Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture[*][†]

Dhiraj K. Pradhan[‡] Nitin H. Vaidya

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

[*]Preliminary version presented at the 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems.

[†]Research reported is supported in part by the Office of Naval Research Grant # ONR N 0014-92-J-1366.

Key Words: Checkpointing, duplex systems, forward recovery, nondedicated spares, transient faults.

Abstract

Proposed here is a novel architecture for a fault-tolerant multiprocessor environment. It is assumed that the multiprocessor organization consists of a pool of active processing modules and either a small number of spare modules or active modules with some spare processing capacity. A fault-tolerance scheme is developed for duplex systems using checkpoints. Our scheme, unlike traditional checkpointing schemes, requires no rollbacks for recovering from single faults. The objective here is to achieve performance of a Triple Modular Redundant system using duplex system redundancy.

In the proposed scheme, at each checkpoint, the state of the two modules executing the task is compared for detection of faults. If a disagreement occurs, indicating a fault, the two differing states are both stored. Instead of performing usual rollback and retry, the following mechanism is used. The state at the preceding checkpoint, where both processing modules had agreed, is loaded into a spare module. The checkpoint interval in which the failure is detected is then "retried" on the spare module. Concurrently, the task continues forward on the two active modules, beyond the checkpoint where the disagreement occurred. At the next checkpoint the state of the spare is compared with the stored states of the two active modules (stored states correspond to where the disagreement occurred). The active module which disagrees with the spare is identified to be faulty. Once the faulty module is identified, the state of the faulty module is restored to the correct state by copying the state from the other active module, which is fault-free. The spare is released to the pool after recovery is completed. It is important to note that the spare is shared among many processor pairs and is used temporarily when faults occur.

Since the above mechanism achieves forward recovery, the proposed scheme is termed Roll-Forward Checkpointing Scheme (RFCS). The RFCS scheme allows recovery from single failures without the overhead of rollback. The advantage of the proposed scheme is that it achieves a lower average execution time with a lower variance as compared to the rollback scheme. This can be crucial for real-time systems with hard deadlines since lower variance enhances the predictability of the task completion time.

I. Introduction

An important aspect of a fault tolerant system is the mechanism used for fault detection and recovery from detected failures. This paper presents a novel roll-forward mechanism for achieving performance comparable to forward error recovery schemes such as TMR using significantly less redundancy. The scheme proposed here is applicable to all modular redundant systems in general. Because duplex systems are the most widely used and cost-effective modular redundant systems, our discussion correspondingly focuses on duplex systems. In a duplex system, whenever a fault is detected, the task is halted and retried. This results in performance degradation. In this paper a novel scheme is proposed where the task continues execution while the fault diagnosis and recovery functions are performed concurrently. The concept developed here has its roots in our earlier work [9]. A roll-forward scheme proposed independently in [8] requires more redundancy than our scheme. It is important to note that in many environments the amount of redundancy can be a concern because of power, weight and volume considerations.

Proposed here is a novel architecture for a fault-tolerant multiprocessor environment. It is assumed that the multiprocessor organization consists of a pool of active processing modules and either a small number of spare modules or active modules with some spare processing capacity. A fault-tolerance scheme is developed for duplex systems using checkpoints. Our scheme, unlike traditional checkpointing schemes, requires no rollbacks for recovering from single faults. The objective here is to achieve performance of a Triple Modular Redundant system using duplex system redundancy.

In the proposed scheme, at each checkpoint, the state of the two modules executing the task is compared for detection of faults. If a disagreement occurs, indicating a fault, the two differing states are both stored. Instead of the usual rollback and retry, the following mechanism is used for identification of the faulty processing module and recovery without rollback. The state at the preceding checkpoint, where both processing modules had agreed, is loaded into a spare module. The checkpoint interval in which the failure is detected is then "retried" on the spare module (this procedure is named "concurrent retry"). Concurrently, the task continues forward on the two active modules, beyond the checkpoint where the disagreement occurred. At the next checkpoint the state of the spare is compared with the stored states of the two active modules (stored states correspond to where the disagreement occurred). The active module which disagrees with the spare is identified to be faulty. Once the faulty module is identified, the state of the faulty module is restored to the correct state by copying the state from the other active module, which is fault-free. The spare is released to the pool after recovery is completed. It is important to note that the spare is shared among many processor pairs and is used temporarily when fault occurs.

Since the above mechanism achieves forward recovery, the proposed scheme is termed the Roll-Forward Checkpointing Scheme (RFCS). The RFCS scheme allows recovery from most common failures without the overhead of rollback. It is demonstrated here that the proposed scheme has potential performance advantages over conventional duplex systems which use rollback. Specifically, the advantage of the proposed scheme is that it achieves a lower average execution time with a lower variance as compared to the rollback scheme. This can be crucial for real-time systems with hard deadlines since lower variance enhances the predictability of the task completion time.

The proposed scheme requires process duplication and checkpointing. Many commercially available fault tolerant systems also employ duplication and checkpointing and architectures similar to that required for the proposed recovery scheme. For example, Sequoia Series 400 [10] system consists of multiple processing elements with large caches. Each processing element consists of two processors performing the same task. Failures are detected by comparing the output of the two processors. The main memory is assumed to be reliable and the cache is made recoverable by checkpointing (flushing) it periodically into the main memory. When a fault is detected, the processors restart execution from the last checkpointed state. Similarly, Tandem Non Stop Cyclone/R system [11] is a parallel architecture that provides greater availability by ensuring that if a processor fails its workload is automatically distributed to some other processor. The state of each process is backed up (checkpointed) periodically on another processor. This corresponds to passive duplication of processes. In the event of a failure, the process starts executing from the last backed up state. The above two commercial system examples illustrate that the approach proposed in this paper can be of practical significance. In particular the hardware overhead will be similar to the existing commercial systems that use duplication. However, the proposed approach differs in a fundamental way in that it uses checkpointing for fault detection *as well as* recovery, the above systems use it for recovery alone.

The rest of the paper is organized as follows. The system architecture under consideration is discussed in Section II. The basic approach is described in Section III. Section IV introduces some of the terminology used in our discussion. The proposed scheme is presented in Sections V and analyzed in Sections VI and VII. Section VIII elaborates on some implementation issues. Section IX discusses application of the proposed scheme to communicating processes. Section X discusses further work on the roll-forward checkpointing scheme presented in the paper. The paper concludes with Section XI. Derivations of results presented here are omitted due to lack of space; the interested reader is referred to [12].

II. System Architecture

The multiprocessor environment to be considered relies on task duplication to achieve fault tolerance. Such an environment has been used in many systems [1, 3, 4]. Figure 1 illustrates an example multiprocessor system organization that can implement the proposed roll-forward checkpointing scheme. Each processing module (PM) is assumed to consist of a processor and a private volatile storage (VS). All the processing modules are assumed identical. It is further assumed that each PM can access a stable storage (SS). The stable storage associated with each PM is accessible by the other modules in the presence of PM failure. A reliable *Checkpoint Processor* (CP) is assumed accessible from all the processing modules in the system. The CP can be centralized or distributed and orchestrates the fault detection and recovery functions. The CP detects module failures by comparing the state of each pair of processing modules (PMs) which perform the same task. The state of a process is an image of all the variable memory and registers associated with the process [6]. One can either compare the complete checkpoints or just signatures of the checkpoints for efficiency.

Apart from the processing modules executing duplicated tasks, it is assumed that a small number of modules are available as spares to be utilized for performing diagnosis and recovery when a duplex system experiences a failure. These modules may be non-dedicated spares to be used temporarily for fault recovery. If spares are not available, it is assumed that active modules with spare capacity can be interrupted and used temporarily as spares.

The architecture of Figure 1 is used as an example to guide the discussion in the paper. Figure 1 illustrates only the connectivity between the modules, the stable storage and the Checkpoint Processor (CP) as required by the proposed scheme. Actual implementation may

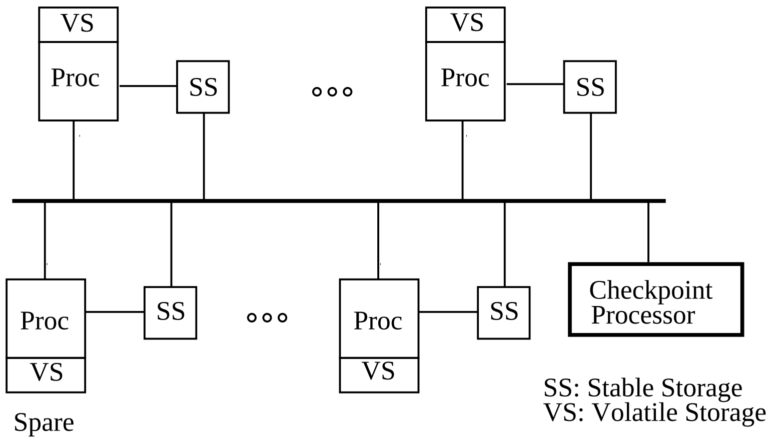

Figure 1: Logical system architecture

be quite different. Each PM, for example, may not have independent stable storage and the PMs may share a stable storage. The physical interconnection structure can be different from that shown in Figure 1.

