Responsive Roll-Forward Recovery in Embedded Real-Time Systems

Jie Xu and Brian Randell

Department of Computing Science

University of Newcastle upon Tyne, Newcastle upon Tyne, UK

ABSTRACT

Roll-forward checkpointing schemes [Long *et al.* 1990; Pradhan and Vaidya 1992] are developed in order to avoid rollback in the presence of independent faults and increase the possibility that a task completes within a tight deadline. Despite of the adoption of roll-forward recovery, these schemes are not necessarily appropriate for time-critical applications because interactions with the external environment and communications between processes must be deferred during checkpoint validation steps (typically, two checkpoint intervals) until the fault-free processors are identified. The deadlines on providing services may thus be violated. In this paper we present and discuss two alternative roll-forward recovery schemes, especially for time-critical and interaction-intensive applications, that deliver correct, timely results even when checkpoint validation is required.

Key Words — Checkpoint validation, dynamic redundancy, embedded real-time systems, forward error recovery, timing constraints.

1 Introduction

Embedded real-time systems for critical applications, such as avionics, nuclear power plants, process control and patient life-support monitoring, often interact with the external environment under strict dependability and timeliness requirements. Such a system may fail to operate correctly either due to errors in hardware and software or due to violation of timing constraints imposed by its environment. Fault tolerance in time-critical systems can thus be considered as the ability of the system to deliver correct *and* timely results despite the occurrence of faults [Jahanian 1994]. However, prevision of fault tolerance and adherence to timing requirements in these systems may to some extent conflict with each other — actions taken to achieve fault tolerance may adversely affect the ability of a system to meet its timing constraints. For example, the traditional Checkpointing and Rollback technique will improve reliability but may increase the likelihood of missing a deadline. Approaches that address system fault tolerance without carefully considering real-time aspects are neither practical nor appropriate for time-critical applications. Advanced techniques must explicitly combine both fault tolerance and timing requirements.

Roll-forward checkpointing schemes (RFCS) [Long *et al.* 1990; Pradhan and Vaidya 1992] are developed to avoid "roll-back" to the previous checkpoints in the presence of faults. Ideally, a system will return to a normal state after roll-forward recovery and thereby deliver correct and timely services. This seems to solve the problem of violating real-time constraints. However, after a careful examination of these RFCS schemes, we determined they are not generally applicable to time-critical systems since they still have the potential to cause delay in delivering results. In a RFCS scheme, whenever a fault occurs, two validation steps (usually two checkpoint intervals) must be taken. Since no processor can be identified as fault-free before the completion of the validation, neither the delivery of results nor communication with other processes can be allowed. This is quite questionable — because faults may occur at any point during the execution of a process. During the two checkpoint intervals for validation, the process may need to interact with the external environment or to communicate with the processes executed on other processors. The delay in response and communication may not be tolerable in hard real-time systems where a missed deadline can be potentially as disastrous as a system crash or an incorrect output.

In this paper we examine the issues involved in the use of roll-forward error recovery in critical, real-time systems, and describe two improved roll-forward recovery schemes that are capable of ensuring timely services and communications even when faults are detected and validation steps are required.

2 Duplex Systems, Error Recovery and Timing Constraints

Our major discussion here focuses on duplex systems because they can achieve performance comparable to TMR using less redundancy and have been widely used in many commercially available systems. Some commercial systems use the Checkpointing and Rollback Recovery (CRR) technique to achieve fault tolerance, such as the Tandem Non Stop system [Gray and Reuter 1993] and the Sequoia multiprocessor [Bernstein 1988]. CRR can decrease the mean execution time of a task by reducing the time spent in retrying the task [Chandy and Ramamoorthy 1972]. (When a fault is detected, the task is retried only from the last checkpoint rather than from the beginning.) The distance of rollback is normally limited to a checkpoint interval of computation. However, the CRR scheme is not very suitable for time-critical applications since it has poor predictability of task completion time — as a result of rollback, a checkpoint interval will be lost once a fault is detected and more intervals may be missed due to the occurrence of multiple faults. Hence, avoiding rollback is crucial for real-time systems with high dependability and tight timeliness requirements.

The use of redundant processors can avoid rollback should an independent, hardware-related fault occur. Two independent processors may be used to duplicate a task and to compare each checkpoint. A checkpoint mismatch leads to a validation step. During the validation step, the two processors continue execution while a third spare processor retries the task to determine which of the diverged processors is fault-free. (It is assumed that the spare processor is shared by multiple duplex systems and may not be used to form a TMR system with a duplex system.) Based on this concept, two similar schemes have been developed. In the look-ahead scheme [Long *et al.* 1990; 1991], the processors duplicating the task are themselves duplicated during the validation step. Five processors are then required for a period of one checkpoint interval. In the second scheme [Pradhan and Vaidya 1992; 1994a; 1994b], the diverged processors are not duplicated and just three processors are used for the duration of two checkpoint intervals, as shown in Figure 1. At the end of the second checkpoint interval, the checkpoint state of the spare processor is compared with those of processors P1 and P2 where the fault was detected. The processor P2 will be identified as faulty after this comparison. A recovery action will then be taken by copying the current state of P1 to the faulty processor P2.

