Performance Optimization of Checkpointing Schemes With Task Duplication*

Avi Ziv Jehoshua Bruck

California Institute of Technology
Mail Code 116-81
Pasadena, CA 91125
E-mail: {avi, bruck}@systems.caltech.edu

Abstract

Checkpointing schemes enable fault-tolerant parallel and distributed computing by leveraging the redundancy in hardware and software resources. In these systems, checkpointing serves two purposes: it helps in detecting faults by comparing the processors states at checkpoints, and it facilitates the reduction of fault recovery time by supplying a safe point to rollback to. The efficiency of checkpointing schemes is influenced by the time it takes to perform the comparisons and to store the states. The fact that checkpoints consist of both storing of states and comparison between states, with conflicting objectives regarding the frequency of those operations, limits the performance of current checkpointing schemes.

In this paper we show that by tuning the checkpointing schemes to a given architecture, a significant reduction in the execution time can be achieved. We will present both analytical results and experimental results that were obtained on a cluster of workstations and a parallel computer.

The main idea is to use two types of checkpoints: compare-checkpoints (comparing the states of the redundant processes to detect faults) and store-checkpoints (storing the states to reduce recovery time). With two types of checkpoints, we can use both the comparison and storage operations in an efficient way and improve the performance of checkpointing schemes. As a particular example of this approach we analyzed the DMR checkpointing scheme with store and compare checkpoints on two types of architectures, one where the comparison time is much higher than the store time (like a cluster of workstations connected by a LAN) and one where the store time is much higher than the comparison time (like the Intel Paragon supercomputer). We have implemented a prototype of the new DMR schemes and run it on workstations connected by a LAN and on the Intel Paragon supercomputer. The experimental results we obtained match the analytical results and show that in some cases the overhead of the DMR checkpointing schemes on both architectures can be improved by as much as 40%.

Key Words: fault-tolerance, checkpointing, task duplication, parallel computing, performance optimization.

*The research reported in this paper was supported in part by the NSF Young Investigator Award CCR-9457811, by a grant from the IBM Almaden Research Center, San Jose, California and by a grant from the AT&T Foundation. This research was performed in part using the CSCC parallel computer system operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Caltech.

1 Introduction

Checkpointing is a common technique for reducing the execution time of tasks in the presence of faults. Checkpointing consists of saving intermediate states of the task in a reliable storage, and upon detection of a fault restoring the previous stored state. Hence, checkpointing enables to reduce the time to recover from a fault while minimizing the lost processing time.

Early studies have shown that the rate of transient faults in a computer system is 10 to 30 times higher than the rate of permanent faults [12]. Transient faults can be hard to detect because they can cause a change in the task state that might lead to a wrong output from the task, without causing the task to crush. Task duplication [1] can be used to detect transient faults that cannot be detected internally, and hence increase the reliability of the system. In task duplication, the task is executed on more than one processor and the states of the processors are compared to detect faults.

Several papers (such as [7],[8],[9],[11], [14]) describe schemes that combine checkpointing and task duplication. In the schemes described in these papers, each checkpoint serves two purposes. The first is to save the processor state, and to reduce the fault-recovery time by supplying an intermediate correct state, thus avoiding rollback to the beginning of the task. The second purpose is fault-detection, which is achieved by executing the task on more than one processor, and comparing the processors states at each checkpoint.

The length of the optimal interval between checkpoints, that minimizes the average execution time of a task, is determined by the checkpointing overhead [2][3][5]. In checkpointing schemes with task duplication this overhead consists of the time to store the processors states and the time to compare these states. When there is a big difference between the time to store the processors states and the time to compare these states, the overhead time is determined mainly by the operation that takes a longer time. As a result, the operation that takes less time is not used efficiently, and therefore the schemes have an unnecessary overhead that can be avoided. An example for systems with a big difference between the storage and comparison times are clusters of workstations connected by a LAN, where the bandwidth of the communication subsystem (using Ethernet technology) is roughly an order of magnitude smaller than the bandwidth of the local storage subsystem. Another example are multiprocessor supercomputers, where the bandwidth of the communication subsystem is usually higher than the bandwidth of the local storage subsystem.

In this paper we present two methods to reduce the average execution time of checkpointing schemes with task duplication. The first method is to tune the the scheme to the specific system it is implemented on, and use both the compare and store operations efficiently. The second method is to reduce the comparison time by using signatures.

Tuning the scheme to the system is done by using two types of checkpoints, *compare-checkpoints (CCP)* and *store-checkpoints (SCP)*. The compare-checkpoints are used to compare the states of the processors without storing them, while in the store-checkpoints, the processors store their states without comparison. The two operations can still be used together in the same checkpoint. We refer to this type of checkpoint, with both store and compare operations, as a *compare-and-store checkpoint (CSCP).* Using two types of checkpoints enables us to choose different frequencies for the two checkpoint operations, and utilize both operations in an efficient way. When the checkpoints that are associated with the operation that takes less time are used more frequently than the checkpoints associated with operation that takes more time, the recovery time after fault can be reduced without increasing the checkpoint overhead. This leads to a significant reduction in the average execution time of a task. This work is extension of [15], where we presented the store and compare checkpoints, and showed how they can be used to reduce the average execution time in LAN based distributed systems.

To illustrate how store and compare checkpoints can be used, we show how to modify the DMR (Double Modular Redundancy) scheme to take advantage of both types of checkpoints. We analyze the DMR scheme with store and compare checkpoints, and use the analysis results to compare the average execution time of a task using the traditional DMR scheme with the average execution time of a task using the proposed DMR scheme with store and compare checkpoints. The comparison results show that in both types of systems, a significant reduction of up to 30% in the overhead of the execution time can be achieved when two types of checkpoints are used.

The execution time of checkpointing schemes can be reduced even more if instead of comparing the whole states of the processors, a comparison of short signatures of the states is performed. In systems with high comparison time signatures can significantly reduce the checkpoint overhead, and hence reduce the execution time of a task. Even in systems where the overhead of compare-checkpoints is small compared to the overhead of store-checkpoints, signatures can still be used to reduce the execution of the task, because with signatures the frequency of compare checkpoints can be increased causing reduction in the recovery time.

Signatures can reduce the reliability of a scheme, because it might happen that a faulty state will be mapped to the same signature as the correct state. We show that by mixing signatures and full-comparisons a fully reliable scheme can be obtained with a significant reduction in the execution time.