The procedure for state or checkpoint comparison is as follows. Whenever a task checkpoints its state in the stable storage, the state is sent to the checkpoint processor (CP). When the CP receives the state from both of the modules executing a task, it compares the two states. If the two states match, the new checkpoint is considered correct and the previous checkpoint is replaced by the new one. If a mismatch occurs, then the previous checkpoint is not discarded and the recovery mechanism discussed in this paper is initiated.

When a write-back cache memory represents the volatile state and the main memory is stable (e.g., as in the Sequoia architecture [3]), the volatile storage (VS) block in a processing module in Figure 1 represents the write-back cache and the stable storage (SS) block represents the stable main memory. In this case, apart from periodic checkpointing, checkpoints need to be taken whenever the cache overflows. The contents of the stable memory locations should not be overwritten at a checkpoint until the comparison of the caches in the two modules in a duplex is completed by the CP. The cache contents may need to be buffered in a separate area in the SS modules (in this case, the stable main memory) until the comparison is complete.

Although our discussion of the RFCS scheme and analysis assumes that processes executed on different duplex systems do not communicate, the RFCS approach is also useful to the environment of communicating processes (see Section IX). In the presence of single faults, RFCS scheme can be used to avoid rollback, even when processes communicate by message passing.

The following discussion and performance analysis implicitly assumes that two faulty modules will always produce different checkpoints. The likelihood that failure will produce exactly identical erroneous checkpoints in both processors can be seen to be small. For further discussion of this issue and analysis the reader is referred to [12].

III. Basic Scheme

This section presents the basic concept behind the proposed approach using the most common fault scenario; a complete description is presented in Section V. Figure 2 depicts execution of two processing modules, named A and B, executing the same task. Assume that B fails in a checkpoint interval and other modules are fault-free. In Figure 2 this interval is named I_j. Then, the checkpoints of A and B will mismatch at the end of interval I_j. This mismatch will activate "concurrent retry" of checkpoint interval I_j on a spare, as follows.

1. The mismatching checkpoints of the two modules are saved. The previous checkpoint is then loaded into a spare module, say module S. The executable code for the task is also loaded into the spare module. The checkpoint interval in which the fault occurred is then retried on the spare module. Concurrently, A and B continue execution of the next checkpoint interval I_{j+1}.

2. After the spare completes interval I_j, the checkpoint of spare S is compared with the mismatching checkpoints of modules A and B. The checkpoint of S will mismatch with the checkpoint of B at the end of interval I_j, and match with A.

3. When this mismatch and match is detected, B is known to be faulty and A fault-free. Therefore, the state of B is made identical to the checkpoint of A. Now, A and B will both be in the correct state (provided module A did not fail in the second checkpoint interval named I_{j+1}).

4. Concurrent retry mechanism then proceeds to determine whether module A failed in interval I_{j+1}. A complete discussion of how this is done is presented in Section V.

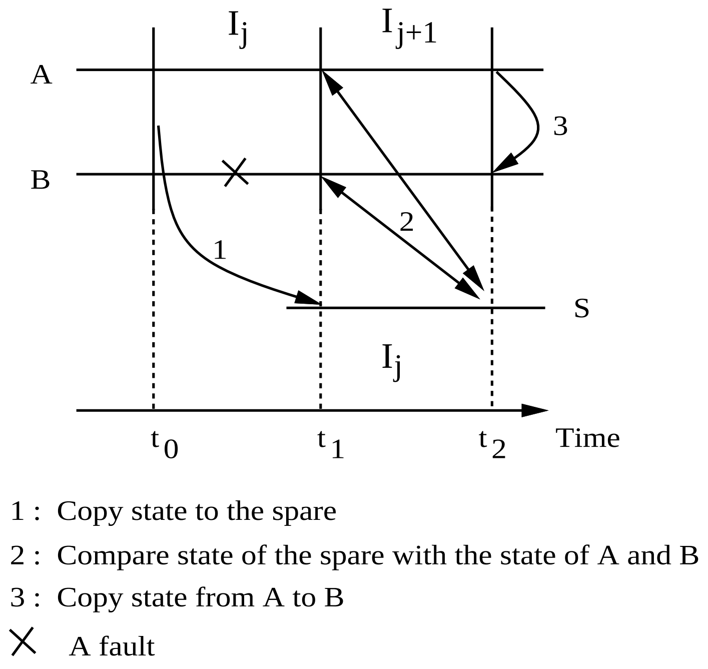
I_j
I_{j+1}
A
B
S
3
2
1
I_j
t_0
t_1
t_2
Time
1 : Copy state to the spare
2 : Compare state of the spare with the state of A and B
3 : Copy state from A to B
A fault

Figure 2: Roll-forward checkpointing scheme: basic concept

The proposed scheme avoids rollback in single fault scenarios. Multiple faults in two consecutive checkpoint intervals would require rollback. However, multiiple faults are much less likely than single faults.

IV. Preliminaries

The analysis is developed in two steps. First, we analyze a configuration consisting of a single duplex system and a spare module available when needed. This is then generalized to an environment where a spare is shared among many duplex systems. The two processing modules in the duplex system are named A and B. The spare module is named S. The replicas of the task executed on modules A and B are also called A and B. We use the terms *state of task* A(B) and *state of module* A(B) interchangeably.

The state of a processing module is assumed to be checkpointed under program control [7]. Checkpointing under program control enables two replicas of a task executed on two PMs to checkpoint at the same points during their execution.

The following introduces certain terminology to be used later. The computation required by the task is referred to as the *useful* computation. Other operations such as checkpointing are not considered a part of the *useful* computation. An *interval* consists of a period of useful computation followed and possibly preceded by other operations such as checkpointing and initiation of concurrent retry. An interval is identified by the useful computation performed in that interval. If module Q takes a checkpoint at the end of interval I_k, this checkpoint (or state of Q) is denoted as CP_{kQ}. If the states of the processing modules A and B at the end of interval I_k are identical, then CP_{kA} and CP_{kB} are identical and both are denoted simply as CP_k. When a processing module Q is rolled back to a state saved in checkpoint CP_x, we say that state of module Q is *made consistent with* CP_x. If module A or B fails in interval I_k then this interval is said to be a *faulty interval.* In the diagrams illustrating various fault scenarios, we use a *box notation* illustrated in Figure 3 below. The different operations listed in Figure 3 will be described later as they are used. Boxes shaded with the same pattern represent the same operation and require the same amounts of time.

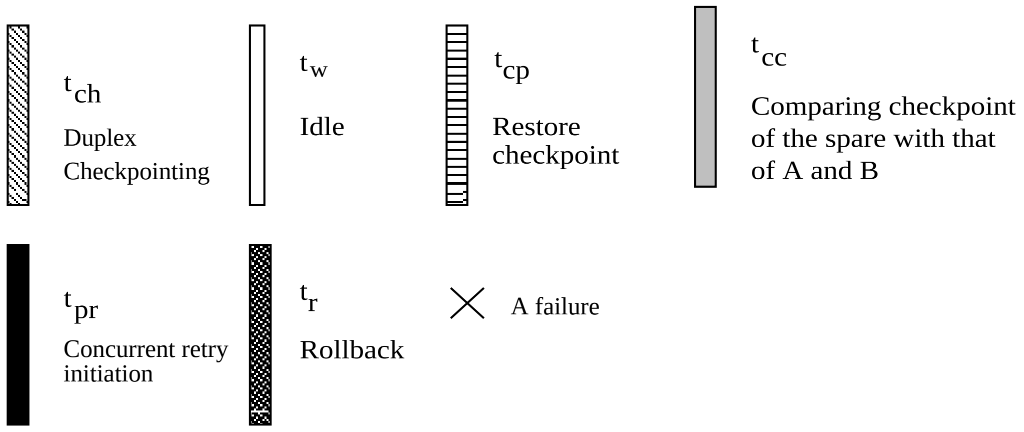

Figure 3: Box notation

Figure 4 illustrates the ROLLBACK scheme for a duplex system. The horizontal axes marked A and B represent execution of the two replicas of the task. Whenever a mismatch is detected in the state of modules A and B, the system is rolled back to the previous checkpoint.

The length of useful computations between two consecutive checkpoints is denoted by t_u. The time taken for checkpointing is denoted by t_{ch} which also includes the time required for comparing the checkpoints of processing modules A and B. We define $T = t_u + t_{ch}$. The time required to make the state of the two modules consistent with a previous checkpoint is named t_r. If the failure occurs in the first interval of execution of the task, then the task is

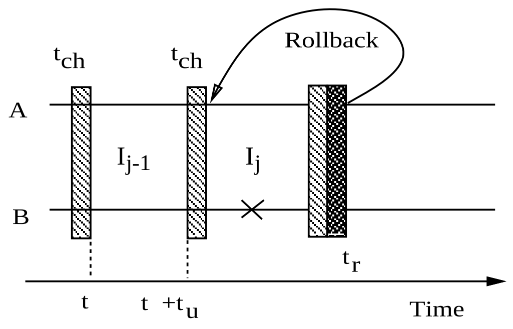

Figure 4: ROLLBACK scheme for duplex systems

restarted instead of rolling back. The time required for initiating a restart is t_s. The time required for making the state of the modules in the duplex consistent with the state saved by one of the modules is named t_{cp}.