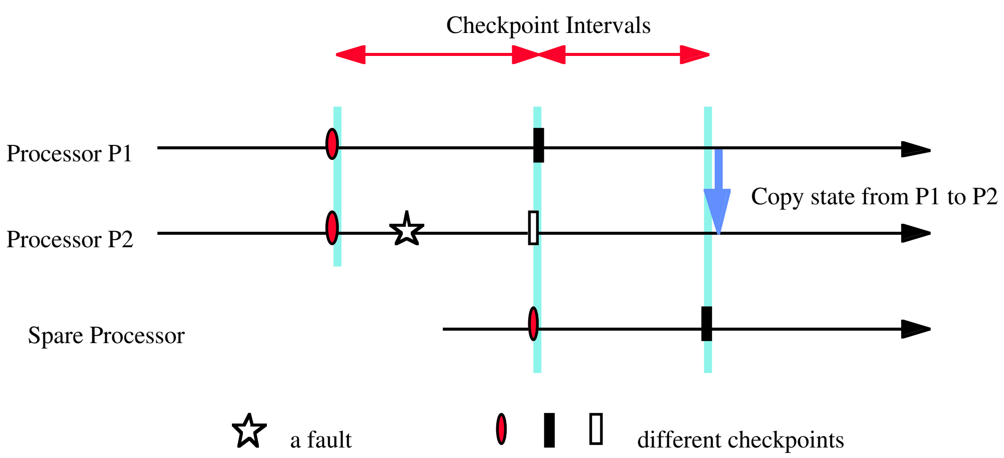

Fig. 1. Roll-forward checkpointing scheme [Pradhan and Vaidya 1992].

The major advantage of the above two schemes is that they have a lower average execution time than the CRR scheme, thereby having higher predictability of task completion time. However, these schemes must retain any results that are to be delivered to the external environment (and possible communications with other tasks) until completion of the validation steps. This could be a significant drawback for time-critical applications where the amount of slack available may be small and the delay of even one or two checkpoint intervals (typically measured in tens seconds [Long *et al.* 1991]) may not be acceptable. Moreover, although it has not been stated explicitly, the advantage of these RFCS schemes holds only under the implicit assumptions that a task (or a process) does not communicate with the others and it delivers the required services only at the end of the last checkpoint interval. These schemes may have to roll back when a fault occurs during the last two checkpoint intervals [Pradhan and Vaidya 1994a]. Unfortunately, these assumptions are often inappropriate for typical real-time applications. Tasks or processes in these applications are usually communication- and interaction-intensive. It is not practical to insert several checkpoint intervals between the interactions of a task with the external environment. Even just using the time duration between two consecutive interactions as a checkpoint interval is unacceptable for some applications; the overhead of execution time can reach hundreds percent in the examples described by [Silva *et al.* 1994]. Clearly, alternative or improved schemes are needed.

It is important to note that the standard TMR and NMR techniques for forward recovery can more effectively resolve the issue of missing a deadline (if the time overhead for managing redundancy can be tolerated). However, the duplex-based schemes have advantage in efficiency — they require less redundancy (fewer processors on average) than TMR. Our objective here is to develop alternative schemes that are capable of i) avoiding roll-back, ii) avoiding delay in interaction with the external environment and in message communication even when a fault occurs, and iii) achieving the same level of fault tolerance as TMR but using less average redundancy.

3 System Model and Basic Assumptions

An embedded real-time system can be partitioned into three components: controlled objects (i.e. sensors and actuators), the computer system, and the operator [Jahanian 1994] (see Figure 2(a)). The controlled objects and the operator constitute the environment of the computer system. The system interacts frequently with its environment, reacting to stimuli of external events and producing timely results. Time in such a system is a scarce resource and the inability of the system to meet the specified timing constraints will be viewed as a failure. In reality, most real-time computer systems are distributed ones, as shown in Figure 2(b), consisting of a set of processors interconnected by a real-time communication subsystem. In order to achieve fault tolerance, some form of redundancy, such as replicated hardware and recovery mechanisms, must be used to detect and to recover from errors. It is assumed that these processors are organized as a pool of active processors and a relatively small number of passive spare processors (or active processors with some spare processing capacity [Dahbura *et al.* 1989]).