We implemented a prototype of the proposed DMR scheme with store and compare checkpoints on a cluster of workstations and on the Intel Paragon supercomputer. We measured the execution time of a test task using the schemes, with and without signatures. Comparison of the measured execution time with the calculated execution time shows that the analytical results for both systems, with and without signatures, match the measured execution time very well. Comparison of the measured execution time of the traditional DMR with the execution time of the new schemes reinforces the conclusion that using two types of checkpoints can result in a significant reduction in the execution time, and that signatures can be used to reduce the execution time even more, for a total of 40% reduction in execution time overhead.

The rest of the paper is organized as follows. In Section 2 we describe how the store and compare checkpoints can be used to reduce the execution time of a task, and provide the analysis of the DMR scheme with store and compare checkpoints. In Section 3 we discuss the use of signatures to shorten the comparison time. In Section 4 the implementations of the DMR scheme with store and compare checkpoints on a cluster of workstations and on the Intel Paragon supercomputer are described, and the experimental results are shown. Section 5 concludes the paper.

2 Checkpointing Schemes with Store and Compare Checkpoints

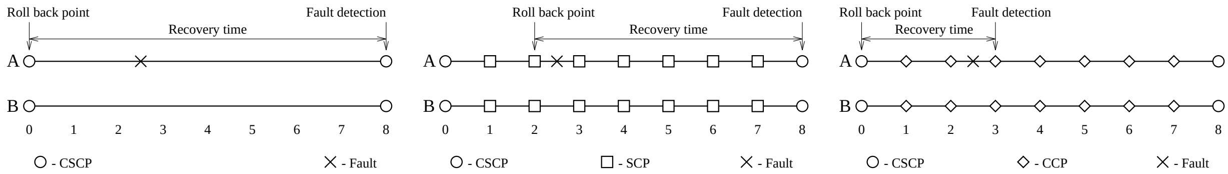
In many parallel and distributed systems there is a big difference between the time to store the processors states and the time to compare these states. In these systems we can add checkpoints that corresponds to the faster operation, and use these operations to reduce the recovery time after faults. For example, in systems with higher comparison time, like clusters of workstations connected by a LAN, we can add SCPs to reduce the recovery time; while in systems with higher storage time, like multiprocessor supercomputers, additional CCPs can be used to reduce the recovery time. Because the additional checkpoints use the operation that takes less time, their effect on the checkpointing overhead is small.

In this section we show how store and compare checkpoints can be used to improve the performance of existing checkpointing schemes. To illustrate how the modifications to the existing schemes are done, and to show how the modified schemes can be analyzed, we use the DMR (Double Modular Redundant) scheme. In this scheme the task is executed on two processors. At each checkpoint the states of the two processors are compared. If the states match, a correct execution is assumed, and the processors continue to the next interval. If the states do not match, both processors are rolled back to the previous checkpoint, and the execution of the same interval is repeated.

In the execution example in Figure 1a, the states of the processors are compared and stored at the end of the interval (checkpoint 8). The states do not match because processor A had a fault. Hence, both processors are rolled back to checkpoint 0 and the whole interval is executed again.

In Figures 1b and 1c the execution of the interval, that was shown in Figure 1a, is repeated with additional SCPs (Figure 1b) or CCPs (Figure 1c). In Figure 1b seven additional SCPs are placed between the CSCPs. These SCPs can be used to reduce the rollback to checkpoint 2 instead of checkpoint 0, and shorten the fault recovery period. In Figure 1c seven CCPs are added between the CSCPs. This cause the fault to be detected at checkpoint 3 instead of checkpoint 8, and hence shorten the recovery time.

To analyze the average execution time of a task using the DMR scheme with additional store or compare checkpoints, we assume that a task of length 1 has to be executed. The task is divided



(a) Traditional DMR

(b) DMR with additional store-checkpoints

(c) DMR with additional compare checkpoints

Figure 1: Execution of one interval with the DMR scheme

into $m \cdot n$ intervals of length $t_I = \frac{1}{m \cdot n}$ and at the end of each interval a checkpoint is placed. A CSCP is placed every n intervals. The task is executed on two processors using the DMR scheme.

The processors that execute the task are vulnerable to transient faults. The faults occur in each processor according to a Poisson random process with rate λ. The faults in the processors are independent of each other. We assume that faults can occur while the processors execute the task, but not during checkpoints.

Let t_s denote the time to store the processors states, t_{cp} denote the time to compare the processors states, and t_r denote the time to rollback the processors to the last saved state. Let c be the probability that no faults occurred in both processors, while executing a single interval. Because the faults in the processors are independent of each other, we can write the following expression for c;

$$c = (e^{-\frac{\lambda}{m \cdot n}})^2 = e^{-\frac{2\lambda}{m \cdot n}}.$$

The probability that no fault occurred between the CSCPs is simply c^n.

Let I denote last interval before the first fault occurred, that is no fault occurred in intervals $1, 2, \ldots, I$, but a fault occurred in interval $I + 1$. Let Q be the number of CSCPs with identical states before the fault is detected, that is $Q = \left\lfloor \frac{I}{n} \right\rfloor$. I and Q are geometric random variables with parameters c and c^n respectively.

2.1 Analysis of DMR with additional SCPs

In schemes with additional SCPs, after a fault is detected, we need to find the most recent checkpoint with identical states and roll back to it. To make the search for the most recent identical checkpoints efficient, a binary search is performed on the Huffman tree [4] induced by the probabilities of rollback to each SCP. The average number of comparisons, $\overline{C}$, using the Huffman tree is approximately $\log_2 n$, when a CSCP is placed every n checkpoints. We assume that the time to rollback the processors is included in the time to find the most recent checkpoint with identical states.

In the example in Figure 1b seven additional SCPs, numbered 1 to 7, are placed between checkpoints 0 and 8. During the execution a fault occur in processor A, and the comparison at checkpoint 8 fails. In the first step of the search for the most recent matching checkpoint, the states at checkpoint 4 are compared. The states do not match because processor A had an earlier fault. Next the states at checkpoint 2 are compared. These states match. After that step we know that the required checkpoint is either 2 or 3. Finally the states at checkpoint 3 are compared. They do not match and a rollback to checkpoint 2 is done.