V. Roll-Forward Checkpointing Scheme

Section III introduced the basic concept behind the proposed roll-forward checkpointing scheme (RFCS). This section describes the RFCS scheme in detail. As shown below, after a fault is detected, the spare module performs at most two successive intervals of concurrent retry to complete the recovery. Therefore, the spare module has three possible states – (i) spare not performing concurrent retry, (ii) spare in the first interval of concurrent retry, and (iii) spare in the second interval of concurrent retry. Depending on how the faults occur, there are four possible fault situations in RFCS. We now discuss each of these. Let t_0 denote the beginning of an interval denoted as I_j. Let the previous interval completed at t_0 be denoted as I_{j-1}. $CP_{(j-1)A}$ and $CP_{(j-1)B}$, checkpoints of A and B at the end of I_{j-1}, are assumed to be identical. The intervals following I_j are named I_{j+1} and I_{j+2}. It is assumed that the spare is not permanently faulty. The concurrent retry scheme cannot be used if no spares are available. The following discusses the four possible fault situations denoted as (A) through (D).

(A) No failure: Both processing modules A and B are fault-free in interval I_j (see Figure 5). If neither A nor B fails in interval I_j then at time t_1, the checkpoints of modules A and B will be identical. The execution continues on to the following interval.

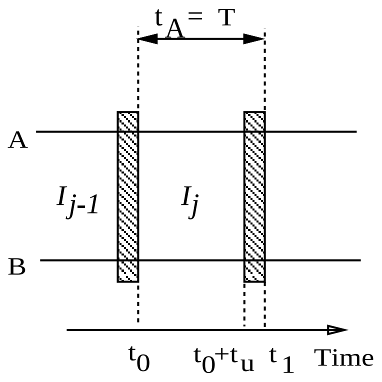

Figure 5: Situation (A) – No failure

(B) Single failure: As seen below, unlike conventional duplex systems, our scheme requires no rollback in this case. This situation occurs when a single module fails in interval I_j. Furthermore, no other module fails in intervals I_j and I_{j+1}.

Without loss of generality, assume that processing module B has a failure during interval I_j and modules A and S remain fault-free in intervals I_j and I_{j+1}. This case is illustrated in Figure 6.

When a fault occurs in interval I_j, the checkpoints CP_{jA} and CP_{jB} of A and B are not identical, and the fault is detected at time t_1 (see Figure 6). When a fault is detected, checkpoint CP_{j-1} is retained in the respective stable storages attached to modules A and B. In addition, both checkpoints CP_{jA} and CP_{jB} are saved. The following steps are then carried out to recover from the failure. At the beginning of the recovery process, identity of the faulty module B is not known to the Checkpoint Processor.

Step 1: Make the state of spare module S consistent with the state CP_{j-1} of modules A and B. Copy the task's executable code to S. The time required for this step, t_{pr}, can be minimized as discussed later. At time t_7, spare module S is ready to perform the computation in interval I_j. Concurrently, A and B continue execution of next interval I_{j+1}.

Step 2: When S completes the computation in interval I_j, its state CP_{jS} is compared with CP_{jA} and CP_{jB}. CP_{jS} is found identical to CP_{jA}, as A and S are both fault-free in interval I_j. Therefore, module A is considered fault-free in interval I_j. The time required for this state

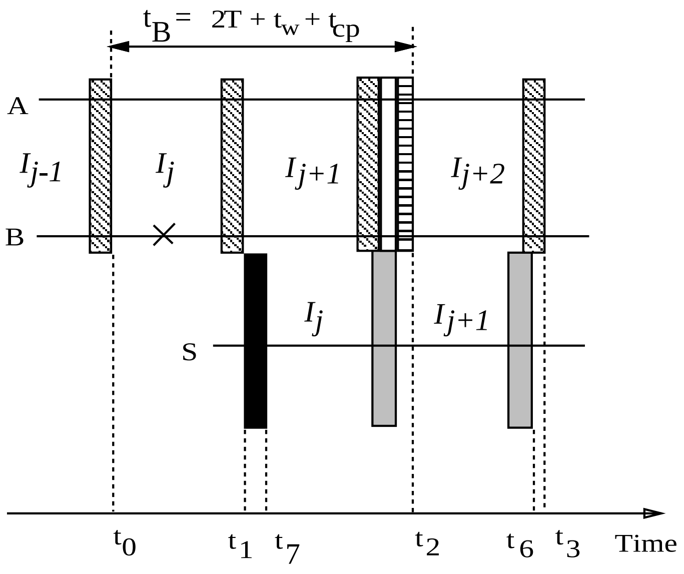

Figure 6: Situation (B) – Concurrent retry without rollback

comparison step is t_{cc}. The state CP_{jS} of spare module S need not be saved on the stable storage as it is used only for the comparison operation.

While S completes interval I_j, A and B complete interval I_{j+1} and take a checkpoint. Note that A and B were in different states at the start of I_{j+1}. A and B wait for state CP_{jS} to be compared with CP_{jA} and CP_{jB}. The length of the wait is denoted by t_w. Once it is determined that CP_{jA} and CP_{jS} are identical, the states of A and B both are made consistent with checkpoint $CP_{(j+1)A}$. The time required for this operation is termed t_{cp}. Note that A and B *did not* rollback to the start of interval I_j though processing module B failed during I_j. In the traditional rollback scheme, A and B would have rolled back.

Step 3: The concurrent retry is not complete yet. In the above step, the state of modules A and B was made consistent with $CP_{(j+1)A}$. However, as yet it is not known whether A failed during interval I_{j+1} and whether $CP_{(j+1)A}$ was erroneous or correct. We only know that CP_{jA} was correct.

After completing the state comparison in step 2, processing module S executes interval I_{j+1}. In the meanwhile, modules A and B execute interval I_{j+2}. When S completes I_{j+1}, its state $CP_{(j+1)S}$ is compared with $CP_{(j+1)A}$. As A and S are both assumed fault-free during

I_{j+1}, $CP_{(j+1)A}$ and $CP_{(j+1)S}$ will be found identical. $CP_{(j+1)A}$ and $CP_{(j+1)S}$ being identical implies that A was fault-free until the end of interval I_{j+1}. This state comparison is completed at time t_6 (see Figure 6).

As the computation performed by B in interval I_{j+1} is irrelevant, the concurrent retry scheme will tolerate a transient failure of module B in interval I_{j+1} without additional overhead. This is advantageous in situations where consecutive transient failures of a module are not independent of each other and a module affected by a failure is more likely to fail soon again. In our analysis of the concurrent retry scheme, however, we assume independence between any two failures.

Step 4: In the previous step, it is determined that processing modules A and B were in correct state at the start of interval I_{j+2}. With this, the concurrent retry initiated by failure of module B in interval I_j is completed. Any failures in interval I_{j+2} can be treated similarly to the failures in interval I_j. At time t_6, the spare is free to perform any other computation.

As seen above, concurrent retry avoided rollback in spite of a fault in B. The overhead incurred is only $(t_w + t_{cp})$. In the traditional rollback scheme the overhead is much larger, at least $(t_u + t_{ch} + t_r)$.

(C) Rollback after one interval of concurrent retry: In this situation, concurrent retry does not succeed and the system is rolled back to the state at time t_0. This situation occurs when one of the duplexed modules has a failure in I_j and another module also fails in I_j. There are three scenarios possible as listed in Table 1. For the sake of illustration, consider scenario C.1 (see Table 1) illustrated in Figure 7.

As shown in Figure 7, concurrent retry begins when CP_{jA} and CP_{jB} are found to be different. The concurrent retry mechanism *attempts* to perform the same steps as in situation (B). The procedure as detailed above for situation (B) is carried out through step 1. As S fails in interval I_j, in step 2, the comparison of CP_{jS} with CP_{jA} and with CP_{jB} will not result in a match. Therefore, the checkpoint processor cannot determine which of A and B is fault-free, if any. Hence, the duplex system must be rolled back to the last known correct checkpoint, CP_{j-1}. In this case, modules A and B rollback by two intervals and the rollback occurs after

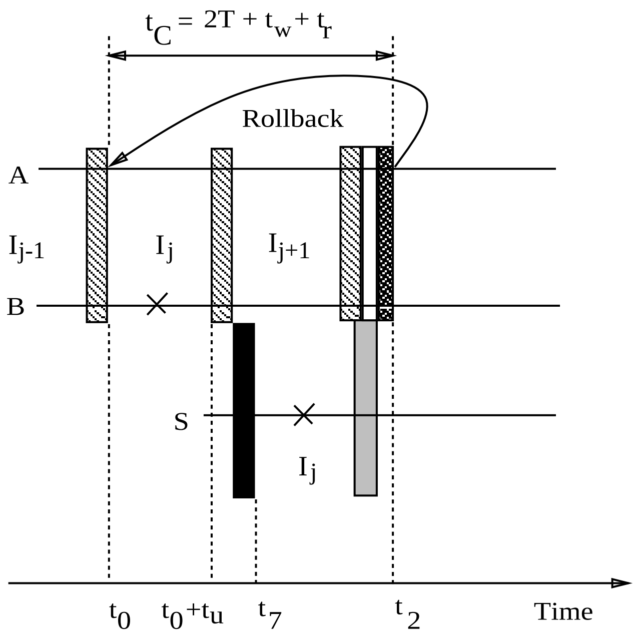

Figure 7: Situation (C) – Rollback after one interval of concurrent retry

the spare has completed one interval of concurrent retry. After the rollback is completed at time t_2, modules A and B are in a state identical to their state at t_0.