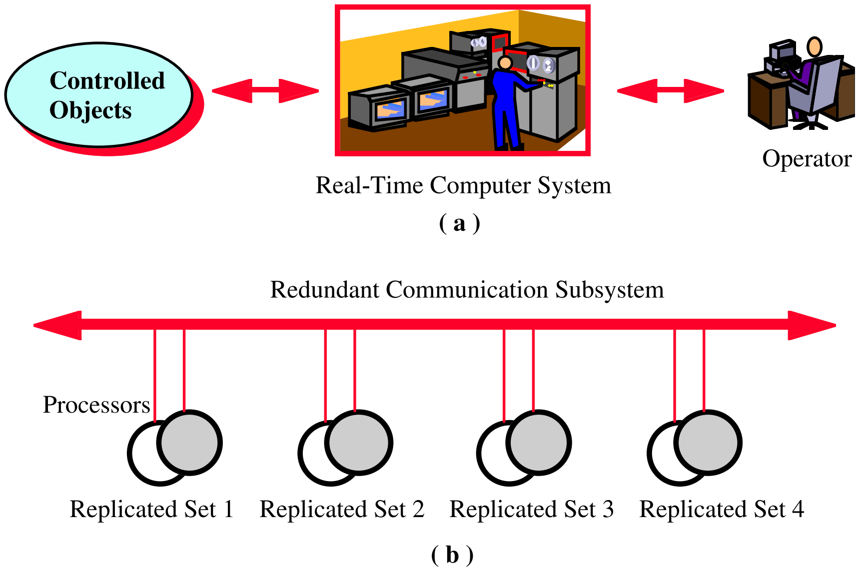

Fig. 2. (a) An embedded system (b) a distributed real-time computer system.

We assume a system organization for checkpointing and state comparison similar to that used in [Pradhan and Vaidya 1994b] where further details of the architecture can be found. Each processor in Figure 1(b) is supposed to contain volatile storage and to be able to access stable storage. A special processor, called the Checkpoint Processor, is used to compare checkpoints and perform recovery when the need arises.

A task, or a process, is an independent computation and may be further divided into a group of related sub-tasks. The execution of a task is partitioned into a series of sequential sub-computations by checkpoints. The time intervals between checkpoints are denoted as I_1, I_2, ..., I_n. A checkpoint interval is said to be *atomic* if and only if none of results of the task is visible to the other tasks and to the external environment during and at the end of that interval; the interval is said to be *interactive* otherwise. Conceptually, for any $i = 1, 2, ..., n$, the checkpoint interval I_i could be interactive (note that the interval I_n must be interactive at least). Our major interest here is in the outputs of a task. However, it is important to note that incoming data and messages from the external environment or from the other tasks can be used during any checkpoint intervals but must also be recorded on the stable storage in case they need to be reused for the purpose of validation.

Checkpoints can be inserted in a task manually. The interval time may be either roughly equal or adjustable. Research on the optimal placement of checkpoints has shown that the optimal checkpoint interval is typically large (in tens seconds) [Long *et al.* 1991]. For a given task with a given placement of checkpoints, we assume that atomic and interactive intervals can be unambiguously identified by examining whether the task needs to output information during (and at the end of) each checkpoint interval. In order to guarantee correct, timely interaction, when a task enters an interactive interval, spare redundancy must be exploited even though a fault may not take place during it. More precisely, a spare processor must be engaged before the current interactive interval. Thus, a set of particular checkpoint intervals must be further identified. A checkpoint interval I_i is said to be *pre-interactive* if and only if the checkpoint interval I_{i+1} is interactive.

To demonstrate the major advantage of our schemes and for purpose of comparison with the two existing FRCS schemes, we will mainly concentrate on single, independent faults. A fault may be detected through result comparison before output and through checkpoint comparison at the end of the checkpoint interval or be detected by the error detection mechanism within the processor and by an acceptance test at the application level. Some multiple faults in consecutive checkpoint intervals may defeat any duplex roll-forward strategies (including our proposed schemes) and thus require roll-back. TMR-type schemes could avoid roll-back for such multiple faults, but they still have to roll back if multiple faults occur in the same checkpoint interval. To simplify the description of our schemes, we will first assume that there is a single fault during any two consecutive intervals and then discuss the ability of our schemes to treat multiple fault situations. In addition, for related faults, the assumption that two faulty duplicated processors will always generate distinct checkpoints is not generally applicable; we will briefly address this issue later.

4 Responsive Roll-Forward Error Recovery

Time-critical applications are essentially *responsive* — the computer system is required to interact with its environments frequently, and in a timely manner. We describe in this section two roll-forward recovery schemes with different error detection mechanisms, which are responsive in the sense that correct and timely interactions are ensured even in the presence of faults.

4.1 Roll-Forward Recovery with Dynamic Replication Checks

This Roll-Forward Recovery scheme uses Replication (or comparison) Checks to detect errors. We call this scheme RFR-RC. The execution of a real-time task in the RFR-RC scheme is organized and dynamically adjusted with respect to different types of checkpoint intervals.

