NASA-CR-195760

January 1994

UILU-ENG-94-2201
CRHC-94-01

Center for Reliable and High-Performance Computing

IN-62-CR
3765
166P

CHECKPOINT-BASED FORWARD RECOVERY USING LOOKAHEAD EXECUTION AND ROLLBACK VALIDATION IN PARALLEL AND DISTRIBUTED SYSTEMS

Junsheng Long

(NASA-CR-195760) CHECKPOINT-BASED FORWARD RECOVERY USING LOOKAHEAD EXECUTION AND ROLLBACK VALIDATION IN PARALLEL AND DISTRIBUTED SYSTEMS Ph.D. Thesis, 1992 (Illinois Univ.) 166 p

N94-29869

Unclas

G3/62 0003765

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-94-2201 (CRHC-94-01)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab
University of Illinois

6b. OFFICE SYMBOL (If applicable)
N/A

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research & NASA

6c. ADDRESS (City, State, and ZIP Code)
1101 W. Springfield Ave.
Urbana, IL 61801

7b. ADDRESS (City, State, and ZIP Code)
800 N. Quincy St.
Arlington, VA 22217

Ames Research Ctr
Moffett Field, CA

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
7a

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-91-J-1283 NASA NAG 1-613

8c. ADDRESS (City, State, and ZIP Code)
800 N. Quincy St.
Arlington, VA 22217 7b.

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) Checkpoint-Based Forward Recovery Using Lookahead Execution and Rollback Validation in Parallel and Distributed Systems

12. PERSONAL AUTHOR(S)
LONG, Junsheng

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT (Year, Month, Day)
94-01-28

15. PAGE COUNT
163

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
checkpointing, evolutionary, checkpoint placement, rollback, lookahead, distributed systems, forward recovery

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis studies a forward recovery strategy using checkpointing and optimistic execution in parallel and distributed systems. The approach uses replicated tasks executing on different processors for forward recovery and checkpoint comparison for error detection. To reduce overall redundancy, this approach employs a lower static redundancy in the common error-free situation to detect error than the standard N Module Redundancy scheme (NMR) does to mask off errors. For the rare occurrence of an error, this approach uses some extra redundancy for recovery. To reduce the run-time recovery overhead, lookahead processes are used to advance computation speculatively and a rollback process is used to produce a diagnosis for correct lookahead processes without rollback of the whole system. Both analytical and experimental evaluation have shown that this strategy can provide a nearly error-free execution time even under faults with a lower average redundancy than NMR.

(continued on back)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

CHECKPOINT-BASED FORWARD RECOVERY USING LOOKAHEAD EXECUTION AND ROLLBACK VALIDATION IN PARALLEL AND DISTRIBUTED SYSTEMS

BY

JUNSHENG LONG

B.S., Beijing University, 1982
M.S., University of Arizona, 1986
M.S., University of Arizona, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

CHECKPOINT-BASED FORWARD RECOVERY USING LOOKAHEAD EXECUTION AND ROLLBACK VALIDATION IN PARALLEL AND DISTRIBUTED SYSTEMS

Junsheng Long, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1992
Jacob A. Abraham and W. Kent Fuchs, Advisors

This thesis studies a forward recovery strategy using checkpointing and optimistic execution in parallel and distributed systems. The approach uses replicated tasks executing on different processors for forward recovery and checkpoint comparison for error detection. To reduce overall redundancy, this approach employs a lower static redundancy in the common error-free situation to detect error than the standard N Module Redundancy scheme (NMR) does to mask off errors. For the rare occurrence of an error, this approach uses some extra redundancy for recovery. To reduce the run-time recovery overhead, lookahead processes are used to advance computation speculatively and a rollback process is used to produce a diagnosis for correct lookahead processes without rollback of the whole system. Both analytical and experimental evaluation have shown that this strategy can provide a nearly error-free execution time even under faults with a lower average redundancy than NMR.