After a CSCP is reached and the fault recovery process is completed, the next CSCP is placed n intervals from the last matching checkpoint. For example, in the execution in Figure 1b, after the rollback to checkpoint 2, the next CSCP is placed at checkpoint 10, and checkpoint 8 becomes a store-checkpoint.

The next proposition gives the average execution time of a task using the DMR scheme with additional store checkpoints.

Proposition 1 *The average execution time of task of length 1, using the DMR scheme with additional SCPs, denoted by $\overline{T}_S$, with $m \cdot n$ checkpoints, CSCP every n intervals, and faults according to independent Poisson processes with rate λ in the processors is*

$$\overline{T}_S = \frac{n(1-c)}{c(1-c^n)} \cdot (1 + mnt_s + m\left[1 + (1-c^n)\overline{C}\right] t_{cp}). \tag{1}$$

Proof: To calculate the average execution time of a task, we need to find the progress, measured in intervals, and elapsed time from the time the last rollback is completed (or the beginning of the execution) until the first fault is detected and its rollback is completed.

After the fault is detected, the task is rolled back to the last matching checkpoint. Therefore, the progress X_S is equal to the last interval without error. The average progress until the first fault is

$$\overline{X}_S = \overline{I} = \frac{c}{1-c}.$$

The time between CSCPs is the time to execute n intervals of the task, the time to store the states of the processors after each interval, and the time to compare the states of the processors at the CSCP. The first fault is detected at the $Q+1$'st CSCP. The time to find the last matching checkpoint after the fault is detected is $\overline{C} \cdot t_{cp}$. Therefore, the amount of time until the first fault is detected, and the rollback to the last matching checkpoint is completed is

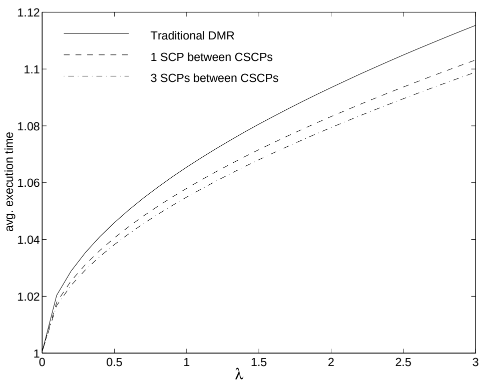
$$D_S = (Q+1)(n \cdot (t_I + t_s) + t_{cp}) + \overline{C} \cdot t_{cp},$$

and the average time is

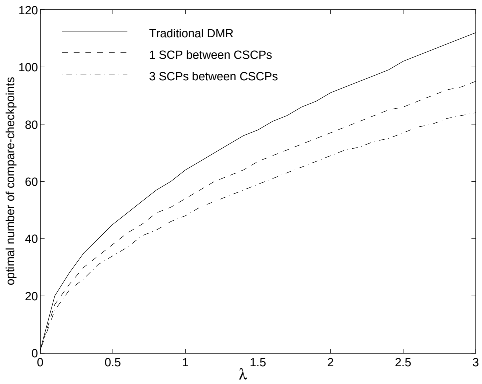
$$\overline{D}_S = (\overline{Q}+1)(n \cdot (t_I + t_s) + t_{cp}) + \overline{C} \cdot t_{cp} = \frac{n \cdot (t_I + t_s) + t_{cp}}{1-c^n} + \overline{C} \cdot t_{cp}.$$

The average time to execute the whole task is

$$\overline{T}_S = n \cdot m \cdot \frac{\overline{D}_S}{\overline{X}_S} = \frac{n(1-c)}{c(1-c^n)} \cdot (1 + mnt_s + m\left[1 + (1-c^n)\overline{C}\right] t_{cp}). \quad \blacksquare$$



(a) average execution time



(b) number of CSCPs

Figure 2: Comparison between DMR scheme with and without additional store-checkpoints

In Figure 2 the traditional DMR scheme performance is compared to the performance of the DMR scheme with 1 or 3 store checkpoints between CSCPs ($n = 2$ or $n = 4$). Figure 2a shows the average execution time of a task as a function of the fault rate λ. The time to store the processors states and compare them are $t_s = 10^{-5}$ and $t_{cp} = 5 \cdot 10^{-4}$. For each value of n and for each λ, the interval between CSCPs is chosen such that the execution time is minimized. It can be seen from the figure that the using two types of checkpoints gives a significant reduction in the overhead of the execution time.

In Figure 2b the number of CSCPs that achieves the average execution time of Figure 2a is shown. The figure shows us that using two types of checkpoints enables to place the CSCPs further apart and hence reduce the needed synchronization intervals between the processors.

2.2 Analysis of DMR with additional CCPs

The analysis of the average execution time of the DMR scheme with additional CCPs is similar to the analysis of the DMR scheme with additional SCPs presented above. At each compare-checkpoint, the states of the processors are compared. If the states are identical, then the execution continues with the next interval. If the states are different, the execution is rolled back to the last stored state. When a CSCP is reached and the states of the processors are identical, these states are saved and they can be used as a point to rollback to.

The average execution time of a task using the DMR scheme with additional compare checkpoints is given in Proposition 2.

Proposition 2 *The average execution time of task of length 1, using the DMR scheme with additional CCPs, denoted by $\overline{T}_C$, with $m \cdot n$ checkpoints, CSCP every n intervals, and faults according* *to independent Poisson processes with rate λ in the processors is*

$$\overline{T}_C = \frac{1-c^n}{nc^n(1-c)}(1+mnt_{cp}) + mt_s + \frac{m(1-c^n)}{c^n}t_r. \tag{2}$$

Proof: To calculate the average execution time of a task, we need to find the progress, measured in intervals, and elapsed time from the time the last rollback is completed (or the beginning of the execution) until the first fault is detected and its rollback is completed.