Atomic Checkpoint Intervals: For any atomic interval, a task is executed on just two independent processors, say P_1 and P_2. At every checkpoint, the duplicated task records its state in the stable storage and the state is also sent to the Checkpoint Processor. At the end of the checkpoint interval, the Checkpoint Processor compares the two states from the processors duplicating the task. If the two checkpoint states match, the checkpoint is committed and both the processors continue executions into the next checkpoint interval. If a mismatch is detected, a validation step starts. During validation, processors P_1 and P_2 continue execution while a spare processor S is engaged to retry the last checkpoint interval using the previously committed checkpoint. After the spare S completes, its state is compared with the previous states of P_1 and P_2. The faulty processor will be identified after this comparison. The state of the faulty processor can then be made identical to that of the other processor. Both processors duplicating the task should now be in the correct state. Under our assumption of single independent faults, a further validation step will not be required (see Figure 3).

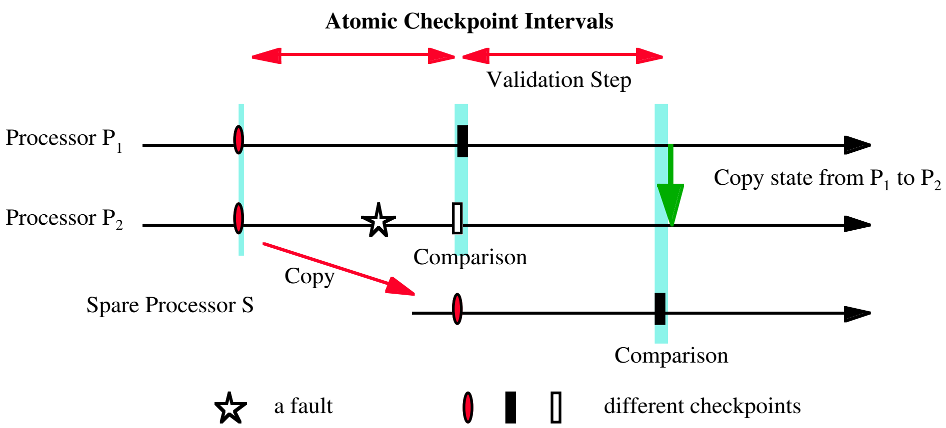

Fig. 3. Roll-forward recovery during atomic checkpoint intervals.

Interactive Checkpoint Intervals: For any interactive interval, a spare processor is always required, in addition to the two processors duplicating the task. During the interactive interval, any results are voted before being delivered to the external environment, or to the other tasks. At the end of this interactive interval, the states of the three processors are compared. If a fault occurs during the interval, it will be detected by the replication check. The state of the faulty processor will then be replaced using the correct state of the other processors. Whenever the next checkpoint interval is atomic, the spare processors will be returned to the pool of spares which are shared by other tasks. Figure 4 illustrates the task execution and roll-forward recovery during interactive checkpoint intervals.

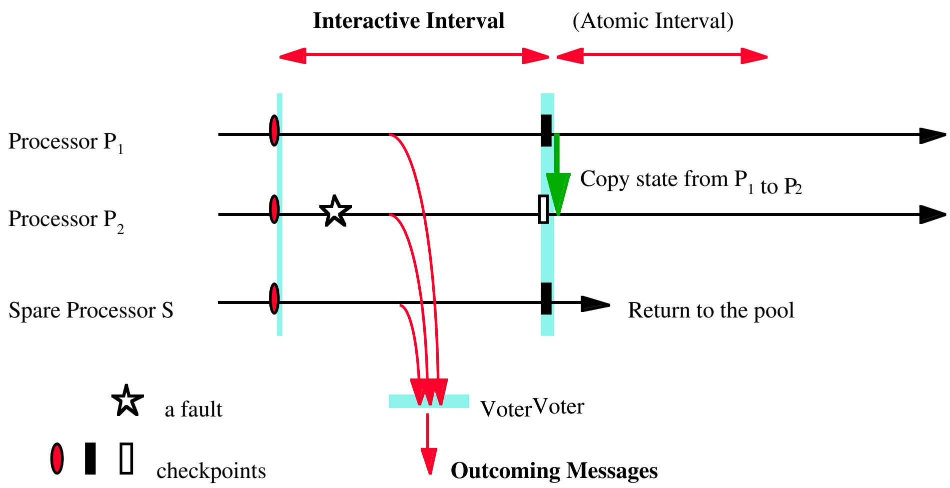

Fig. 4. Task execution and roll-forward recovery during interactive intervals.