(D) Rollback after two intervals of concurrent retry: In this situation also, concurrent retry does not succeed and the system is rolled back. This case covers the four scenarios listed in Table 2. The four scenarios may be summarized as follows: Module B (A) has a failure in interval I_j and processing modules A (B) and S are fault-free in intervals I_j but A (B) fails in interval I_{j+1} and/or S fails in interval I_{j+1}.

Table 1: Fault scenarios possible in situation (C)

$X \equiv$ don't care

Scenario	Status in interval I_j		
	A	B	S
C.1	fault-free	faulty	faulty
C.2	faulty	fault-free	faulty
C.3	faulty	faulty	X

Table 2: Fault scenarios possible in situation (D)

$X \equiv$ don't care

Scenario	Status in interval I_j			Status in interval I_{j+1}		
	A	B	S	A	B	S
D.1	fault-free	faulty	fault-free	fault-free	X	faulty
D.2	fault-free	faulty	fault-free	faulty	X	X
D.3	faulty	fault-free	fault-free	X	fault-free	faulty
D.4	faulty	fault-free	fault-free	X	faulty	X

For the sake of illustration, consider fault scenario D.1 (see Table 2) illustrated in Figure 8. As shown in Figure 8, concurrent retry begins when CP_{jA} and CP_{jB} are found to be different. The concurrent retry mechanism *attempts* to perform the same steps as in situation (B). The procedure as detailed earlier for situation (B) is carried out through step 2. The

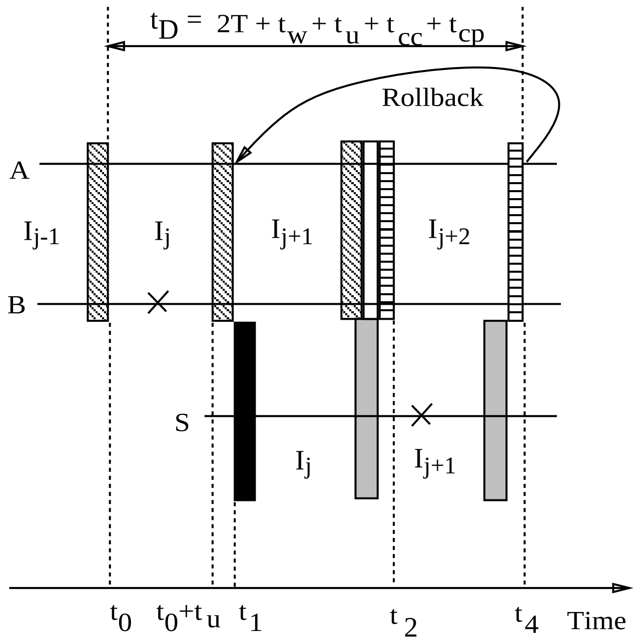

Figure 8: Situation (D) – Rollback after two intervals of concurrent retry

state comparison in step 2 will indicate that CP_{jA} and CP_{jS} are identical, implying that state CP_{jA} was the correct state at t_1. As explained in step 2 of (B), at time t_2, the state of A and B is made consistent with $CP_{(j+1)A}$.

As S fails in interval I_{j+1}, the comparison of $CP_{(j+1)A}$ and $CP_{(j+1)S}$ performed in step 3 will result in a mismatch. Now, there is no way to determine whether $CP_{(j+1)A}$ (state of A at the end of I_{j+1}) was correct. Therefore, the state of the duplex system at the start of interval I_{j+2} cannot be guaranteed to be correct. Hence the duplex system rolls back to the last known correct checkpoint, CP_{jA}. In this case, processing modules A and B rollback by two intervals and the rollback occurs after the spare has completed two intervals of concurrent retry. The time required for this rollback is t_{cp}. Note that we have two parameters associated with rollback – t_r and t_{cp}. The difference is that t_r is the time required when both the modules are restored to the state saved by the modules in their respective stable storage, while t_{cp} is the time required when the state of the two modules is made consistent with the checkpoint saved by one of the two modules. In some implementations, t_r and t_{cp} could very well be equal.

Table 3 summarizes the actions taken in the above four situations. As shown in Section VI, concurrent retry can achieve lower average task completion time with lower variance by avoiding rollback for the most likely fault scenarios.

Table 3: Actions required in various situations

Situation	Concurrent retry	Rollback
(A)	No	No
(B)	Yes	No
(C)	Yes	Yes
(D)	Yes	Yes

It may be noted that, although the above discussion pertains to a duplex system and a spare module, this spare may not be used to convert the duplex system into a triple-modular redundant (TMR) system, as the spare is shared by many duplex systems. When a spare is shared, a duplex system utilizes the spare only when one of its modules fails, unlike a TMR system where three modules are used at *all* times.

A. Optimizations

To reduce t_{pr}, the time required for initiating concurrent retry, a spare should be designated for each task. Once the spare is designated, the executable code for the task can be sent to the spare when the task starts executing rather than when a fault is detected. Similarly, the checkpointed state may also be sent to the spare immediately after the duplex system takes a checkpoint rather than sending the state after a fault is detected. (This is analogous to the backup process approach used in Tandem systems [5]).

In step 1 of concurrent retry, instead of storing the entire checkpoints, just the signatures of checkpoints CP_{jA} and CP_{jB} may be saved on the stable storage. This scheme requires fewer checkpoints to be stored simultaneously. However, with this modification, in situation (D), the system will have to rollback to checkpoint CP_{j-1}.

If the number of spares available is more than one, then concurrent retry can be attempted simultaneously on multiple spare modules. This can significantly increase the likelihood of success of concurrent retry by tolerating multiple failures. The proposed mechanism can be extended to tolerate multiple simultaneous failures without the overhead of retry.

B. Permanent Faults

The scheme described above can also locate permanent faults. Observe that in each of situations (B) through (D) above, it is either possible to locate the faulty module or one can determine the modules that may be suspected to be faulty. For instance, in situation (B) a faulty module can be correctly identified, while in situation (C) any of the three modules (A, B and S) may be faulty. If any particular module is determined faulty or suspected to be faulty too many times within a short interval of time, then the module may be assumed to be permanently faulty and replaced.

For example, Figure 9 shows a scenario in which module B has developed a permanent fault. Module B is determined as faulty in two consecutive concurrent retries. In this case, B may be assumed to be permanently faulty and replaced.

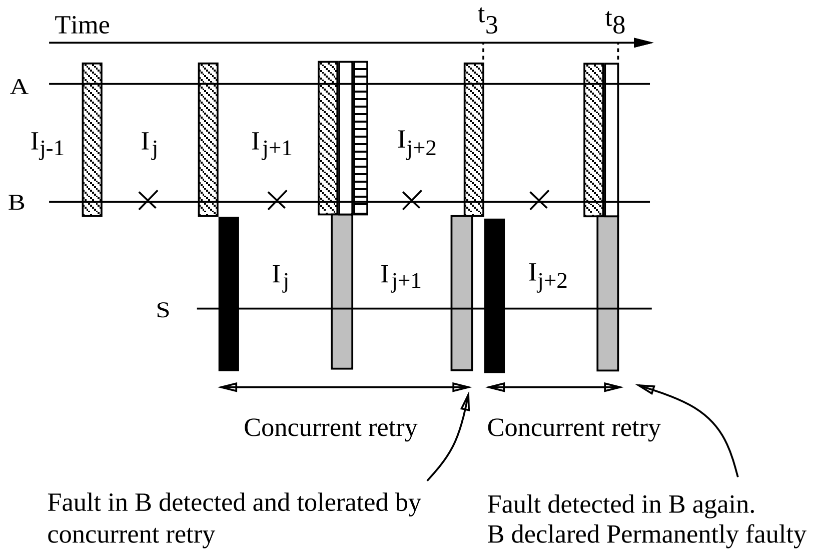
Time
t_3
t_8
A
B
S
I_{j-1}
I_j
I_{j+1}
I_{j+2}
I_j
I_{j+1}
I_{j+2}
Concurrent retry
Concurrent retry
Fault in B detected and tolerated by concurrent retry
Fault detected in B again. B declared Permanently faulty

Figure 9: Tolerating a permanent fault: An example

VI. Performance of the RFCS Scheme

As seen above, both transient and permanent as well as single and multiple faults are considered. The proposed recovery technique considers all possible fault scenarios; however, the analysis given here is for transient faults. For the analytical model developed here it is assumed that failures of any two modules are independent. Occurrence of a transient failure of a module is assumed to be a Poisson process with failure rate λ. The processing modules are assumed to be prone to transient faults during all operations including checkpointing and retry. Only the operation of making the state of a processing module consistent with a previously saved checkpoint is assumed to be performed reliably as in any checkpointing scheme. This operation can be made robust in practice by detecting failures by comparing the restored checkpoint and the restored module state after state restoration. If an error is detected then this process is repeated and the state is restored again from the checkpoint.