After the fault is detected at the end of interval $I+1$, a rollback to the end of interval $n \cdot Q$ is performed. Therefore the progress between faults is $X_C = n \cdot Q$, and the average progress until the first fault is

$$\overline{X}_C = n \cdot \overline{Q} = \frac{n \cdot c^n}{1-c^n}.$$

The time between CSCPs is the time to execute n intervals of the task, the time to compare the states of the processors after each interval, and the time to store the states at the CSCP. The first fault is detected between CSCPs Q and $Q+1$, after interval $I+1$. The elapsed time until this fault is detected and the rollback is completed is

$$D_C = (I+1)(t_I + t_{cp}) + Q \cdot t_s + t_r,$$

and the average time is

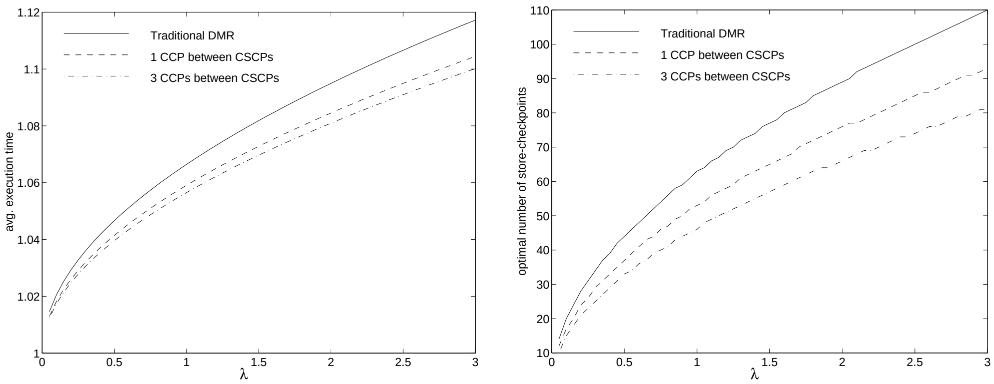
$$\overline{D}_C = (\overline{I}+1)(t_I + t_{cp}) + \overline{Q} \cdot t_s + t_r = \frac{t_I + t_{cp}}{1-c} + \frac{c^n}{1-c^n}t_s + t_r.$$

The average time to execute the whole task is

$$\overline{T}_C = n \cdot m \cdot \frac{\overline{D}_C}{\overline{X}_C} = \frac{1-c^n}{nc^n(1-c)}(1+mnt_{cp}) + mt_s + \frac{m(1-c^n)}{c^n}t_r. \quad \blacksquare$$

In Figure 3 the traditional DMR scheme performance is compared to the performance of the DMR scheme with 1 or 3 compare checkpoints between CSCPs ($n = 2$ or $n = 4$). Figure 3a shows the average execution time of a task as a function of the fault rate λ when the faults in the processors are iid Poison processes with rate λ. The time to compare the processors states, store them and rollback are $t_{cp} = 2.5 \cdot 10^{-5}$, $t_s = 5 \cdot 10^{-4}$ and $t_r = 5 \cdot 10^{-4}$. For each value of n and for each λ, the interval between checkpoints is chosen such that the execution time is minimized. It can be seen from the figure that the usage of compare points give a significant reduction in the overhead of the execution time.

In Figure 3b the number of CSCPs that achieves the average execution time of Figure 3a is shown. The figure shows us that the usage of compare checkpoints enables to reduce the number of checkpoints, and hence reduce the load of the I/O subsystem.



(a) average execution time (b) number of CSCPs

Figure 3: Comparison between DMR scheme with and without additional compare-checkpoints

3 Signatures

So far we assumed that at each compare-checkpoint the complete states of the processors are compared. Comparison of the complete states ensures detection of faults at the earliest possible compare-checkpoint, but it might result in a long checkpointing overhead. In this section we show how signatures can be used to significantly reduce the overhead of compare-checkpoints, without increasing the recovery time, causing overall reduction in the execution time of a task.

A signature is a mapping of the original space into a much smaller space. For example, a parity bit is a signature that maps the original space into a single bit. When signatures are used, at compare-checkpoints each processor calculates a signature of its state, and this signature is used to check if the states of the processors are identical. If the signatures are different, then the states that correspond to the signatures are different and faults have occurred. If the signatures are identical, we assume that the originating states are also identical, and no fault has occurred. Note that because the signature space is much smaller than the program-state space, many different states are mapped into the same signature, and there is a possibility that a fault might not be detected.

Signatures are used to detect and correct faults in many applications, such as communication channels and storage systems. Specific signatures are created to fit the application and environment in which they are used, so that the most likely fault patterns are always going to be detected. For example, many communications protocols use CRCs to detect errors [13], because they can detect bursts of errors.

In Sections 3.1 and 3.2 we describe the modifications to checkpointing schemes with additional SCPs or CCPs when signatures are used. To avoid reduction in the reliability of the schemes, the modified schemes combine signatures and full comparisons of the states. Further discussion on the scheme reliability is given in Section 3.3. We show how to analyze the DMR scheme with additional SCPs or CCPs and signatures. In the analysis we assume that the time to calculate and compare signatures is t_{sig}, and the the probability of miss-detections is ϵ. K denotes the number of signature comparisons that are made after the fault has occurred, until the fault is detected. K is a geometric random variable with parameter ϵ and mean $\frac{1}{1-\epsilon}$. All other assumptions made in the analysis in Section 2 are used here as well.

3.1 Signatures in Schemes with Additional SCPs

In systems with low communication bandwidth, the dominant time in comparing the states of two processors is the time to send this state from one processor to another. When signatures are used, the amount of data that need to be sent is much smaller, therefore the comparison time can be significantly reduced causing a big reduction in the average execution time.

Because there is a possibility of miss-detection when signatures are used, we can no longer assume that the state stored after the last CSCP is a correct state. Hence, to avoid unnecessary roll-backs, we need to keep all the stored states. Keeping all the stored states can very easily overload the storage system, and therefore is not practical. Therefore, the scheme that we use keeps only the last checkpoint that was verified by a full comparison and the states of the store checkpoints following the most recent CSCP. In the scheme all the comparisons are of signatures, except the comparison at the end of the program and comparisons to detect the last matching states. When the signatures of the states at a compare checkpoint are not identical, a binary search, similar to the one described in Section 2 is performed to find the most recent identical states among the store-checkpoints states. The comparisons in this search are full comparisons. If no identical states are found, the states of the last CSCP are compared. If this states are not identical, then a miss-detection occurred, and the task is rolled back to the last correct saved checkpoint. If during the search identical states are found, then the task is rolled back to that state, and the state is saved as the last correct state.