Pre-Interactive Checkpoint Intervals: For any pre-interactive interval, in addition to the two processors duplicating the task, a spare processor may be required unless it has been obtained in the last interval. There are two cases to consider:

1) A spare processor has not been involved during the last checkpoint interval. In this case, if the states of two processors P_1 and P_2 are identical at the end of the last interval, a spare processor will copy the current state of one of P_1 and P_2 starting with the same execution as P_1 and P_2. However, if the states of P_1 and P_2 disagree, the spare will have to use the start state of the last checkpoint interval and retry the

execution of that interval. The retry and recovery operation is essentially same as the roll-forward recovery action taken for atomic checkpoint intervals (see Figure 3).

2) A spare processor has been involved during the last checkpoint interval. When the three processors P_1, P_2 and S performed the same execution during the last checkpoint interval, a faulty processor may be detected at the end of that interval. The state of faulty processor will be replaced by the correct one and the three processors will then continue the same execution during the pre-interactive interval, as shown in Figure 5(a). If the spare S retried a different execution during the last interval, one of P_1 and P_2 will be identified as fault-free at the end of that interval. The state of the fault-free processor will be copied to the other processors and the three processors will then start with the same execution (see Figure 5(b)).

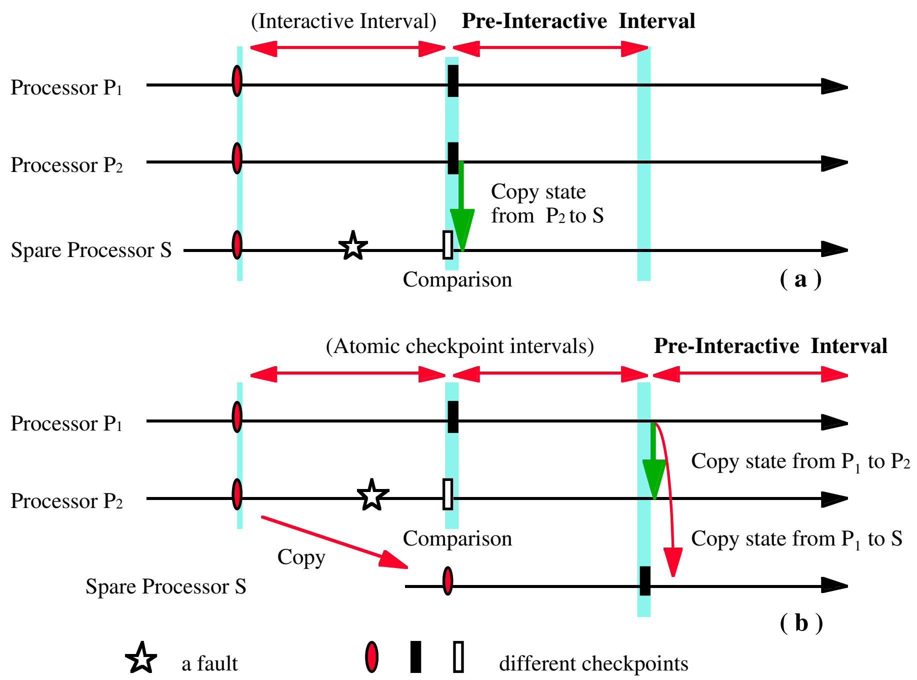

Fig. 5. Roll-forward recovery during the pre-interactive checkpoint interval.

Remarks

The RFR-RC scheme supports both roll-forward recovery and timely response. Under a common assumption (also made for other duplex roll-forward schemes) that multiple faults do not occur within two consecutive checkpoint intervals, RFR-RC can entirely avoid rollback and ensure interactions without delay. This is a significant advantage over the existing duplex schemes. Depending upon specific applications, our scheme may use more redundancy than the previously developed duplex RFCS schemes, but it generally requires less redundancy than the standard TMR technique. For environments where a spare processor is not available, the following scheme exploits self-checks for error detection, and in many cases is effective in avoiding roll-back.

4.2 Roll-Forward Recovery with Behaviour-Based Checks

In some time-critical environments the amount of redundancy can be an important concern due to considerations of cost, power, weight and volume etc. The use of spare processors may in fact be impractical. In order to avoid roll-back and ensure timely response, self-checks must be employed to identify the faulty processors. These error self-detection methods are behaviour-based ones, such as illegal instruction detection, memory protection, control-flow monitoring, watchdog timers and assertions (or acceptance tests [Randell 1975]). Although such fault self-detection has imperfect error coverage and cannot detect certain types of faults, recent research and experiments have shown that their coverage can be high enough to enable us to consider behaviour-based methods as an independent detection mechanism. (A detailed discussion on behaviour-based methods is given in [Silva *et al.* 1994].)