Using checkpoint comparison for error detection calls for a static checkpoint placement in user programs. Checkpoint insertions based on the system clock produce dynamic checkpoints. A compiler-enhanced polling mechanism using instruction-based time measures is utilized to insert static checkpoints into user programs automatically. The technique has been implemented in a GNU CC compiler for Sun workstations. Experiments demonstrate that the approach provides stable checkpoint intervals and reproducible checkpoint placements with performance overhead comparable to a previous compiler-assisted dynamic scheme (CATCH).

Obtaining a consistent recovery line is another issue to consider in this forward recovery strategy. Checkpointing concurrent processes independently may lead to an inconsistent recovery line that causes rollback propagations. In this thesis, an evolutionary approach to establish a consistent recovery line with low overhead is also described. This approach starts a checkpointing session by checkpointing each process locally and independently. During the checkpoint session, those local checkpoints may be updated, and this updating drives the recovery line evolve into a consistent line. Unlike the globally synchronized approach, the evolutionary approach requires no synchronization protocols to reach a consistent state for checkpointing. Unlike the communication synchronized approach, this approach avoids excessive checkpointing by providing a controllable checkpoint placement. Unlike the loosely synchronized schemes, this approach requires neither message retry nor message replay during recovery.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisors, Professors Abraham and Fuchs, for their support, patience, and guidance throughout this thesis research. It is Professor Abraham who gave me the opportunity to pursue my graduate study at Illinois. After Professor Abraham left for the University of Texas at Austin, Professor Fuchs took me under his wing.

I would also like to thank Professors Iyer, Banerjee, Wah, and Ng for serving on my committee, and all of my colleagues at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory for their friendship and assistance. In particular, I wish to thank David Blaauw, Prakash Narain, Hongchao Dong, Yi-min Wang, Bob Janssens, and Vicki McDaniel.

Finally, I would like to thank my wife, Susan, for her love, understanding and encouragement throughout my graduate study. I am also grateful to my parents for their support and guidance throughout my life.

This research was supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283, and in part by the National Aeronautics and Space Administration (NASA) under Contract NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

1.1. Error Recovery

Error recovery is to remove all errors resulting from a fault in a system. There are two basic approaches to error recovery: backward and forward. Backward recovery removes errors by restoring a previous state of a system regardless of the current state (e.g, discarding the current erroneous state altogether). The state restoration approach simulates the reversal of time during recovery. Backward techniques are suitable for the unanticipated errors, since all current errors are simply discarded by rolling back to a previously saved state. Checkpointing and rollback comprise the most common backward recovery technique in practice [1–6]. During checkpointing, the system state is saved in a *checkpoint*. During recovery, the current state is discarded and the computation is restarted from the last checkpoint. Checkpointing time and reprocessing time due to rollback are the major components of recovery overhead. The validation of an error-free state or a checkpoint is very important for a successful rollback recovery.

Forward recovery removes errors by manipulating some portion of the current (erroneous) state and generating a valid new next state. Forward recovery does not go back in time during recovery. However, this approach depends on either an error correction mechanism or an error masking mechanism. The error correction mechanism requires accurate error damage assessment and prediction, and specific knowledge about error correction for a particular system [7]. The error detection and correction codes, and the correctable data structures are examples of the error correction-based forward recovery schemes [8,9]. The error masking mechanism uses a static massive redundancy and majority voting, such as in the standard NMR (*N Module Redundacy*).

Considerable research has been devoted to checkpoint-based backward recovery schemes [1–4,7]. Few published efforts are known concerning checkpoint-based forward recovery which do not depend on a specific error correction mechanism or massive redundancy. In this thesis, such a checkpoint-based forward recovery strategy is studied for parallel and distributed systems. This approach exploits the inherent redundancy in today's performance-oriented parallel and distributed systems. It requires neither a specific knowledge about error correction nor a complete NMR for error masking. It performs checkpoint validation through checkpoint comparison.

