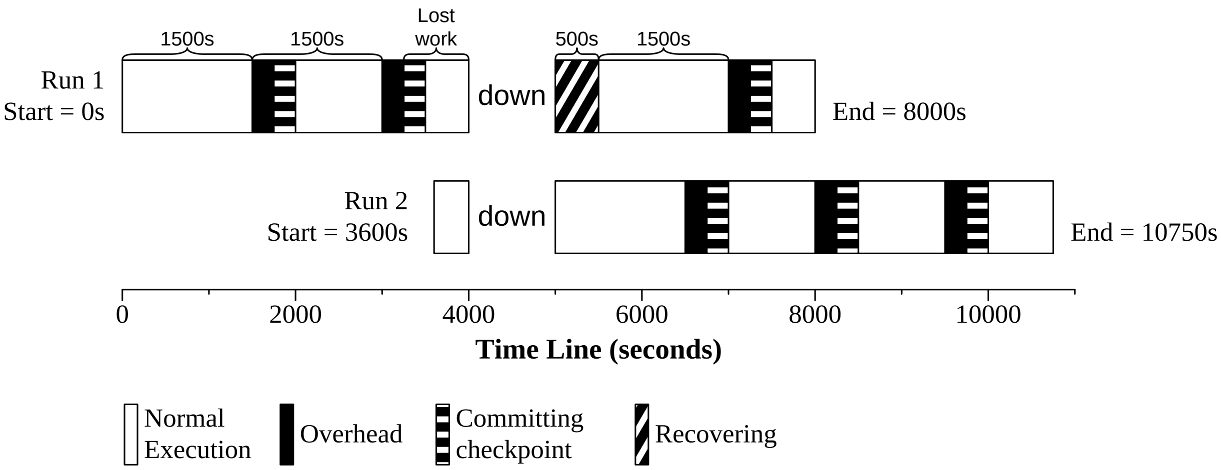


Figure 2: Example program simulation,

and is not committed until L seconds after it started. When the program gets F seconds of running time (this is in addition to the checkpoint overhead), it finishes. If the machine fails before the program finishes, then the program must wait until the machine becomes functional again, and then it takes R seconds to recover from the most recently committed checkpoint (zero seconds if there is no committed checkpoint). It then continues processing until the program completes, or until the next failure occurs.

One such simulation is calculated for each hour of the machine's lifetime. Specifically, if the last simulation started at time t, then the next simulation starts at $t + 1$h. If the machine is down at $t + 1$h, then the next simulation starts at the next uptime after $t + 1$h. Simulations are run in this manner until we get to the end of the data set. The running times are then averaged to yield an expected running time for that program on all machines in the collection.

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 program with F = 5000s, C = 250s, L = R = 500s, and I = 1500s. The graph in Figure 2 shows how each simulation proceeds, including the checkpoints, failures and recoveries. In the first run, two checkpoints are completed before the machine fails. After the machine comes back up, the program is

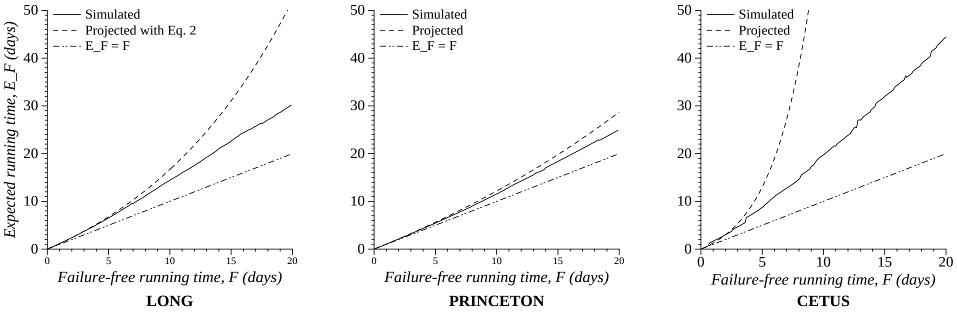


Figure 3: Running times in the presence of failure

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 computation, 1000 downtime). In the second run, the program executes for 400s before the machine fails. When it comes back up, the program 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 computation, 1000 downtime). There is no third run, because the program cannot complete if it starts at 7200s. Therefore, the expected time to completion of this program on this machine is 7575 seconds.