Our second scheme, called RFR-BC (Roll-Forward Recovery with Behaviour-based Checks), uses a process pair approach similar to that used in Tandem [Gray and Reuter 1993], without having to roll back in the presence of faults. Whenever an active process or task fails, the "hot" spare task becomes active and delivers the required services. Any messages to the other tasks or to the external environment are not deferred, but they are checked by acceptance tests before being sent [Randell and Xu 1994]. The states of two processes at the end of a checkpoint interval are checked as well and the state passing the test is committed. Checkpointing here is used for fault detection and fault identification as well as roll-forward error recovery. Once a faulty processor is located, its state is made identical to the checkpoint state of the fault-free processor. Therefore, at the beginning of the next checkpoint interval, both processors will be in the correct state. Figure 6 illustrates how this RFR-BC scheme works.

Remarks

The idea behind RFR-BC is obvious and has been used in several commercial systems such as STRATUS (see the related chapter in [Lee and Anderson 1990] and some interesting discussion in [Laprie *et al.* 1987]) where each task is actually executed simultaneously by four CPUs with comparison checks. Comparison-based checks may detect some errors which escape from the behaviour-based checks. However, RFR-BC uses much less of the system resources and would be more practical for certain application environments. Of course, result and checkpoint comparison can be further incorporated into RFR-BC for the purpose of error detection.

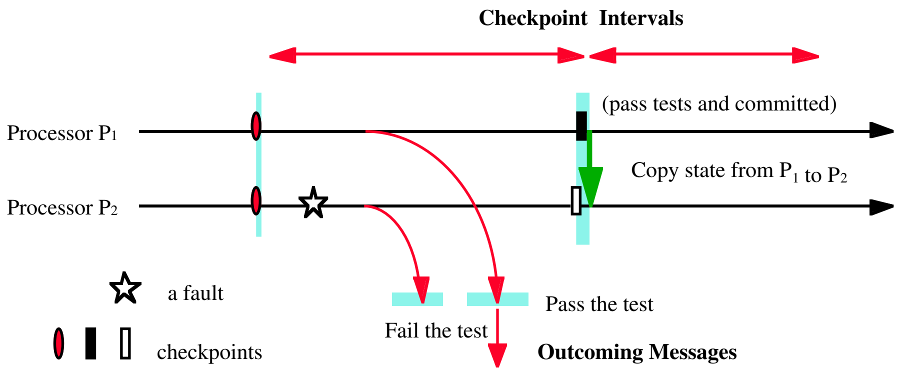

Fig. 6. Roll-forward recovery in the RFR-BC scheme.

5 Analysis and Evaluation

In this section we present a brief analysis and compare our schemes with two existing RFCS schemes [Long *et al.* 1990; Pradhan and Vaidya 1992], with respect to the level of fault tolerance, execution time (and its predictability) and processor costs. Some results obtained in experimental setups are used to support our discussion.

5.1 Level of Fault Tolerance

Our discussion mainly focuses on transient faults. Several fault situations are considered.

A. An Independent Fault in Any Two Consecutive Checkpoint Intervals: In this fault situation, our RFR-RC scheme can avoid any roll-back and ensure timely interactions. If such a fault can be caught by its behaviour-based detection mechanism, our RFR-BC scheme has the same characteristics. However, the previously developed RFCS schemes have to roll back or defer interactions whenever such a fault occurs during an interactive or pre-interactive checkpoint interval. Since the deferment is visible to the other tasks and to the outside environment, a failure of violating the response deadline could be caused.

B. Two Correlated Faults in Two Consecutive Checkpoint Intervals: Due to the use of three processors, the RFR-RC scheme can avoid roll-back and provide timely response if two correlated faults occur in non atomic checkpoint intervals; a failure and the deferment of interactions may take place otherwise. The RFR-BC scheme is quite effective in tolerating such correlated faults, provided that the faults can be detected by its behaviour-based detection mechanism. The RFCS schemes will suffer from the same problem as that in the situation A when the correlated faults appear within an interactive or pre-interactive interval. However, during atomic checkpoint intervals, in some cases they may tolerate the faults and in other cases they may not (i.e. require roll-back). (A detailed discussion can be found in [Pradhan and Vaidya 1994b].)

C. Two Correlated Faults in a Checkpoint Interval: Though such a fault situation may be detected, they cannot be effectively tolerated by our schemes and the other duplex schemes because interactions must be deferred. Furthermore, it is possible that two faulty processors produce the identical incorrect results or the same erroneous checkpoints. This delicate situation could still be detected by our schemes so as to prevent the erroneous information from having impact on the other tasks and the outside environment. For example, the RFR-RC scheme may be designed to demand strict agreement between the three processors replicating the task. This type of correlated faults may also be detected by self-checks used in the RFR-BC scheme. However, the previous duplex schemes cannot detect such a fault situation and would thus deliver (undetected) erroneous services.