1.2. Motivation

Fault tolerance is an integrated part for systems that require high reliability and availability. The following observations about fault-tolerant computing are made:

- Fault tolerance usually degrades the overall performance of the system. Redundancy has been known as the heart of any fault-tolerant techniques [7]. The redundancy can be either *space* redundancy which uses extra processing elements or *time* redundancy which consumes additional processing time.

- Fault-tolerant systems with specially built hardware may be costly due to the economics of scale. The advent of high-performance low-cost microprocessor and interconnection technology has helped spur the growth of parallel and distributed systems. Most of today's parallel and distributed systems employ (possibly massive) redundancy for the purpose of high performance. If this inherent hardware redundancy can be utilized for fault tolerance, these systems may support fault-tolerant operations in a cost-effective fashion.

- Not every user requires fault tolerance. Given the reliability of today's computing systems, most applications are executed in such a short time that an error occurrence is very unlikely. When designing fault-tolerant techniques, we do not want to penalize those users who do not want fault tolerance.

- The overall performance degradation can be reduced by making common cases efficient. Amdahl's law implies that any reductions in overhead from the frequent situations at the expense of increased overhead for the infrequent situations may decease the overall overhead [10].

- If a software approach can achieve the fault-tolerant objectives with an overhead comparable to the special hardware approach, then the software approach should be preferred because software is flexible to accommodate changes in technologies and can provide the desirable user transparency.

We strongly feel that the development of fault-tolerant designs should take the above observations into consideration with an emphasis on high performance and low cost. This thesis is motivated to exploit the inherent redundancy in the existing performance-oriented parallel and distributed systems for fast error recovery. Our focus is software and low overhead techniques, and their evaluations through implementation and experimentation.

1.3. Objective

The objective of this thesis is to study a forward recovery strategy using optimistic execution and rollback validation, and to examine the issues concerning this forward recovery strategy such as performance evaluation, implementation, and experimentation. In addition, we want to study the practical issues such as compiler-assisted checkpoint insertions and low-cost concurrent checkpointing.

1.3.1. Forward recovery using lookahead execution and rollback validation

Our approach to a general forward recovery in parallel and distributed systems is to combine checkpointing with the replicated tasks that execute on different processors. Optimistic execution of the replicating tasks facilitates forward recovery, while checkpoint comparison is used to detect errors in computation (checkpoint validation). Compared with the standard **NMR** approach, this strategy uses, on average, fewer processors and can provide an execution time close to that of error-free execution. If one is willing to replicate processes over different processors, our approach can be an attractive alternative to achieve forward recovery in a parallel or distributed environment. The motivations behind this forward recovery are to use existing redundancy in performance-oriented parallel and distributed systems and to reduce the overhead of the common case.

1.3.2. Implementation and experimentation in a distributed system

We want to build our forward recovery strategy in a software tool. With this tool, users can use it as they desire, without penalizing other users who do not. In addition, we want to evaluate the effectiveness and performance of our forward recovery strategy with experiments using real applications. Thus, we want to implement our strategy in a distributed system. Our approach to this implementation is to identify the basic problems of checkpointing in a real operating system and to develop a checkpoint library that separates the process information from the operating system and that can be inserted into application programs either manually or automatically by a compiler.

1.3.3. Compiler-assisted static checkpoint insertion

Our forward recovery strategy requires checkpoints to be inserted in user programs at fixed locations. Employing hardware or not, any checkpoint insertion approaches based on the system clock do not support such static checkpoint insertion. We want to develop compiler-assisted techniques to insert static checkpoints with an overhead comparable to the dynamic checkpoint insertion schemes, without requiring special hardware [11]. We want the static checkpoint insertion techniques to be implemented as an option in a real compiler and let users decide whether they want to use this option.