In our analysis, we make a simplifying assumption that the time required to rollback (t_r) is equal to the time required for initiating a restart (t_s). The analysis without the simplifying assumption is very similar to the analysis presented here, only somewhat more tedious. The notations used here are summarized below. Some of the notations were introduced in earlier sections.

T_u = total useful execution time of a task

t_u	$=$	T_u/n, where
n	$=$	number of equidistant checkpoints
τ_k	$=$	time required to execute last k checkpoint intervals τ_n is the time required to complete the task. After the task completes the first interval of execution, time required to complete the remaining task is τ_{n-1}. For other $k \geq 1$, τ_k is defined similarly. Also, $\tau_0 = 0$.
$\overline{\tau_k}$	$=$	expected (average) value of τ_k
$\overline{\tau_{n\|f}}$	$=$	expected completion time of a task given that at least one failure occurred during task execution.
v_k	$=$	variance of $\tau_k \ = \ \overline{\tau_k^2} - (\overline{\tau_k})^2$
$F_k(t)$	$=$	Cumulative Distribution Function (CDF) of $\tau_k \ = \ Prob(\tau_k \leq t)$
t_{ch}	$=$	time required to checkpoint the two modules in a duplex system. t_{ch} includes the time required to compare the two checkpoints.
T	$=$	$t_u + t_{ch}$. With no failures, task completion time is nT.
t_r	$=$	time required to rollback.
t_{cp}	$=$	time required to rollback to a previous state of one of the modules.
t_{cc}	$=$	time required for comparing state of the spare with checkpoints of the processing modules in a duplex system. We assume that $t_{cc} \leq t_{cp} + t_{ch}$.
t_{pr}	$=$	time required to initiate a concurrent retry.
t_w	$=$	$max\ (t_{pr} + t_{cc} - t_{ch},\ 0)$. Idle time.

The quantities of interest are:

- $\overline{\tau_{n|f}}$. In the absence of failures, RFCS and ROLLBACK schemes perform identically; $\overline{\tau_{n|f}}$ is a good measure of how a scheme performs when failures occur.

- Average task completion time ($\overline{\tau_n}$).

- Variance (v_n) of the task completion time.

- CDF ($F_n(t)$) of the task completion time.

When only one interval remains to be executed after a fault is detected by checkpoint comparison (as in situations (B) through (D)), concurrent retry does not result in early task completion compared to the ROLLBACK scheme. Therefore, our analysis assumes that concurrent retry is initiated only when the number of checkpoint intervals remaining to be executed after fault detection is at least two. If the number of intervals remaining is 0 or 1 then the duplex system is rolled back to the previous checkpoint (no concurrent retry is attempted).

Let p_A through p_D be the likelihood of occurrence of situations (A) through (D), respectively, enumerated in Section V. From the discussion in Section V and the fault model presented earlier, the following expressions are obtained.

$$\begin{aligned}
p_A &= \text{Prob(A and B are fault-free in interval } I_j) \;=\; e^{-2\lambda T} \\
p_B &= \text{Prob(B faulty in } I_j\text{, A and S fault-free in } I_j \text{ and } I_{j+1}) \\
&\quad + \text{Prob(A faulty in } I_j\text{, B and S fault-free in } I_j \text{ and } I_{j+1}) \\
&= 2\,(1-e^{-\lambda T})\; e^{-\lambda T}\; e^{-\lambda(T+t_{pr}+2t_u+2t_{cc})} \\
p_C &= \text{Prob(A and B faulty in } I_j) + \text{Prob(A or B (not both) faulty in } I_j \text{ and } S \text{ faulty in } I_j) \\
&= (1-e^{-\lambda T})^2 \;+\; 2\,(1-e^{-\lambda T})\; e^{-\lambda T}(1-e^{-\lambda(t_{pr}+t_u+t_{cc})}) \\
p_D &= \text{Prob(B faulty in } I_j\text{, A and S fault-free in } I_j\text{, A and/or S faulty in } I_{j+1}) \\
&\quad + \text{Prob(A faulty in } I_j\text{, B and S fault-free in } I_j\text{, B and/or S faulty in } I_{j+1}) \\
&= 2\,(1-e^{-\lambda T})\; e^{-\lambda T}\; e^{-\lambda(t_{pr}+t_u+t_{cc})}\;(1-e^{-\lambda(T+t_u+t_{cc})})
\end{aligned}$$

Also, let $p_{roll} = 1 - p_A = 1 - e^{-2\lambda T}$. Note that $p_A + p_B + p_C + p_D = 1$ (as should be expected). Let $t_A = T$, $t_B = 2T + t_w + t_{cp}$, $t_C = 2T + t_w + t_r$, $t_D = 2T + t_w + t_u + t_{cc} + t_{cp}$ and $t_{roll} = T + t_r$. Now, we obtain recursions for $\overline{\tau_k}$ and $F_k(t)$. Recall that if a fault occurs in the last two intervals, the system is rolled back. Therefore,

$$\overline{\tau_1} \;=\; p_A\, t_A \;+ p_{roll}\,(\overline{\tau_1} + t_{roll}) \;\text{ and }\; \overline{\tau_2} = 2\,\overline{\tau_1} \tag{1}$$

If a fault occurs in any interval other than the last two, then concurrent retry is performed. Therefore, when $k \geq 3$, from Figures 5 through 8, the following recursions are obtained.

$$\overline{\tau_k} \;=\; p_A\,(\overline{\tau_{k-1}} + t_A) + p_B(\overline{\tau_{k-2}} + t_B) + p_C(\overline{\tau_k} + t_C) + p_D(\overline{\tau_{k-1}} + t_D) \tag{2}$$

and

$$F_k(t) \;=\; p_A F_{k-1}(t-t_A) + p_B F_{k-2}(t-t_B) + p_C F_k(t-t_C) + p_D F_{k-1}(t-t_D) \qquad (3)$$

Starting with the above recursions, the following expressions can be obtained for $n > 2$ [12, 13]. [1]

$$\overline{\tau_n} \;=\; \frac{q_B}{1+q_B}\left(\begin{array}{l} (q_A t_A + q_B t_B + q_C t_C + q_D t_D)\ \left((n-2)q_B^{-1} + \frac{1-(-q_B)^{n-2}}{1+q_B}\right) \\ \\ +\ \overline{\tau_1}\ (q_B^{-1} + (-q_B)^{n-1})\ +\ (\overline{\tau_2} - q_{AD}\ \overline{\tau_1})(q_B^{-1} + (-q_B)^{n-2}) \end{array}\right) \qquad (4)$$

$$v_n \;=\; \frac{q_B}{1+q_B}\left(\begin{array}{l} 2\ q_C\ t_C\ \sum_{i=3}^{n}\left(\overline{\tau_i}\ \left[q_B^{-1} + (-q_B)^{n-i}\right]\right) \\ \\ +2\ (q_A\ t_A + q_D\ t_D)\ \sum_{i=2}^{n-1}\left(\overline{\tau_i}\ \left[q_B^{-1} + (-q_B)^{n-1-i}\right]\right) \\ \\ +2\ q_B\ t_B\ \sum_{i=1}^{n-2}\left(\overline{\tau_i}\ \left[q_B^{-1} + (-q_B)^{n-2-i}\right]\right) \\ \\ +(q_A\ t_A^2\ +\ q_B\ t_B^2\ +\ q_C\ t_C^2\ +\ q_D\ t_D^2)\left((n-2)q_B^{-1}\ +\ \frac{1-(-q_B)^{n-2}}{1+q_B}\right) \\ \\ +S_1(q_B^{-1} + (-q_B)^{n-1})\ +\ (S_2 - q_{AD}S_1)\ (q_B^{-1} + (-q_B)^{n-2}) \end{array}\right) - (\overline{\tau_n})^2 \qquad (5)$$

where

$$\begin{array}{lll} q_X = p_X/(1-p_C), \ \text{for } X = A,B,C,D, & & q_{AD} = q_A + q_D, \\ \overline{\tau_1} = \frac{T+t_r}{e^{-2\lambda T}} - t_r, & & \overline{\tau_2} = 2\ \overline{\tau_1}, \\ v_1 = (T+t_r)^2\ \frac{1-e^{-2\lambda T}}{e^{-4\lambda T}}, & & v_2 = 2\ v_1, \\ S_1 = v_1 + (\overline{\tau_1})^2 \ \text{and} & & S_2 = v_2 + (\overline{\tau_2})^2. \end{array}$$

Also,

$$\overline{\tau_{n|f}} \;=\; \frac{\overline{\tau_n} - p_A^n\ n\ T}{1 - p_A^n}. \qquad (6)$$

Analysis of the ROLLBACK scheme

[1]When $n \le 2$, the RFCS scheme is identical to the ROLLBACK scheme.