7 Simulation Results

We ran many simulations on the three data sets, and compared the results to 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 3 displays the results of simulating programs with no checkpointing and varying running times on each of the networks. Unsurprisingly, the **PRINCETON** network gives the fastest running times, and the **CETUS** network gives the slowest. Figure 3 also plots the projected values of E_F using Eq. 2 from Section 3, and a line at $E_F = F$ for reference. In all networks, the projected and simulated values start off the same, but as the running time increases, the projected values rise much more rapidly than the simulated values. The effect is more marked in the networks with higher failure rates.

This suggests that as E_F deviates from F, Eq. 2 loses its utility.

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

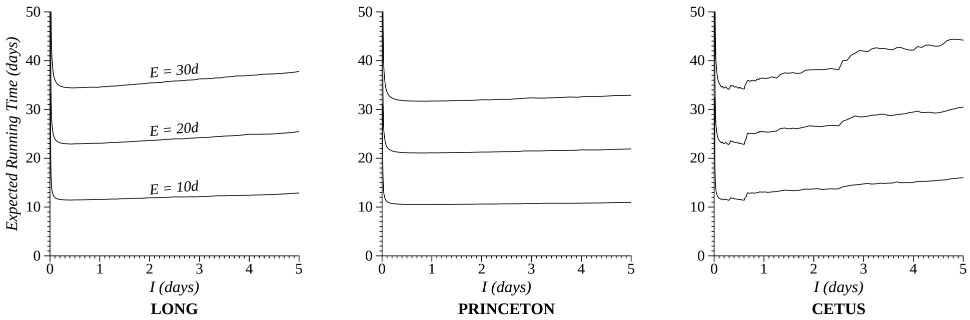


Figure 4: Effect of modifying I on programs with $C = L = R = 10m$, $F = 10d, 20d, 30d$

programs that ran for 10, 20, and 30 days, had overheads of 1, 10, and 60 minutes, and varied I from just greater than the overhead to ten days. These overheads are consistent with those reported in real-life checkpointing applications [WHV$^+$95]. In these tests, we assume that $C = L = R$.

We display the results for 10-minute overheads, and I varying from 15 minutes to 5 days in Figure 4. 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 $I = I_{opt}$. After that, the running time increases more slowly with I as lost work due to failures because significant.

Second, for each network the shape of each curve is quite similar. Thus, we can say that to a first approximation, the effect of varying I on the running time of the program is independent of F. Third, in all cases, the minimum running time is at a value of I that is well less than 5 days.

The shape of the curve for the **CETUS** network is more jagged than for the others. This is a direct result of the TTF distribution. As displayed in Table 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 5, we plot both simulated and projected values of E_I for programs where $C = L = R =$ 1m, 10m, and 60m. To project 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. Similarly, when the overhead is higher, E_I is higher for all values of I and the value of I_{opt} is higher. A striking feature of all the graphs in Figure 5 is that the projected 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 surprising conclusion that for small values of I, Eq. 3 is a reasonable predictor of checkpointing performance.