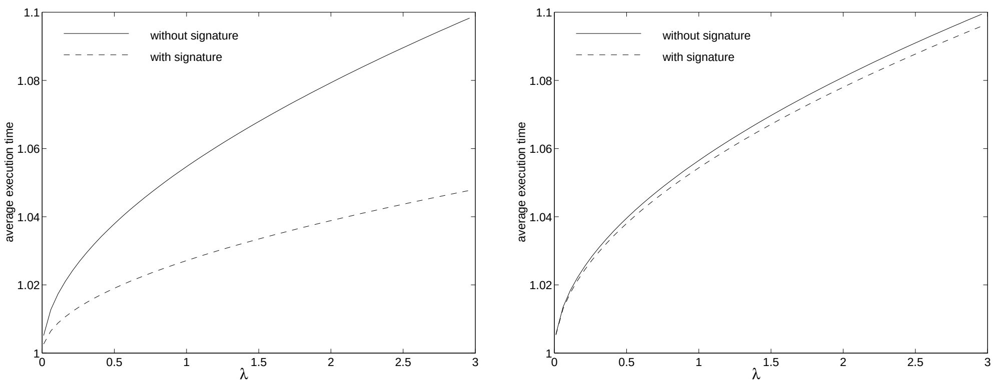
The average execution time of a task using the DMR scheme with additional store checkpoints and signatures is given in Proposition 3.

Proposition 3 *The average execution time of task of length 1, using the DMR scheme with additional SCPs and signatures, denoted by $\overline{T'}_S$, with $m \cdot n$ checkpoints, CSCP every n intervals, and faults according to independent Poisson processes with rate λ in the processors is*

$$\overline{T'}_S = \frac{n(1-\epsilon c^n)(1-c)}{(1-\epsilon)^2(1-c^n)c}(1 + m \cdot n \cdot t_s + m \cdot t_{sig}) + \frac{m \cdot n(1-c)}{(1-\epsilon)c}\overline{C}t_{cp}. \tag{3}$$

Proof: If the first fault occurred in interval number $I+1$, after the Q'th CCP, then the progress X'_S is given by

$$X'_S = \begin{cases} I & \text{if the fault is detected in CCP } Q+1, \\ 0 & \text{if the fault is not detected in CCP } Q+1. \end{cases}$$



(a) LAN based system (b) supercomputer system

Figure 4: Comparison between DMR schemes with and without signatures

and the average progress is

$$\overline{X'}_S = \overline{I}(1-\epsilon) = \frac{c(1-\epsilon)}{1-c}.$$

The time until the fault is detected and the rollback is completed depends on the number of miss-detections that occur.

$$D'_S = (Q+K)(n \cdot (t_I + t_s) + t_{sig}) + \overline{C} \cdot t_{cp}.$$

The average time until the fault is detected is

$$\overline{D'}_S = (\overline{Q} + \overline{K})(n \cdot (t_I + t_s) + t_{sig}) + \overline{C} \cdot t_{cp} = \frac{1-\epsilon c^n}{(1-\epsilon)(1-c^n)}(n \cdot (t_I + t_s) + t_{sig}) + \overline{C} \cdot t_{cp}.$$

The average time to execute the whole task is

$$\overline{T'}_S = \frac{n(1-\epsilon c^n)(1-c)}{(1-\epsilon)^2(1-c^n)c}(1 + m \cdot n \cdot t_s + m \cdot t_{sig}) + \frac{m \cdot n(1-c)}{(1-\epsilon)c}\overline{C}t_{cp}. \quad \blacksquare$$

Figure 4a shows the average execution time of a task using the DMR scheme with store-checkpoints, with and without signatures. The figure shows the average execution time as a function of the fault rate λ. The time to store the processors states is $t_s = 10^{-5}$. The time to compare the states signatures is $t_{sig} = 10^{-4}$, and the time to compare the whole state is $t_{cp} = 5 \cdot 10^{-4}$. The values of m and n where chosen such that the average execution Time is minimized for both the scheme with the signature and without it. The figure shows that for systems with low communication bandwidth, signature can significantly reduce the execution time of a task.

3.2 Signatures in Schemes with Additional CCPs

In systems with high communication bandwidth, the processors can share data very fast. Therefore to benefit from signatures, the signatures have to be simple and fast to calculate. In these systems reducing the comparison time enables us to place more compare-checkpoints between store-checkpoints, and hence reduce the fault detection and recovery time. Although this reduction in recovery time does not result in as big improvement in the performance as in systems with low communication bandwidth, still some improvement is possible.

To avoid unnecessary stores in case of miss-detection of faults, and to avoid rollback behind the last stored state, a full comparison of the states is performed before each store-checkpoint. As the time to compare the full states is still short compared to the store time, this full comparison has almost no effect on the checkpoint overhead. On the other hand, this full comparison ensures that every stored state is a correct state, and so when a fault is detected we, can roll back to the last saved checkpoint.

The average execution time of a task using the DMR scheme with additional compare checkpoints and signatures is given in Proposition 4.

Proposition 4 *The average execution time of task of length 1, using the DMR scheme with additional CCPs and signatures, denoted by $\overline{T'}_C$, with $m \cdot n$ checkpoints, CSCP every n intervals, and faults according to independent Poisson processes with rate λ in the processors is*

$$\overline{T'}_C = \frac{(1-c^n)(1-c\epsilon)}{nc^n(1-c)(1-\epsilon)}(1+mnt_{cp}) + mt_s + \frac{m(1-c^n)}{c^n}t_r. \tag{4}$$

Proof: The analysis of the scheme is identical to the analysis in Section 2.2. Because we perform a full comparison before storing a state, after a fault is detected we can always roll back to the last saved state. Therefore, the progress is the same as the scheme without signatures, and the average progress is

$$\overline{X'}_C = n \cdot \overline{Q} = \frac{n \cdot c^n}{1-c^n}.$$

The time until the fault is detected and the rollback is completed is given by

$$D'_C = (I+K)(t_I+t_{sig}) + Q \cdot t_s + t_r,$$

and the average time until the fault is detected is

$$\overline{D'}_C = (\overline{I}+\overline{K})(t_I+t_{sig}) + \overline{Q} \cdot t_s + t_r = \frac{1-c\epsilon}{(1-c)(1-\epsilon)}(t_I+t_{cp}) + \frac{c^n}{1-c^n}t_s + t_r.$$