1.3.4. Evolutionary approach to concurrent checkpointing

Communication between concurrent processes makes checkpointing individual processes independently inadequate because this inconsistent recovery line may lead to the rollback domino effect [12]. The common concurrent checkpointing schemes often result in large checkpointing overhead due to either extra message rounds or excessive checkpointing. The global coordinated approach requires stopping execution and reaching an agreement after rounds of messages [13, 14]. The communication synchronized approach always maintains a consistent recovery line by synchronizing checkpointing with communication, and thus produces uncontrollable checkpoint placements and possibly excessive checkpointig [15–18]. The message logging-based schemes are the communication-synchronized checkpointing schemes in which message logging is used as checkpoint operations to reduce checkpointing costs. They synchronize logging with every message and maintain the past message exchange history to determine a consistent recovery line [19–26]. The loosely synchronized approach needs either message retry or message replay from message logs. We want to develop a concurrent checkpointing approach to produce controllable checkpoint intervals, with limited run-time overhead and without complex message management during recovery. We also want to apply our approach to shared-memory multiprocessor and distributed virtual memory systems to avoid excessive checkpointing caused by the current schemes.

1.4. Thesis Overview

The three specific topics studied in this thesis are: 1) a general forward recovery strategy in parallel and distributed systems, 2) compiler-assisted static checkpoint insertion, and 3) an evolutionary approach to concurrent checkpointing.

In Chapter 2, we describe a checkpoint-based forward recovery approach in parallel and distributed systems using optimistic execution. The replication of a task makes forward recovery independent of the computation since the correct next new state is obtainable from some error-free replica. Lookahead execution is scheduled to avoid reprocessing due to rollback. A rollback process serves as the diagnostic process to determine the correctly scheduled lookahead.

Chapter 3 presents an analytical evaluation of our forward recovery strategy. Two schemes derived from our recovery strategy are compared with three common recovery schemes. We investigate the impacts of checkpointing and recovery overhead on recovery performance, the impact of file servers and the optimal selection of an optimal checkpoint interval.

In Chapter 4, an experimental evaluation of one of our recovery schemes is described. This evaluation is based on an implementation in a Sun NFS network environment and a set of realistic benchmarking programs. We will also describe how to separate process information from operating system information to make a comparable and restartable checkpoint. Experiments show that the forward recovery scheme can achieve a near error-free execution time even under fault occurrences.

In Chapter 5, we describe the compiler-assisted techniques for static checkpoint insertion. These techniques use an instruction-based time measure instead of a real-time clock to space checkpoints. This insertion guarantees a static checkpoint location with respect to program execution. That is, the checkpoint locations will not change in the program execution even with the different runs. This property makes it possible to apply this technique to our forward recovery strategy which may require checkpoint comparison as the error detection means. The checkpoint insertion techniques are implemented in a GNU C compiler for SUN 3 and SPARC workstations.

Chapter 6 describes an evolutionary checkpointing for concurrent processes. With this approach, individual processes are allowed to checkpoint independently at the start of a checkpointing session. This initial recovery line may not be consistent. As the computation executes, the processes update their local checkpoints when a communication arises. This local checkpoint updating makes the inconsistent recovery line converge to a consistent one when the checkpointing session ends. We also consider the performance and application of our approach to shared-memory multiprocessor and distributed memory systems.

Finally, in Chapter 7, the results obtained in preceding chapters are summarized. The potential areas of future research are also discussed.

CHAPTER 2.

FORWARD RECOVERY USING LOOKAHEAD EXECUTION AND ROLLBACK VALIDATION

2.1. Introduction

Considerable research has been devoted to checkpoint-based backward recovery schemes [1–4, 7]. There have also been techniques proposed which combine replication with voting and checkpoint rollback recovery. The RAFT algorithm replicates the computation on two processors to achieve error detection and rollback recovery [27, 28]. If the results produced by the replicated tasks do not match, the task is executed on other processors until a pair of matched results is found. Checkpoint-based backward recovery has two drawbacks: an execution time penalty due to checkpointing and rollback, and the problem of determining if a checkpoint is error free. Although placing checkpoints optimally can reduce the execution time penalty to some extent, the computation lost by rollback is inherent [1–4]. One approach to validating a checkpoint is to validate the system state via concurrent error detection or system diagnosis, before a checkpoint is taken [29, 30]. Another is to simply keep a series of consecutive checkpoints and perform multiple rollbacks when necessary [31]. In contrast to backward recovery, forward recovery attempts to reduce the lost computation by manipulating some portion of the current state to produce an error-free new state.