Table 4: Parameters for task 1

T_u	t_{ch}	t_r	t_s	t_{cc}	t_{cp}	t_{pr}
50	0.50	0.30	0.30	0.70	0.30	0.40

To compare the performance of the RFCS scheme with the performance of the ROLLBACK scheme, the following expressions for the mean completion time and its variance for the ROLLBACK scheme are obtained [12, 13].

$$\overline{\tau_n} = n\left(\frac{T+t_r}{e^{-2\lambda T}} - t_r\right) \quad \text{and} \quad v_n = n\,(T+t_r)^2\,\frac{1-e^{-2\lambda T}}{e^{-4\lambda T}} \tag{7}$$

A. Performance Comparison

Performance of RFCS scheme is compared with the ROLLBACK scheme.[2] Parameters for a hypothetical task named task 1 are listed in Table 4. Task 1 is used to compare performance of RFCS and ROLLBACK schemes. The results presented here for task 1 are also valid over a wide range of task parameters. For brevity, we have chosen only one set of parameter values.

Comparison of $\overline{\tau_{n|f}}$

Recall that $\overline{\tau_{n|f}}$ is the expected task completion time given that at least one failure occurs during the execution of the task. Figure 10 compares $\overline{\tau_{n|f}}$ for the RFCS and ROLLBACK schemes. Observe that for the RFCS scheme, $\overline{\tau_{n|f}}$ is closer to nT as compared to the ROLLBACK scheme. This is essentially because the RFCS scheme tries to avoid rollback even in the presence of a fault, and therefore completes the task in about the same time as a fault-free execution. Define

$$g(\text{RFCS}) = \frac{\overline{\tau_{n|f}}(\text{ROLLBACK}) - \overline{\tau_{n|f}}(\text{RFCS})}{(T_u/n)}.$$

g is called the "relative gain" in $\overline{\tau_{n|f}}$ achieved by the RFCS scheme with respect to the ROLLBACK scheme. In Table 5 relative gains for the RFCS scheme are listed for various

[2]The ROLLBACK scheme was presented in Section IV.

values of n and λ. Observe that the performance of the RFCS scheme remains better over a wide range of failure rate λ. Table 5 lists the relative gain for $\lambda = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$. However, to minimize the number of graphs in the paper, in most of the following discussion, λ is assumed to be 10^{-3}. Similar results can be obtained for other values of λ as well.

Table 5: Relative gain achieved by the RFCS scheme

	n								
λ	3	4	5	6	7	8	10	12	14
10^{-3}	.325	.488	.590	.660	.710	.747	.800	.834	.858
10^{-6}	.331	.495	.594	.658	.704	.738	.784	.813	.833
10^{-9}	.331	.496	.594	.658	.704	.738	.784	.813	.833
10^{-12}	.331	.496	.594	.658	.704	.738	.784	.813	.833

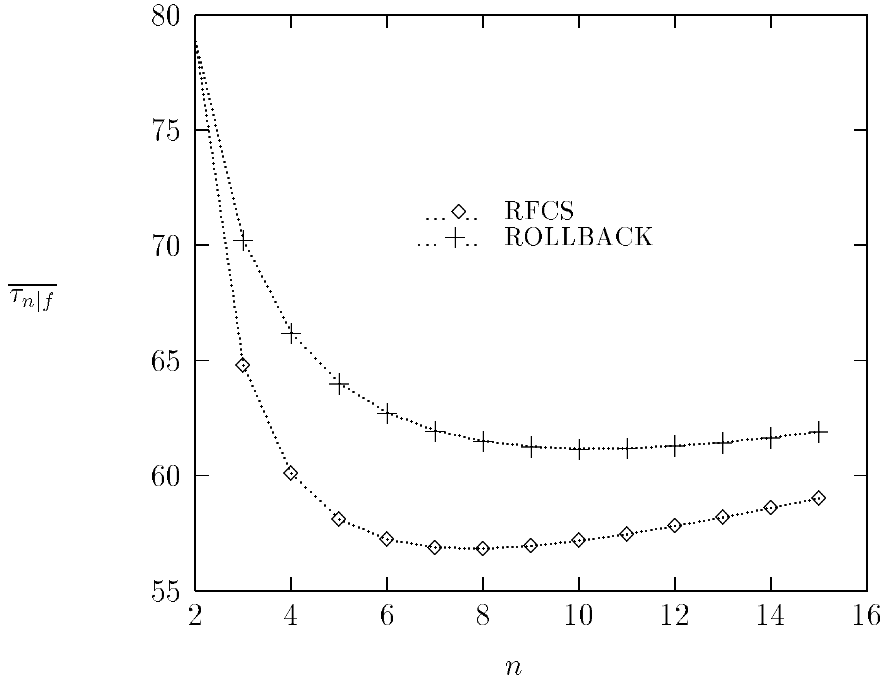

Figure 10: $\overline{\tau_{n|f}}$ for task 1 with $\lambda = 10^{-3}$

Mean and variance comparison

In Figure 11 variance v_n is plotted versus the mean completion time $\overline{\tau_n}$ for the example task. Each point on the mean-variance plot corresponds to a specific number of checkpoints. By varying the number of checkpoints, different means and variances can be achieved. Observe that for any mean and variance pair achieved using the ROLLBACK scheme, a pair with lower mean and variance can be achieved using the RFCS scheme. For example, in Figure 11, observe that if ROLLBACK scheme with $n = 6$ is used, then one may use the RFCS scheme with $n = 5$, 6 or 7 and achieve lower mean completion time with lower variance. Also, in general, the RFCS scheme can achieve a lower minimum average task completion time as compared to the ROLLBACK scheme.

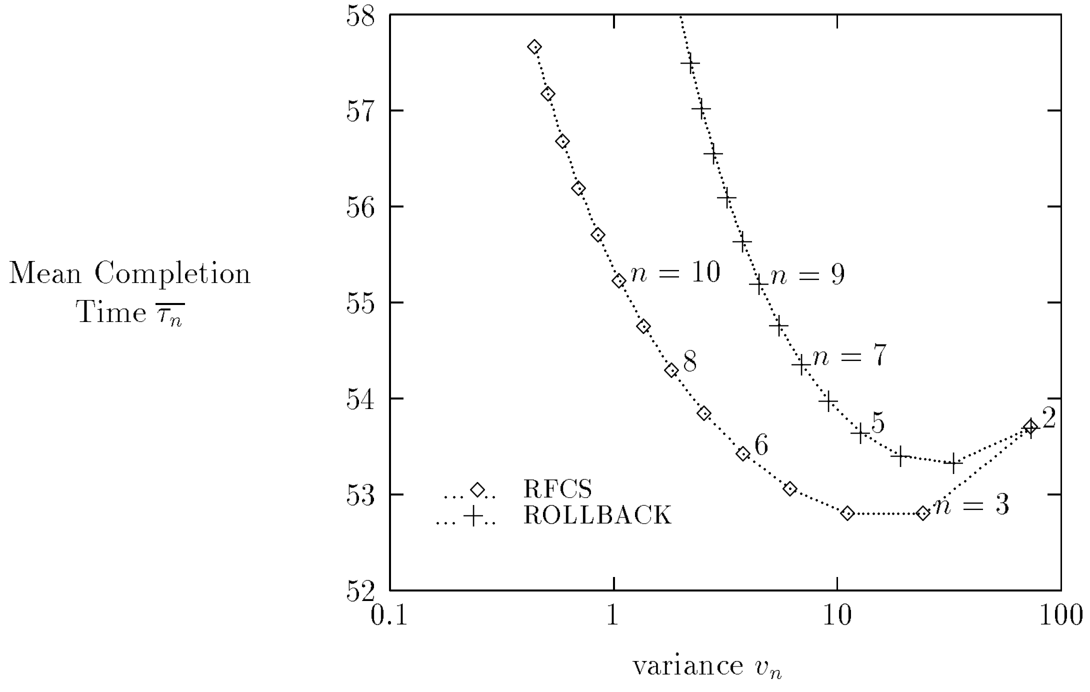

Figure 11: Mean completion time versus variance for task 1 with $\lambda = 10^{-3}$

When the failure rate is low, the mean completion time is very close to the minimum possible completion time nT, as failures occur infrequently. In such a situation one may still use a larger number of checkpoints than the number that minimizes the average completion time so as to reduce its variance. In Figure 11 for instance, for ROLLBACK scheme, $\overline{\tau_n}$ is minimized with $n = 3$. One may still use 10 checkpoints as the variance achieved with 10 checkpoints is lower (specifically, the variance is 3.76 with the mean being 55.64). In such a situation, the concurrent retry scheme is useful to further reduce the variance while keeping the mean low. RFCS scheme with 10 checkpoints achieves variance 1.06 with the mean being 55.22 – lower mean with lower variance as compared to the ROLLBACK scheme (see Figure 11).

Comparison of the CDF

The cumulative distribution function (CDF) of the completion time τ_n is useful to determine the percentage of jobs that finish by a given deadline. If the deadline requires that the task be completed within t_d time units after it starts execution, then $(1 - F_n(t_d))$ is the probability that the deadline is missed. Figure 12 plots $(1 - F_n(t))$ for task 1. Comparison of the plots for RFCS and ROLLBACK schemes indicates that the likelihood that a job will miss a tight deadline is lower with the RFCS scheme as compared to the ROLLBACK scheme.