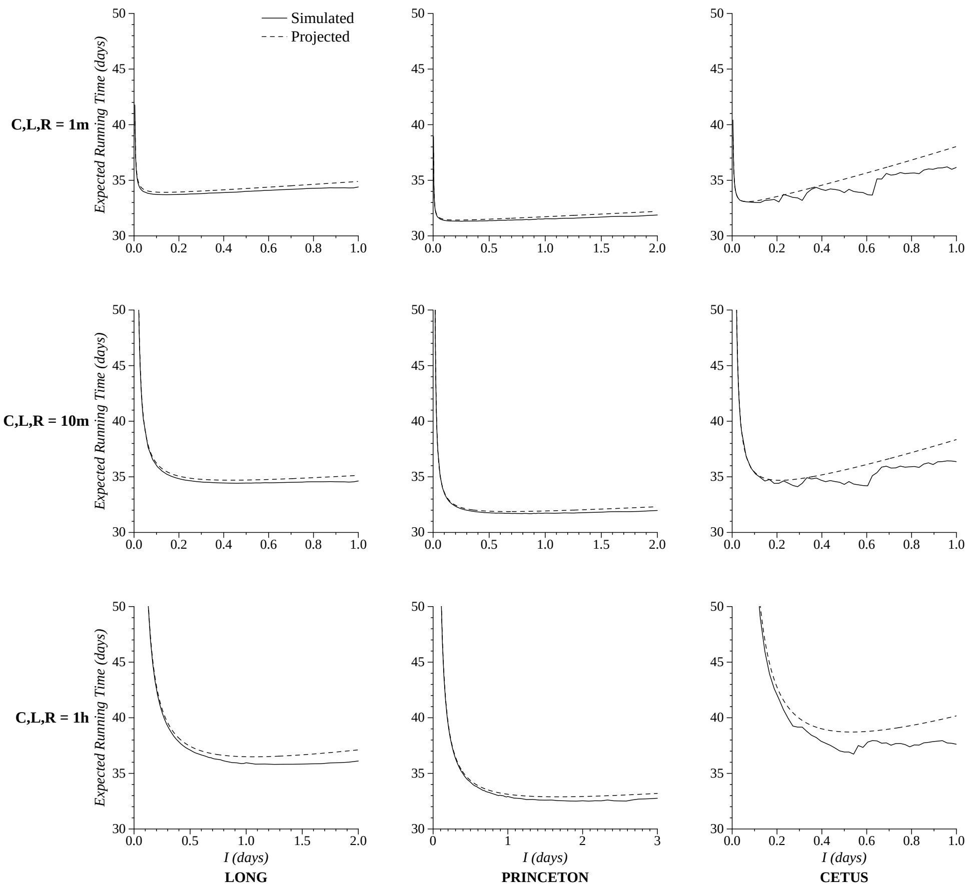


Figure 5: The effect of modifying I on E_I, simulated and projected, for $F = 30d$, $C = L = R = 1m, 10m, 1h$.

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 approximation and Vaidya's approximation 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 approximations.

Figure 6 shows the results of these tests. The top row of graphs plots all three values of I_{opt} as a function of checkpoint overhead. For the bottom two rows of graphs, 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. In the middle row of graphs, we plot the expected running times as a function of checkpoint overhead. By definition, the simulated value of I_{opt} will yield the lowest expected running time. The bottom row of graphs prints the performance penalty in using Young or Vaidya's approximation to I_{opt} instead of using the simulated value.

There are several interesting results in Figure 6. First, if we consider the simulator's determination of I_{opt} to

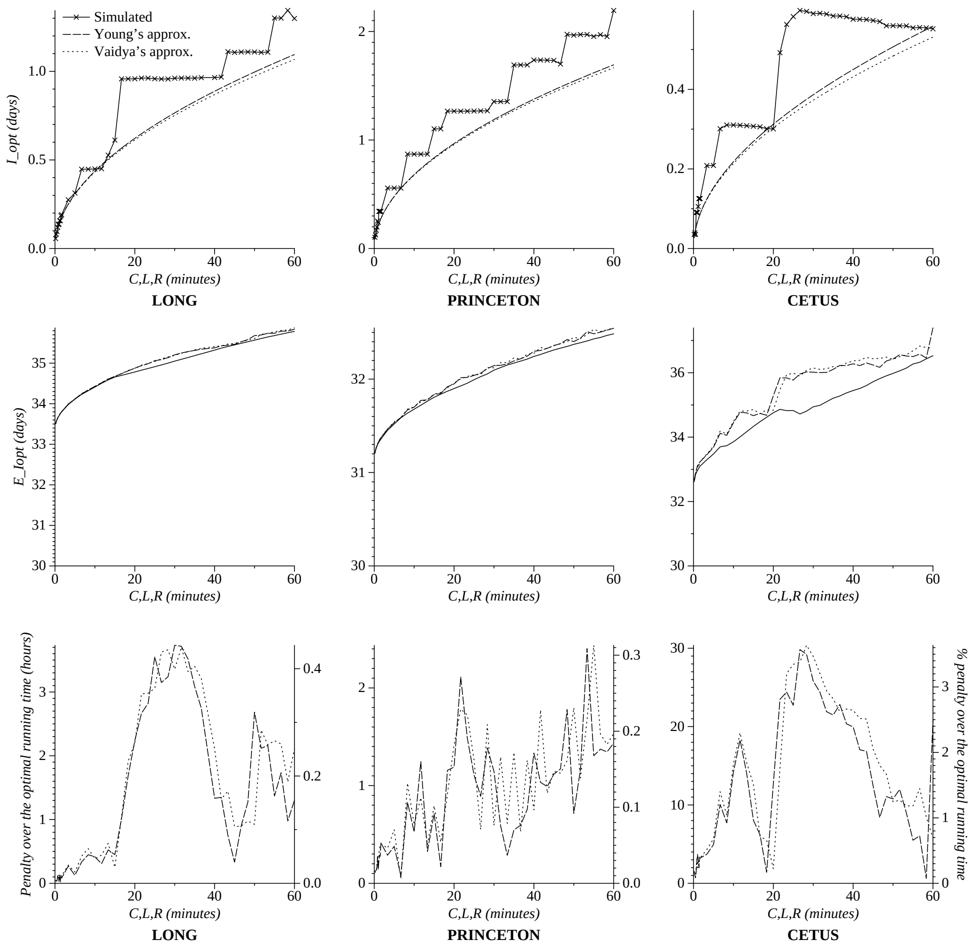


Figure 6: Comparing simulated and projected values of I_{opt} for $C = L = R$ between 1m and 1h, $F = 30$d.