However, forward recovery generally depends on accurate damage assessment, a correction mechanism, and sometimes massive redundancy (e.g., NMR) [7,8].

However, there is little published work on checkpoint-based forward recovery which does not depend on a specific error correction mechanism or massive redundancy [32]. In this chapter, we present a forward recovery strategy using checkpointing and optimistic execution for parallel and distributed systems. This approach exploits the inherent redundancy in today's performance-oriented parallel and distributed systems. It requires neither a specific knowledge about error correction in a particular computation nor a complete NMR for error masking. It uses a limited static redundancy for error-free computations, for error detection and for state information retention, and it also employs a dynamic redundancy for recovery. On average, it uses a redundancy less than its NMR counterparts.

The following two sections describe the computation and system model, and fault model used in this thesis. Section 2.4 describes our checkpoint-based forward recovery. The subsequent section discusses the design parameters of a recovery scheme derived from our basic strategy.

2.2. Computation and System Model

2.2.1. Computation and system

The system considered consists of homogeneous processing elements connected to each other and to secondary storage by a network. The processing element can be either a computer node in a distributed system or a CPU node in a multiprocessor system. The network can be a LAN for a distributed system or a general connection network for a parallel system. We assume that the necessary checkpoints are retained on a reliable secondary storage and are accessible through the interconnection network.

2.2.2. Computation task

A task is an independent computation and it can be a group of related subtasks. A task is divided into a series of sequential subcomputations by checkpoints. The execution of a subcomputation is called a computation session, while the checkpointing period is a checkpointing session. A checkpoint interval is defined as the execution time of a computation session.

A process is the task running on a processing element. A process can be replicated on different processors. This replication can be physical or logical. A physical replication is the execution of the same task on another processor, while a logical replication is the execution of different software versions or recovery blocks for the same computation on another processor. The physical replication is used to tolerate physical faults, whereas the logical is used to tolerate software faults [12,33].

2.2.3. Checkpoint

A checkpoint consists of two types of information: the current process state for process restart and the test information for process state (checkpoint) validation. They are called the state and test portions of a checkpoint, respectively. These two may or may not be separate entities within a checkpoint. If the checkpoint is the complete run-time image of the process, the test portion can be the image itself or the signature of the image. The latter is a case of separate state and test portions. Both the state and the test portions of a checkpoint are saved in the reliable secondary storage in this thesis. Each state portion in a checkpoint is a complete (consistent) recovery line if the computation is a group of concurrent processes (we will discuss this in detail in Chapter 6).

2.2.4. Checkpoint test

A test to detect an erroneous state or checkpoint can be a comparison test or a self-test. In a comparison test, the test portions of the checkpoints for the same computation are compared. If they are identical, the test indicates a valid checkpoint, and an erroneous one, otherwise. This implies that the probabilities of the two test portions being identical as a result of one or two erroneous processes are negligible. If the test portion alone can detect errors in a checkpoint, this test is said to be a self-test such as in some cases of algorithm-based fault tolerance techniques [34,35]. A majority voting test is simply an extension of the comparison test. The checkpoints generated from different processors for the same computation should produce the same test portion if a comparison test is used. Without sacrificing clarity, we often use checkpoint comparison for the comparison test of a checkpoint and a restart from a checkpoint for the restart from the state portion of a checkpoint.