Table 1 summarizes the fault tolerance capability of these duplex schemes. In our opinion, the use of roll-back (for some fault situations) in a roll-forward recovery scheme doesn't seem to be appropriate. For a given application that tolerates occasional roll-back or loss of checkpoint intervals, the standard CRR schemes are more cost-effective and practical [Chandy and Ramamoorthy 1972]. When such loss of time is not acceptable, roll-back must be avoided *completely* using either more redundancy or more complicated self-diagnosis programs. This is a major reason why we have omitted the discussion on possibilities of using roll-back to treat some fault situations. We have also omitted the discussion on permanent fault situations which will generally require the reconfiguration of the hardware system.

Table 1 Fault tolerance capability of various duplex schemes

Schemes / Fault Situations	RFR-RC Scheme	RFR-BC Scheme	RFCS Schemes
Situation A	Avoid roll-back and ensure timely response	Avoid roll-back and ensure timely response if faults are self-detected	Defer interactions & communications and may require roll-back
Situation B	Avoid roll-back and ensure timely response if B occurs during pre- or interactive checkpoint intervals; a failure or deferment takes place otherwise	Avoid roll-back and ensure timely response if faults are self-detected	Defer interactions & communications and may require roll-back if B occurs during pre- or interactive checkpoint intervals; Avoid roll-back in some cases if B occurs during certain atomic intervals
Situation C	Cannot be tolerated but can be detected even in some delicate cases, e.g. two same erroneous checkpoints	Cannot be tolerated but could be self-detected even in some delicate cases, e.g. two same erroneous checkpoints	Cannot be tolerated but can be detected if the erroneous checkpoints are different; otherwise cannot be detected

5.2 Execution Time and Its Predictability

In order to investigate the predictability of task execution time, we conduct a small experiment in a local network environment that consists of a number of Sun 3 workstations. Two versions, *a* and *b*, of a simulation program of controlling a moving object were used in our experiment, with different interaction frequencies. We only take independent faults into account. The execution time of the program is partitioned into 30 roughly equal checkpoint intervals. During the interactive intervals, this simulation program reads control information from a buffer and updates the outputs that control moves of the object. Atomic checkpoint intervals are used to perform pure computations. The version *a* is programmed to have higher precision of controlling the moves and require 11 interactive checkpoint intervals, whereas the version *b* has only 3 interactive intervals with lower control precision. Independent faults are injected into both interactive and atomic intervals to examine the effect of possible deferments. Table 2 gives the information about the task execution with or without the use of roll-forward recovery schemes.

Table 2 Execution time of the simulation program with or without roll-forward recovery

Program version	Normal Execution Time in msec.	Checkpointing & Comparison	Checkpointing & Self-detection	No. of Intervals/ Interactive Intervals	Checkpoint Interval
Version a	840576	330	589	30 / 11	~ 28419
Version b	840221	330	589	30 / 3	~ 28407

(a)

Version a				
Schemes	Execution Time in msec.	Extra Execution Time	Response Delay if a fault occurs	Overhead of Execution
RFR-RC	850928	10352	~ 0	1.23 %
RFR-BC	858084	17508	~ 0	2.08 %
RFCS	850479	9903	28353 ~ 56705	1.17 ~ 6.75 %

(b)

Version b				
Schemes	Execution Time in msec.	Extra Execution Time	Response Delay if a fault occurs	Overhead of Execution
RFR-RC	850326	10105	~ 0	1.20 %
RFR-BC	857719	17498	~ 0	2.08 %
RFCS	850225	10004	0 ~ 28369	1.19 ~ 3.38 %

(c)

Despite of some practical implications, the data listed above are not used to derive definitive conclusions. For this special example, the overhead caused by roll-forward recovery schemes is relatively small. Since the version *a* requires intensive interactions, the probability that a given fault occurs within a non atomic interval is very high (more than 0.6). The delay in interactions, caused by the RFCS scheme when a fault is injected into an interactive interval, is about one to two checkpoint intervals. However, such delay may be avoided in the execution of the version *b* when an injected fault is detected and recovered within several consecutive atomic intervals. This experiment shows an important fact that our schemes (both RFR-RC and RFR-BC) have well predictability of task complete time, being independent of whether a fault occurs in an atomic checkpoint interval, whereas the execution time (or response time) of the RFCS scheme varies depending upon the fault distribution (i.e. upon whether they take place within atomic checkpoint intervals) and upon the interaction frequency of a given application.

5.3 Demands for Spare Processors

If the error coverage provided by its behaviour-based self-check mechanism is high enough for specific applications, the RFR-BC scheme can avoid roll-back without using any spare processor. The probability that the RFCS scheme requests a spare processor is related to the probability that a fault is detected. In most likely cases, only two processors (or a duplex system) are involved.

Our RFR-RC scheme, however, ensures timely response at the price of using more redundancy as compared with the previous RFCS schemes. In our programming experiment, the interaction-intensive version *a* needs a spare processor during 67% of the whole execution time. This is necessary because the application has high likelihood that a fault occurs within pre- or interactive checkpoint intervals, thereby deferring normal interactions. If an application has a relatively low frequency of interactions and communications, the demand for the spare will be reduced. For example, the version *b* of the control program decreases the use of the spare processor down to 20%.