The average time to execute the whole task is

$$\overline{T'}_C = \frac{(1-c^n)(1-c\epsilon)}{nc^n(1-c)(1-\epsilon)}(1+mnt_{cp}) + mt_s + \frac{m(1-c^n)}{c^n}t_r. \quad \blacksquare$$

Figure 4b shows the average execution time of a task using the DMR scheme with compare-checkpoints, with and without signatures. The figure shows the average execution time as a function of the fault rate λ. The time to store the processors states and the rollback time are $t_s t_r = 5 \cdot 10^{-4}$. The time to compare the states signatures is $t_{sig} = 1.5 \cdot 10^{-5}$, and the time to compare the whole

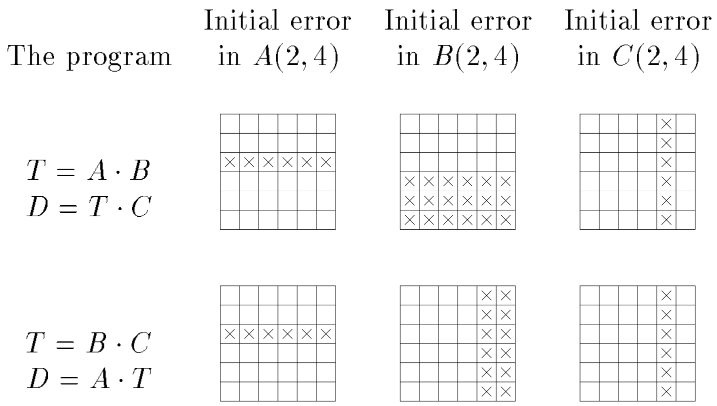


Figure 5: Propagation of a single error in a computer program

state is $t_{cp} = 2.5 \cdot 10^{-5}$. The values of m and n where chosen such that the average execution Time is minimized for both the scheme with the signature and without it. The figure shows that even in systems with high communication bandwidth, where most of the checkpointing overhead is caused by storing the processors states, signatures can still be used to reduce the execution time of a task.

3.3 Practical Issues

Unlike other applications that use signatures, in computer programs error patterns cannot be predicted. After a fault caused an error in some location of the program, this error is going to propagate to other locations, as the execution of the program continues. By the time the signature of the program is calculated, the pattern of the errors can not be predicted. Figure 5 shows the propagation of a single error in a program that multiplies three matrices. The program calculates $D = A \cdot B \cdot C$, where A, B, C and D are 6×6 matrices. The figure shows the effects of the location of the initial error and the implementation of the program on the locations of the errors in D. In all the cases shown in the figure the initial error occurred at the exact same time, about halfway through the first multiplication. As can be seen in Figure 5, even in a simple program it is hard to predict the pattern of the errors.

Because the pattern of the errors in a computer program is hard to predict, there is no signature that can always detect the most likely error patterns. So when signatures are used there is a non-zero probability of miss-detections of faults. Miss-detection of faults raises two important issues that have to be addressed when signatures are used. The first issue is the reliability of the scheme, or the probability that executing a program using the scheme will produce wrong results. The second issue is the increased recovery time that is caused by the miss-detections.

Reliability is an important issue, and the desire for high reliability is the reason task duplication is used for fault detection. Therefore, if signatures reduce the reliability of a scheme, they cannot be used despite the reduction in execution time they offer. We handle the reliability issue by mixing signatures comparisons and comparisons of the complete states of the processors. This way we can eliminate the reduction in reliability that is caused by signatures, without eliminating their benefits. To make the scheme fully reliable, a complete comparison is performed in the last checkpoint. Complete comparisons are also performed when the time it takes to perform them is small compared to the time it takes to perform other operations, such as storing the processors states in systems with high store time.

Another effect that signatures have on the execution time of a program, is the increased recovery time that is caused by the use of signatures. If a fault is not detected at a compare-checkpoint because signatures are used, the detection of the fault is delayed, and hence the recovery time is increased. If the probability of miss-detection is high enough, the longer recovery time might overcome the benefits of the signatures, and the overall effect of using signatures is an increase in the execution time.

Next we show that if the probability of miss-detection is reasonably low, the increase in recovery time is very small. Let T' be the average execution time of a program when the probability of miss-detection using signatures is 0. We assume that the faults in the system occur according to a Poisson process with rate λ, and that the probability of miss-detection is ϵ. We also assume a worst case scenario, where a miss-detected fault is detected only at the end of the program, and the execution of the whole program has to be repeated after the detection. The probability that a miss-detected fault occurred is given by

$$M = \Pr\{\text{miss-detected fault}\} = 1 - e^{-\lambda T' \epsilon} \simeq \lambda T' \epsilon.$$

The average number of times it takes to execute the program until there are no miss-detections is $\frac{1}{1-M}$, therefore the average execution time in this worst case scenario is

$$\overline{T} = \frac{T'}{1-M} = \frac{T'}{1-\lambda T' \epsilon}.$$

For example, if $\lambda T' = 1$ and $\epsilon = 10^{-4}$, then $\overline{T} = 1.0001 \cdot T'$.

Another requirement from the signatures is that they are simple to compute, so that the time it takes to calculate and compare signatures is shorter than the time it takes to compare the complete states of the processors. We tried several simple signatures, and checked the probability of miss-detection in several simple programs such as matrix multiplication, FFT, and matrix convolution. In every program, a single fault that caused a flip of uniformly distributed random bit in the program data space occurred. The time of the fault was uniformly distributed over the whole execution time of the program. The first signature that we tried was a 32 bit word checksum of the complete program state. It turned out that for some of the programs, this signature has a probability of miss-detection as high as 0.01. Better results were observed when we broke the state of the program into blocks, and rotated the bytes in the words. The signature that we decided to