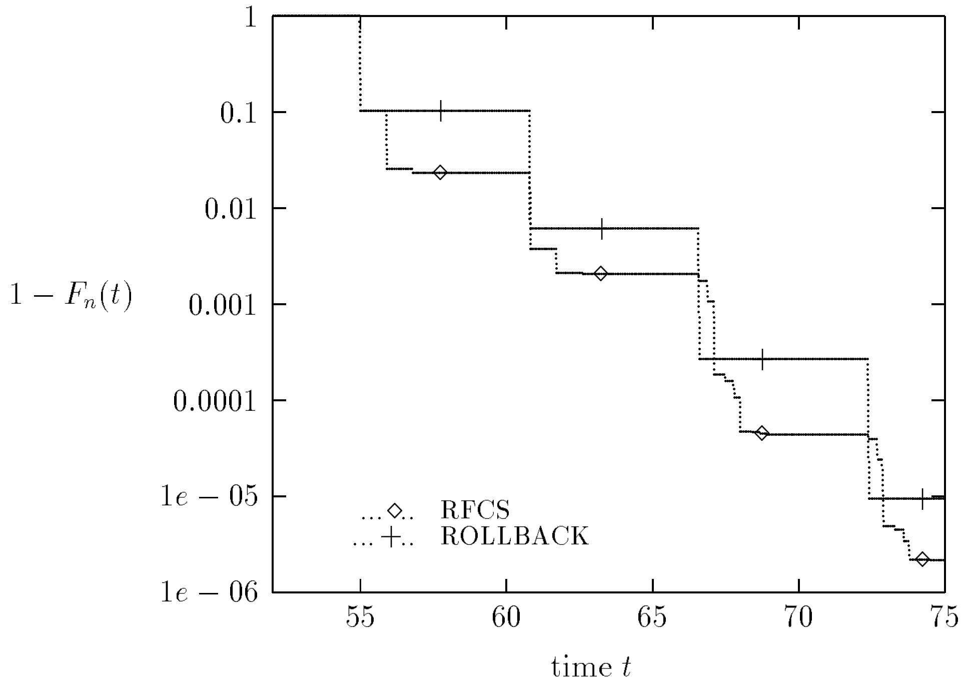

Figure 12: $(1 - F_n(t))$ versus t for task 1 with $n = 10$ and $\lambda = 10^{-3}$.

In Figure 12, observe that the ROLLBACK scheme performs better than the RFCS scheme when t_d is in a small interval around $t = 67$. The reason is that when a rollback occurs in the concurrent retry scheme, the overhead is larger compared to the ROLLBACK scheme. In spite of this, the mean and variance achieved with the RFCS scheme are lower because the RFCS scheme results in a rollback only when multiple modules fail within a short interval of time; the likelihood of such multiple faults is much smaller than a single fault.

VII. Spare Utilization

The analysis in Section VI assumed that a spare is available for concurrent retry whenever needed. When many duplex systems share a small number of spares for concurrent retry, a spare may not be available for concurrent retry if it is busy performing a retry for some other duplex system. When a failure occurs, if a spare is not available, the duplex system rolls back. When more than one duplex system shares a spare, spare availability perceived by any duplex systems is less than 1. The earlier analysis for $\overline{\tau_n}$ and v_n is valid if average spare utilization U is small. Table 6 enumerates the length of time for which the spare is used in various situations described in earlier sections.

Table 6: Length of spare use in various situations

Situation	Spare Use
(A)	$s_A = 0$
(B)	$s_B = t_{pr} + 2t_u + 2t_{cc}$
(C)	$s_C = t_{pr} + t_u + t_{cc}$
(D)	$s_D = t_{pr} + 2t_u + 2t_{cc}$

The following closed form expressions for utilization U of a spare by a single duplex system can be obtained for $n > 2$ [12, 13].

$$U = \frac{\left(\frac{q_B}{1+q_B}\right)\ (q_B\, s_B + q_C\, s_C + q_D\, s_D)\ \left((n-2)q_B^{-1} + \frac{1-(-q_B)^{n-2}}{1+q_B}\right)}{\overline{\tau_n}} \tag{8}$$

Table 7 lists spare utilization U for the RFCS scheme. Observe that the average spare utilization is quite low and decreases as the number of checkpoints (n) increases or as λ decreases. If the failure rate is very high and the checkpoint interval is large, then the likelihood that a module fails in any checkpoint interval would be high, resulting in a high spare utilization. Such a situation may be avoided by taking checkpoints more frequently.

Table 7: Spare utilization by a single duplex system with task 1

λ	n	U(RFCS)
10^{-3}	4	0.02549
	8	0.02085
	10	0.01844
	16	0.01386

λ	n	U(RFCS)
10^{-6}	4	2.6×10^{-5}
	8	2.1×10^{-5}
	10	1.8×10^{-5}
	16	1.4×10^{-6}

A. Multiple duplex systems

A multiprocessor system with a single spare module shared by up to six duplex systems was simulated. The simulation assumed that all the duplex systems execute task 1 repeatedly. λ was chosen to be 10^{-3}.

The system is simulated for 10^{10} time units with an event-driven simulator developed in language C. Table 8 lists the mean completion time $\overline{\tau_n}$, variance v_n, and spare utilization obtained by simulation. D is the number of duplex systems that share a single spare.

Table 8: Simulation results for $n = 10$ and $\lambda = 10^{-3}$: D duplexes sharing a single spare

D	$\overline{\tau_n}$	v_n	U
1	55.22	1.06	0.0184
2	55.23	1.10	0.0362
3	55.23	1.15	0.0533
4	55.24	1.20	0.0699
5	55.25	1.24	0.0859
6	55.25	1.28	0.1013

Observe that even when many duplex systems share a single spare, the spare utilization is quite low. Also, when $D > 1$, mean task completion time $\overline{\tau_n}$ and variance v_n achieved by the RFCS scheme remain better compared to the mean (55.64) and variance (3.76) achieved by the ROLLBACK scheme.

VIII. Implementation Issues

Stable Storage The performance of the RFSC scheme depends on the ability to take checkpoints efficiently. The checkpointing operation requires that the state of the two replicas be compared and the state saved on a stable storage. Although the conventional mirrored-disk stable storage may serve the purpose, special purpose hardware can improve the performance significantly. For instance, the architecture of Figure 1 facilitates fast checkpointing if the stable storage is implemented similar to the "fast stable storage" proposed by Banatre et al. [2]. The architecture in Figure 1 is similar to an architecture presented in [2]. The checkpointing operation with this architecture can be performed as follows: (a) the two modules in the duplex system store their state in the respective fast stable storages, (b) The fast stable storage sends the signature of the checkpoints to the checkpoint processor, (c) the checkpoint processor compares the signatures to detect any failures. Thus, this architecture can reduce the checkpointing time by minimizing the time required to save the state in stable storage and also the time required to compare the two states (only signatures need be sent over the network).

Another possibility is to make each of the SS modules in Figure 1 self-checking, instead of stable. It is cheaper and easier to make a memory module self-checking (as compared to stable). This organization is a subject of future research. If the SS modules are self-checking, then the SS modules can be used as a fast temporary storage for the checkpoints. In this organization, the processors would save their state in the SS modules which would then asynchronously download the checkpoints into a stable storage such as a mirrored disk. A failure of an SS module, before the state is downloaded into the stable storage, will result in a rollback of the duplex system.

Stable Storage Size The proposed scheme requires five process images to be stored on a stable storage when a failure occurs. This requirement is larger than in traditional duplex and triple modular redundant systems. The proposed RFCS approach achieves improved performance at a higher stable storage cost. When the checkpoint size is very large, the increase in the stable storage cost may be a constraint in implementing the proposed approach.

As pointed out earlier, when a write-back cache memory represents the volatile state and the main memory is stable (e.g., similar to Sequoia architecture [3]), the volatile storage (VS) block in a processing module in Figure 1 represents the write-back cache and the stable storage (SS) block represents the stable main memory. In this case, the size of the checkpoint is determined by the number of dirty cache blocks. The checkpoint size in this system is likely to be much smaller (than a system where entire memory needs to be checkpointed), making it more practical to use the roll-forward checkpoiting scheme.

Equidistant Checkpointing The discussion in the paper assumed that all the checkpoint intervals are of identical length. There are two aspects of this issue. (a) The proposed scheme can also be used when the checkpoints are not equidistant. An adaptive scheme suggests itself – concurrent retry should be performed only if the length of the interval in which the failure is detected is at least t_l for a given t_l, otherwise the system should be rolled back. Essentially, when the overhead of performing a concurrent retry is *not* small compared to the length of the faulty checkpoint interval, concurrent retry should not be performed. (b) Although it may not be possible to make all checkpoint interval lengths exactly identical, it is possible to insert checkpoints in the executable code such that the interval lengths are approximately equal. For example, [7] presents a compiler-driven approach for this purpose. Such an approach is adequate for achieving performance improvements using the proposed scheme.