In most practical multiprocessors and distributed systems, the requirement for system resources varies in a stochastic manner. Spare capacity is an essential demand for such a system to guarantee its performance when it is most heavily loaded. Clearly, spare capacity can be used for other purposes, such as fault detection and error recovery. An implementation of fault diagnosis strategies using the Bell Data Flow Architecture is presented in [Dahbura *et al.*

1989], which has shown that using spare capacity (rather than structuring a fixed TMR subsystem) to support fault detection and diagnosis is practically viable.

6 Conclusions

For most time-critical applications, the execution of a task consists typically of iterations of the periodic control cycle — *read inputs*, *compute*, *produce new output*. The task can be divided into several sub-tasks that communicate with each other by message passing. Because of strict timing constraints on interactions, the delay in updating the output and communicating with the other tasks may not be acceptable. We have proposed two alternative roll-forward recovery schemes with different error detection mechanisms. These schemes are shown to be able to avoid roll-back totally and to guarantee timely response in the presence of independent faults.

It is important to notice that real-time tasks can be non-deterministic and real-time data may be perishable — if retried they could produce results different from those of the first run since such executions begin at different absolute times. In our schemes, timely response even in the occurrence of faults can effectively avoid upsets in the externally visible time behaviour of a real-time system. The retry of a task on a spare processor is performed only to identify faulty processors. This process of roll-forward recovery is therefore completely transparent to the other tasks and the external environment.

Acknowledgments

This research was supported by the CEC-sponsored ESPRIT Basic Research Actions on Predictably Dependable Computing Systems (PDCS and PDCS2).

References

[Bernstein 1988] P.A. Bernstein, “Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing,” *Computer*, vol. 21, no. 2, pp.37-45, Feb. 1988.

[Chandy and Ramamoorthy 1972] K.M. Chandy and C.V. Ramamoorthy, “Rollback and recovery strategies for computer programs,” *IEEE Trans. Comput.*, vol. 21, no. 6, pp.546-556, June 1972.

[Dahbura *et al.* 1989] A.T. Dahbura, K.K. Sabnani and W.J. Hery, “Spare capacity as a means of fault detection and diagnosis in multiprocessor systems,” *IEEE Trans. Comput.*, vol. 38, no. 6, pp.881-891, June 1989.

[Gray and Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Jahanian 1994] F. Jahanian, “Fault-tolerance in embedded real-time systems,” in *Hardware and Software Architectures for Fault Tolerance*, (eds. M. Banatre and P.A. Lee), pp.237-249, 1994.

[Laprie *et al.* 1987] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle, “Hardware and software fault tolerance: Definition and analysis of architectural solutions,” in *FTCS-17*, pp.116-121, June 1987.

[Lee and Anderson 1990] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, Second Edition, Prentice-Hall, 1990.

[Long *et al.* 1990] J. Long, W.K. Fuchs and J.A. Abraham, “A forward recovery strategy using checkpointing in parallel systems,” in *Proc. Int. Conf. Parallel Processing*, pp.272-275, Aug. 1990.

[Long *et al.* 1991] J. Long, W.K. Fuchs and J.A. Abraham, “Implementing forward recovery using checkpointing in distributed systems,” in *DCCA-2*, pp.20-27, Feb. 1991.

[Pradhan and Vaidya 1992] D.K. Pradhan and N.H. Vaidya, “Roll-forward checkpointing scheme: Concurrent retry with nondedicated spares,” in *Proc. Workshop Fault-Tolerant Parallel and Distributed Systems*, pp.166-174, 1992.

[Pradhan and Vaidya 1994a] D.K. Pradhan and N.H. Vaidya, “Roll-forward and rollback recovery: Performance-reliability trade-off,” in *FTCS-24*, pp.186-195, June 1994.

[Pradhan and Vaidya 1994b] D.K. Pradhan and N.H. Vaidya, “Roll-forward checkpointing scheme: A novel fault-tolerant architecture,” *IEEE Trans. Comput.*, vol. 43, no. 10, pp.1163-1174, Oct. 1994.

[Randell 1975] B. Randell, “System structure for software fault tolerance,” *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.

[Randell and Xu 1994] B. Randell and J. Xu, “Chapter 1: The evolution of the recovery block concept,” in *Software Fault Tolerance*, (ed. M. Lyu), John Wiley & Sons Ltd, pp.1-22, Sept. 1994.

[Silva *et al.* 1994] J.G. Silva, L.M. Silva, H. Madeira and J. Bernardino, “A fault-tolerant mechanism for simple controllers,” in *EDCC-1*, pp.39-55, June 1994.