be its true value, then we note that the true value 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, when Young or Vaidya's approximation says, for example, that I_{opt} should be 0.25d in the **CETUS** network, I_{opt} should actually be somewhere near 0.32d, because the same number of checkpoints (two) will be committed in the most frequent TTF interval, and less computation will be lost when the TTF interval is 0.64d, and the checkpoint interval is just less than 0.32d.

is 0.32d. Obviously, there is more going on in Figure 6 than can be explained by this effect, but the disproportionately represented TTF values in **CETUS** and **LONG** seem to have an impact on I_{opt}.

The second and third rows of graphs quantify the performance penalty of choosing Young or Vaidya's approximation to I_{opt} instead of the correct 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. In the

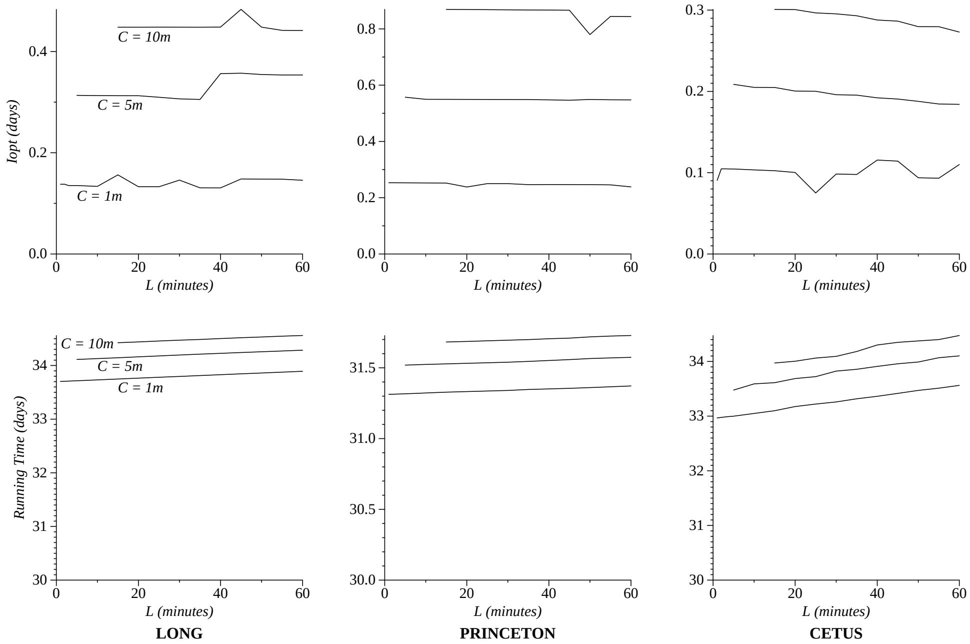


Figure 7: The optimal checkpoint interval as a function of latency

CETUS network, the places where the approximations differ the most from the correct values show the worst performance penalties. 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 optimal 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.

As a final remark about Figure 6, the middle row of graphs displays 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. Note that the points at which these curves meet the y-axis correspond to the availability of the respective systems. For example, the availability of the **PRINCETON** network is 0.962. Thus, one would expect an optimal expected running time of 30d/0.962 = 31.2d, which is approximately where its curve intersects the y-axis.

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. The results are displayed in Figure 7.

The top row of graphs shows the value of I_{opt} as a function of the latency. Though no consistent pattern emerges, to a first approximation it does appear that I_{opt} is independent of L. The second row of graphs shows

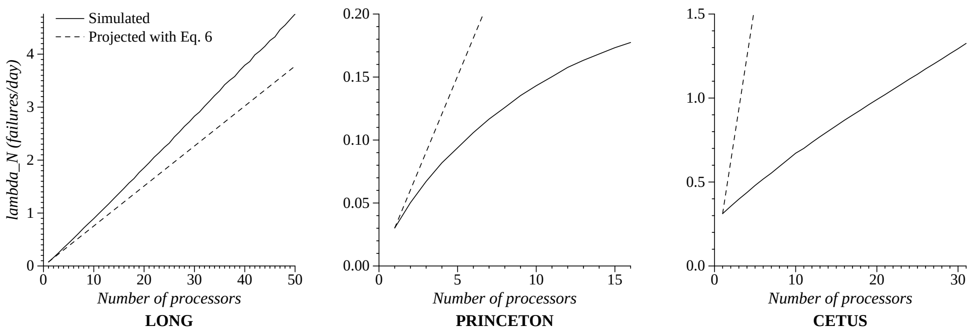


Figure 8: Determination of λ_N for each network.

the optimal running time for each value of L. As L increases, the running time 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 [Vai97].