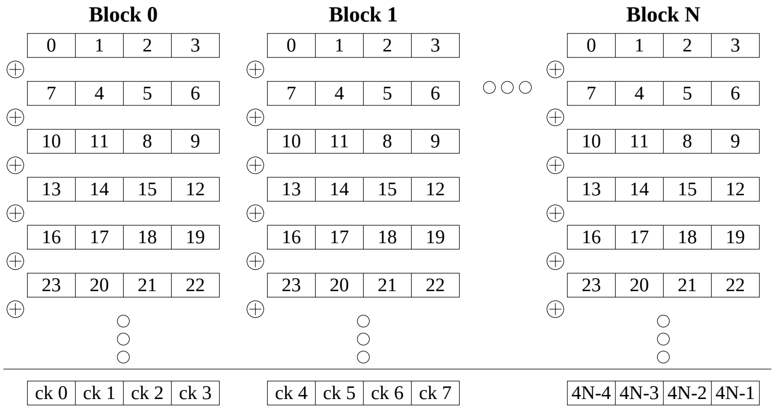


Figure 6: Simple signature for program state

use when we implemented the schemes is described in Figure 6. In this signature the program state is broken into blocks of 10K bytes. For each block a 32 bit checksum is calculated, such that every word is rotated one byte to the left compared to the previous word. The size of this signature is $4N$, where N is the number of 10K blocks in the program state ($N = \left\lceil \frac{\text{state size}}{10K} \right\rceil$). In the test programs that we tried, with state size of 100K byte to 10M byte, the probability of miss-detection using this signature was less than 10^{-4}. The time to calculate this signature is about the same time as the time to compare the complete states without sending it from one processor to another.

4 Experimental Results

To check the performance of the proposed DMR schemes with store and compare checkpoints, we implemented a prototype of the schemes on two types of systems, a cluster of workstations connected by a LAN (as an example for a system with higher comparison time) and the Intel Paragon supercomputer (as an example for a system with higher storage time). We measured the execution time of a test task, using the proposed scheme with two types of checkpoints, with and without signatures.

In the cluster of workstations, each workstation saves its own state on its local disk. The comparison of the states is done by sending the state of one of the workstations to the other using a bidirectional reliable stream, based on the TCP protocol [13]. In the Paragon system, the states of the processors are saved on the Parallel File System (PFS). The comparison of the states is done by sending the state of one of the processors to the other using the Paragon OSF/1 operating system calls [6]. In both systems, in the schemes that use signatures, we used the simple signature that is shown in Figure 6.

To compare the measured execution time of a task with the analytical results of Sections 2 and 3, we first measured the time to perform the operations the schemes need to achieve fault

Operation	Time in cluster [milliseconds]	Time in Paragon [milliseconds]
Store	10	150
Compare full state	360	10
Compare signatures	100	8

Table 1: Time to perform store and compare operations for 100K byte state

tolerance, namely storing the states of the processors, comparing the full states of the processors and comparing the signatures of the states. Table 1 shows the time to perform these operations in the cluster of workstations and the Paragon system, for a task with state size of 100K bytes. We found out that the time to store the states and compare the whole states grow linearly with the state size in both systems. The time to compare signatures grows linearly in the Paragon system, but remains constant in the cluster of workstations, probably because of the large set-up time for sending the signature.

The test task, which we used to check the performance of the schemes, consists of a block of memory that represents the state of the task. While the task executes, it continuously changes its state by performing read and write operations on it. The execution time of the test task, without checkpoints and faults in both systems, is 400 seconds, and the state size of it is 100K byte. In real life applications that use checkpoints, usually the execution time of the task is at least several hours and the state size of the task is several mega bytes. In order to perform many experiments and collect meaningful statistics, we used a test task with low execution time. The state size of the task was also reduced to keep the ratio between the execution time and the checkpointing overhead close to what we expect to find in real-life programs. In the experiments faults were generated synthetically according to a Poisson process, and each fault changed the value of a single random bit in the task state. The faults in the processors are independent processes with identical rates.

Table 2 compares the measured average execution time of the test task on the cluster of workstations to the analytical execution time of Eq. (1) (without signatures) and (3) (with signatures). For the scheme with signatures we assumed that the probability of miss-detection is $\epsilon = 10^{-4}$, although during all the experiments no miss-detection has occurred. The interval between the CSCPs is 8 seconds, and 3 store-checkpoints are placed between CSCPs ($n = 4$). The table shows, for four different values of fault rate λ, the average measured execution time, the calculated execution time, and the difference between them.

The table shows that the analysis results are very close to the measured values, with error of less than 3.5%. In all the cases the measured values are slightly larger than the calculated values. The reason for this difference is the need for synchronization of the two workstations at at each compare checkpoint.

	without signatures			with signatures		
Fault rate (faults/sec)	Measured time (sec)	Calculated time (sec)	Difference (%)	Measured time (sec)	Calculated time (sec)	Difference (%)
0.0025	441.49	432.09	2.2%	426.47	417.81	2.0%
0.0050	459.61	444.34	3.4%	438.21	428.85	2.2%
0.0075	465.39	456.77	1.9%	453.83	440.10	3.1%
0.0100	482.13	469.36	2.7%	463.26	451.57	2.6%

Table 2: Comparison between measured and calculated execution time of the DMR scheme with additional SCPs on cluster of workstations

	without signatures			with signatures		
Fault rate (faults/sec)	Measured time (sec)	Calculated time (sec)	Difference (%)	Measured time (sec)	Calculated time (sec)	Difference (%)
0.0025	423.17	420.06	0.7%	423.10	419.72	0.8%
0.0050	434.26	430.95	0.8%	436.49	430.68	1.3%
0.0075	446.67	442.18	1.0%	445.22	441.98	0.7%
0.0100	454.65	453.76	0.2%	454.83	453.63	0.3%

Table 3: Comparison between measured and calculated execution time of the DMR scheme with additional CCPs on the Intel Paragon

The analytical results for schemes with additional compare checkpoints (Eq. (2) and (4)) are compared to the measured execution time on the Paragon system in Table 3. The interval between the CSCPs is 8 seconds, and 3 compare-checkpoints are placed between CSCPs ($n = 4$). The table shows, for four different values of fault rate λ, the average measured execution time, the calculated execution time, and the difference between them.

The table shows that the analysis results are even closer to the measured values than in workstations cluster, with error of about 1%. The reason for the smaller difference is that in the Paragon the processors execute only the application program, so the synchronization of the processors at compare checkpoints takes less time.