Checkpoint Processor The existence of a reliable checkpoint processor is necessary to coordinate the proposed scheme. Two approaches may be used to achieve this. One approach is to implement a reliable checkpoint processor using masking redundancy and ensure that the likelihood of failure of the checkpoint processor is much smaller as compared to other components in the system. The other approach is to distribute the functionality of the checkpoint processor into multiple checkpoint processors, each being self-checking. These checkpoint processors must collectively coordinate the RFCS scheme. As the function of the checkpoint processor is quite simple, it should be possible to make it self-checking without exorbitant overhead.

IX. Communicating Processes

The discussion in the paper assumed that the processes executing on different duplex systems do not communicate with each other. In this section we argue that if the processes communicate via message passing, then the roll-forward recovery scheme may result in better or comparable performance as a rollback scheme. However, in this case, the RFCS scheme must be combined with message logging. (For event-driven processes, input events should be logged.) With single faults, no rollback is necessary even if processes communicate by message passing. This itself can be quite useful in an environment of communicating processes because recovery using coordinated checkpoints may need to be invoked rarely. Rollback to coordinated checkpoints is only required when there are multiple failures.

When processes communicate with each other via message passing and each process is duplicated, to protect from an arbitrary failure of a process, it is necessary to use a Byzantine agreement algorithm with authenticated messages. Provided at most one sender or receiver replica is faulty, it is possible to design an agreement protocol whereby each replica of a receiver process will either obtain the correct message or detect failure of a sender process replica [14]. Additionally, it is possible to ensure that the fault-free replica of a process will detect the failure of the other replica before the effect of the failure is propagated to other processes [14]. We consider single fault situations only. In the following we omit the details of the agreement protocol. It is assumed that messages are being logged for the purpose of recovery.

When using RFCS scheme for communicating processes, when a spare re-executes a checkpoint interval, appropriate messages logged on the stable storage should be sent to the spare, to allow it to reach the correct state. Other than this modification, the RFCS scheme described earlier can be used as such for communicating processes.

Figure 13 illustrates a scenario where concurrent retry can successfully identify the failure of a process and other processes continue execution without any performance penalty. P_1 and P_2 are replicas of process P (they form a duplex system) and Q_1 and Q_2 are replicas of process Q. P_1 failed at time t_1 but sent the correct message M to process Q, and then sent an incorrect message R'. Failure of P_1 is not detected until message R' is sent. Process Q assumes that message M received from P is error-free, this assumption is correct provided at most one of P_1 and P_2 is faulty (or the probability of two faulty replicas sending same

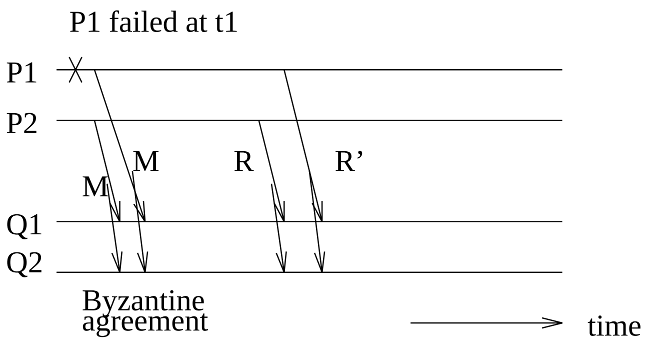

Figure 13: RFCS scheme and communicating processes

erroneous message is small). Failure of P_1 is detected when it sends message R' and P_2 sends message R. However, it is not known which of P_1 and P_2 has failed. Concurrent retry can be used to determine which of P_1 and P_2 is faulty. Two cases arise:

- During concurrent retry, no process blocks waiting for a message from process P: In this case, there will be no loss of performance in spite of failure. If a rollback scheme were used, there would be performance penalty (for process P) due to the single failure.
- Some processes block waiting for a message from process P: In this case, the performance penalty is no worse that that for the rollback scheme. When the rollback scheme is used, each process blocked on P must wait for P to recover from the failure. For both rollback and roll-forward schemes, the duration for which such processes are blocked is approximately equal to the duration from the previous checkpoint of P till the time when failure of P_1 was detected.

When multiple failures occur, the performance penalty could be larger than the rollback scheme. From the analysis in Section VI it is apparent that the impact of multiple failures on the average performance is much smaller than single failures. Therefore, we conjecture that roll-forward scheme will perform well in the environment of communicating processes also. Further research is needed to verify this conjecture.

X. Further Work

The discussion in this paper implicitly assumed that a module fault is detected only by comparing the state of the two modules in a duplex system. However, in reality, some of the faults in a module can be detected by the error detection mechanisms built into a processing module. The fault coverage, say c, of such mechanisms is typically non-zero but less than perfect. The faults that escape detection by the built-in detection mechanism are detected by comparing the state of the two modules in a duplex system at each checkpoint. For the sake of simplicity, the discussion here assumed that coverage c is 0. However, when $0 < c < 1$, two roll-forward checkpointing schemes can be obtained (similar to the RFCS scheme presented here). The two schemes differ primarily in their treatment of a fault situation where, in a checkpoint interval, one of the modules has a fault that is detected by the error detection mechanism built into the module. Two actions are possible in such a scenario which leads to two different roll-forward schemes [12]:

- One option is to assume that the other module is fault-free, and copy the state of this module to the faulty module (which had a detected failure).
- The other option is to *not* assume that the other module is fault-free. Instead, a concurrent retry is performed to achieve recovery.

Note that the first of the above two schemes results in an unreliable outcome, if in a checkpoint interval, one module has a failure detected by its built-in detection mechanism and the other module has an undetected failure. Therefore, in general, the first scheme achieves a lower reliability as compared to the second scheme. However, the first scheme has a better performance as compared to the second scheme. Also, note that both the schemes perform better than rollback schemes with comparable reliabilities. A detailed analysis of these two schemes can be found in [12].

XI. Conclusion

In this paper, a fault-tolerant multiprocessor environment wherein each task is executed simultaneously on two processing modules is considered. A pool of a small number of nondedicated spares is assumed available. A pair of processing modules performing the same task forms a duplex system. A scheme is proposed to improve the performance of such duplex systems. In the proposed scheme, at each checkpoint the states of the two processing modules executing the task are compared for detection of faults. If a fault is detected, instead of usual rollback, the proposed concurrent retry mechanism is used for identification of the faulty module. The concurrent retry mechanism uses a nondedicated spare to perform recovery. The scheme is named Roll-Forward Checkpointing Scheme (RFCS).

RFCS scheme provides a mechanism for identifying the faulty module and recovering, in most likely cases, without the overhead of rollback. For this purpose, a small number of spares is shared by many duplex systems in the multiprocessor. The proposed scheme achieves a lower average execution time with a lower variance as compared to the rollback scheme. It is demonstrated that the proposed scheme increases the likelihood that a task will complete within a tight deadline in spite of transient failures. Analytical and simulation results are obtained to demonstrate the performance improvement achieved by the proposed RFCS scheme.

Acknowledgements

We would like to thank the referees for their comments. Thanks are due to A. Finn for carefully reading an earlier version of this paper. We also thank N. Bowen, C. Krishna and A. Singh for helpful discussions.

References

[1] P. Agrawal, "Fault tolerance in multiprocessor systems without dedicated redundancy," *IEEE Trans. Computers*, vol. 37, pp. 358–362, March 1988.

[2] J. P. Banatre, M. Banatre, and G. Muller, "Ensuring data security and integrity with a fast stable storage," in *Proc.* 4th *Int'l Conf. Data Eng.*, pp. 285–293, 1988.

[3] P. A. Bernstein, "Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing," *Computer*, pp. 37–45, February 1988.

[4] Y. Deswarte, "A high safety multi-processor architecture," in *Digest of papers: The* 6th *Int. Symp. Fault-Tolerant Comp.*, pp. 171–175, 1976.

[5] C. I. Dimmer, "The Tandem non-stop system," in *Resilient Computing Systems* (T. Anderson, ed.), John Wiley & Sons, 1985.

[6] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Symposium on Reliable Distributed Systems*, 1992.

[7] C.-C. J. Li and W. K. Fuchs, "Catch – compiler assisted techniques for checkpointing," in *Digest of papers: The* 20^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 74–81, 1990.

[8] J. Long, W. K. Fuchs, and J. A. Abraham, "Forward recovery using checkpointing in parallel systems," in *Proc. Int. Conf. Parallel Proc.*, pp. 272–275, August 1990.

[9] D. K. Pradhan, "Redundancy schemes for recovery," Tech. Rep. TR-89-CSE-16, ECE Department, University of Massachusetts, 1989.

[10] Sequoia Systems, "The Series 400," Product information.

[11] Tandem Computers Inc., "NonStop Cyclone/R System," Product information.

[12] N. H. Vaidya, *Low-Cost Schemes for Fault Tolerance.* PhD thesis, University of Massachusetts-Amherst, February 1993.

[13] N. H. Vaidya and D. K. Pradhan, "Concurrent retry with nondedicated spares: A fault-tolerant checkpointing scheme without rollback," Tech. Rep. TR-91-CSE-23, ECE Department, University of Massachusetts, October 1991.

[14] N. H. Vaidya and D. K. Pradhan, "A fault tolerance scheme for a system of duplicated communicating processes," in *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pp. 98–104, July 1992.