7.4 Predicting 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, and the MTTF is calculated. 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 $N\lambda_1$. 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 $1.26N\lambda_1$, rather than $N\lambda_1$. In the **PRINCETON** and **CETUS** networks, λ_N grows much more slowly than $N\lambda_1$. 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_1 than to $N\lambda_1$.

From this data, it is hard to draw general conclusions concerning the failure rate of parallel systems. Certainly assuming that $\lambda_N = N\lambda_1$ 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. [Vai95, WF96, KS97]).

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 require 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. To our surprise, 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 $N\lambda_1$. In systems where failures are not independent, like the **CETUS** network, $\lambda_N \ll N\lambda_1$.

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. It is a subject of future work analyze the mathematical underpinnings of our results.

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.

References

[ACD+96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[Bac81] B. Baccelli. Analysis of a service facility with periodic checkpointing. *Acta Informatica*, 15:67–81, 1981.

[CKSS91] M. J. Crowder, A. C. Kimber, R. L. Smith, and T. J. Sweeting. *Statistical Analysis of Reliability Data*. Chapman & Hall, London, 1991.

[CPL97] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing*, San Jose, November 1997.

[CR72] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21:546–556, June 1972.

[CS84] L. H. Crow and N. D. Singpurwalla. An empirically developed fourier series model for describing software failures. *IEEE Transactions on Reliability*, R-33:176–183, June 1984.

[Dud83] A. Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, 1983.

[EJW96] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[GD78] E. Gelenbe and D. Derochette. Performance of rollback recovery systems under intermittant failures. *Communications of the ACM*, 21(6):493–499, June 1978.

[Gel79] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26:259–270, April 1979.

[GRW88] R. Geist, R. Reynolds, and J. Westall. Selection of a checkpoint interval in a critical-task environment. *IEEE Transactions on Reliability*, 37:395—400, October 1988.

[HKW95] Y. Huang, C. Kintala, and Y-M. Wang. Software tools and libraries for fault tolerance. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):5–9, Winter 1995.

[KS97] G. P. Kavanaugh and W. H. Sanders. Performance analysis of two time-based coordinated checkpointing protocols. In *1997 Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'97),*, Taipei, Taiwan, December 1997.

[KSL84] C. M. Krishna, K. G. Shin, and Y. H. Lee. Optimization criteria for checkpoint placement. *Communications of the ACM*, 27:1008–1012, October 1984.

[LM88] P. L'Ecuyer and J. Malenfant. Computing optimal checkpointing strategies for rollback and recovery systems. *IEEE Transactions on Computers*, 37(4):491–496, April 1988.

[LMG95] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *14th Symposium on Reliable Distributed Systems*, pages 2–9, Bad Neuenahr, September 1995. IEEE.

[LS92] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Usenix Winter 1992 Technical Conference*, pages 283–290, San Francisco, CA, January 1992.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under unix. In *Usenix Winter 1995 Technical Conference*, pages 213–223, January 1995.

[PL94] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[SGDM93] V. S. Sunderam, G. A. Geist, J. J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Journal of Parallel and Distributed Computing*, 1993.

[SLL87] K. G. Shin, T-H. Lin, and Y-H. Lee. Optimal checkpointing of real-time tasks. *IEEE Transactions on Computers*, 36(11):1328–1341, November 1987.

[SPS96] SPSS, Inc. SPSS for Windows. Release 7.5, see `http://www.spss.com`, 1996.

[Ste96] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium*, April 1996.

[TB84] S. Toueg and Ö. Babaoglu. On the optimum checkpoint selection problem. *SIAM Journal on Computing*, 13:630–649, August 1984.

[Vai95] N. H. Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Ottawa, May 1995.

[Vai97] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, August 1997.

[WF96] K. Wong and M. Franklin. Checkpointing in distributed systems. *Journal of Parallel & Distributed Systems*, 35(1), May 1996.

[WHV+95] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and its applications. In *25th International Symposium on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

[You74] J. S. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.