To check the improvement in the execution time when additional store or compare checkpoints are used, and the additional improvement that signatures give, we executed the test task on both systems with different values of additional checkpoints, with and without signatures. In all the executions the interval between the CSCPs was 8 seconds. The execution time was measured for six values of n; 1 (traditional DMR), 2, 4, 6, 8 and 10. Table 4 shows the reduction in execution time when additional store-checkpoints are used and when signatures are used in the cluster of workstations. The second column shows the measured execution time for the traditional DMR scheme without signatures. The third column shows the lowest execution time when additional store-checkpoints are used, and the value of n for which this value was measured. The forth

	Traditional DMR	Additional SCPs without signatures		Additional SCPs with signatures	
Fault rate (faults/sec)	Execution time [Seconds]	Execution time [Seconds]	Overhead reduction	Execution time [Seconds]	overhead reduction
0.0025	446.50	438.87 $(n = 6)$	16.4%	426.47 $(n = 4)$	43.1%
0.0050	467.48	459.61 $(n = 4)$	11.7%	437.76 $(n = 6)$	44.0%
0.0075	485.08	465.39 $(n = 6)$	23.4%	451.22 $(n = 6)$	39.8%
0.0100	503.45	481.97 $(n = 6)$	20.7%	461.39 $(n = 6)$	40.6%

Table 4: Improvement in execution time with additional store-checkpoints and signatures in cluster of workstations

	Traditional DMR	Additional CCPs without signatures		Additional CCPs with signatures	
Fault rate (faults/sec)	Execution time [Seconds]	Execution time [Seconds]	Overhead reduction	Execution time [Seconds]	overhead reduction
0.0025	430.61	423.17 $(n = 4)$	24.3%	422.02 $(n = 6)$	28.1%
0.0050	447.08	432.61 $(n = 6)$	30.7%	430.92 $(n = 8)$	34.3%
0.0075	465.30	444.28 $(n = 6)$	32.2%	441.62 $(n = 8)$	36.3%
0.0100	483.78	453.32 $(n = 8)$	36.3%	448.00 $(n = 10)$	42.7%

Table 5: Improvement in execution time with additional compare-checkpoints and signatures in the Paragon supercomputer

column shows the reduction in overhead time when additional store-checkpoints are used, but without signatures. The last two columns show the lowest execution time and overhead reduction when signatures are used. Table 5 shows the same results for the Paragon system with additional compare-checkpoints.

The results in Tables 4 and 5 reinforce the analytical results of Sections 2 and 3 and show that using store or compare checkpoints between compare-and-store checkpoints can reduce the execution time of a task. Replacing some of the comparisons of full states by comparisons of signatures reduces the execution time even more. Overall the execution time of a task can be reduced by up to 10% and the overhead of the execution time by 40%. The optimal number of store or compare checkpoints depends on the time to store and compare the processors states, and the fault rate. The optimal number can be found using the analytical results given in Sections 2 and 3.

5 Conclusions

In this paper we have introduced two methods to improve the performance of checkpointing schemes with task duplication. One improvement method is tuning the scheme to the specific system it is implemented on, and the other improvement method is shortening the comparison time of a scheme by using signatures.

Tuning the scheme to the system is done by using two types of checkpoints, compare-checkpoints (comparing the states of the redundant processes to detect faults) and store-checkpoints (storing the states to reduce recovery time). Separating the comparison and store operations enables us to choose the optimal interval for each operation, without concerning about the other.

Another method to improve the performance of checkpointing schemes is to shorten the comparison time by using signatures. We have shown that simple signature can be used to reduce the execution time without affecting the reliability of the scheme.

We showed how the proposed schemes can be analyzed, and used the analysis results to compare the average execution time between the traditional DMR scheme and the DMR scheme with store and compare checkpoints, with and without signatures. The analysis was done for two types of systems, one with high comparison time compared to the storage time, and another with high storage time compared to the comparison time. The analysis results show that using two types of checkpoints can result in a significant reduction in the overhead of the execution time, and that signatures can further reduce the execution time, specially in systems with high comparison time.

We implemented a prototype of the proposed DMR schemes on a cluster of workstations and the Intel Paragon supercomputer, and measured the execution time of a task using the schemes. Comparison of the measured execution time with the calculated execution time shows a small difference, that is caused by the need to synchronize the processors that executed the task at compare checkpoints. Comparison of the measured execution time reinforces the conclusion that using two types of checkpoints and signatures can result in a significant reduction in the overhead of the execution time.

A challenging research direction is to create schemes that simultaneously address both relevant aspects of checkpointing in parallel and distributed systems, namely, the design of an efficient checkpointing scheme that achieves a consistent global state [10] and uses task duplication for fault detection. The combined checkpointing schemes have to address the complexity associated with both the detection of faults and maintaining of a consistent state. For example, if two copies of the same process receive messages in a different order, their states are going to be different and a comparison of the states will fail, even if no fault occurred. Also, when task duplication is used for fault detection it may cause a delayed detection of faults, as faults are detected only when comparison is performed. Namely, faults in one process can spread to other processes when this process sends messages to other processes.

References

[1] P. Agrawal. Fault tolerance in multiprocessor systems without dedicated redundancy. *IEEE Transactions on Computers*, 37:358–362, March 1988.

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

[3] E. G. Coffman and E. N. Gilbert. Optimal strategies for scheduling checkpoints and preventive maintenance. *IEEE Transactions on Reliability*, 39:9–18, April 1990.

[4] T. A. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley, 1991.

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

[6] Intel Corparation. *Paragon User's Guide*, June 1994. Order Number 312489-003.

[7] J. Long, W. K. Fuchs, and J. A. Abraham. Forward recovery using checkpointing in parallel systems. In *The 19th International Conference on Parallel Processing*, pages 272–275, August 1990.

[8] D. K. Pradhan. Redundancy schemes for recovery. TR-89-cse-16, ECE Department, University of Massachusetts, Amherst, 1989.

[9] D. K. Pradhan and N. H. Vaidya. Roll-forward checkpointing scheme: Concurrent retry with nondedicated spares. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 166–174, July 1992.

[10] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, June 1975.

[11] D. D. Sharma and D. K. Pradhan. A static roll-forward checkpointing scheme using three processors. Tr-93-061, Computer Science Department, Texas A&M University, 1993.

[12] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[13] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1989.

[14] A. Ziv and J. Bruck. Analysis of checkpointing schemes for multiprocessor systems. In *The 13th symposium on Reliable Distributed Systems*, pages 52–61, October 1994.

[15] A. Ziv and J. Bruck. Efficient checkpointing schemes over local area networks. In *The 1994 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, June 1994.