2.3. Fault Model

This thesis deals with the faults that cause an error in a process and result in an erroneous checkpoint. However, faults in the processor interconnection network or the secondary storage may neither be detectable nor recoverable in our approach. To tolerate software faults, the logical replication of processes is necessary. The replication must be either an alternative version of the same task in the N-version programming or an alternative recovery block [12,33,36,37]. For physical faults, either physical or logical replication can be employed. The physical replication is a straightforward approach.

2.4. Recovery Using Optimistic Execution and Rollback Validation

Two essential features of our forward recovery strategy are lookahead execution to reduce the computation loss due to recovery and rollback validation to diagnose the correctly scheduled lookaheads. These concepts are illustrated in Figure 2.1. In the RAFT scheme, a task is replicated and executed concurrently on two different processors [27,28]. At the end of one computation session, two checkpoints are produced by the replicated process pair. A voter process compares the newly generated but uncommitted checkpoints to determine if the process state is error free. If the two checkpoints are identical, the system state is valid. Either of the checkpoints can be committed for the past computation session, and the process pair advances to the next session.

If the uncommitted checkpoints disagree, then the checkpoints contain an erroneous state. Instead of rolling back, two identical task processes are started from the uncommitted checkpoints on two additional processors. This optimistic scheduling is called lookahead execution. Meanwhile, another task process rolls back to the last committed checkpoint on a fifth processor. After a checkpoint interval, Δ, a diagnosis checkpoint is produced by the rollback (validation) process. This checkpoint is compared to the two disagreeing (uncommitted) checkpoints. If there is a match, the error-free checkpoint is identified. The process pair that was executed ahead from the disagreeing erroneous checkpoint and the rollback validation process are terminated. The correct checkpoint is then committed and

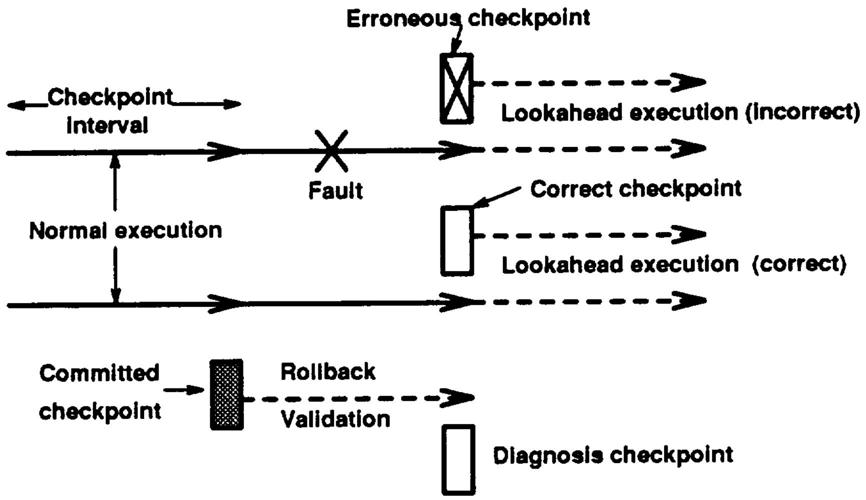

Figure 2.1. Lookahead Execution and Rollback Validation.

the incorrect checkpoints are removed. In this strategy, the two additionally scheduled lookaheads make it possible not to roll back the whole system when there is an error during lookahead executions. In this case, the lookahead pair from the newly verified checkpoint is treated as the normal pair. This pair can start a new round of lookahead and rollback validation without rolling back the whole system.

This recovery strategy is indeed a forward recovery one. In fact, the state of the computation task in the duplex system is the tuple of two individual states of the replicated process pair. If one of these two individual states is erroneous, our approach uses the redundant state (the one that is error free) to generate the next valid state by scheduling an additional process from this redundant state without reprocessing the past computation session. Therefore, this recovery strategy is forward. Clearly, this recovery strategy depends on no error correction or error masking mechanism.

Compared to the static redundancy of three processors for TMR, this strategy uses two processors for the common error-free situation and a dynamic redundancy of five for the rare occurrence of an error. The potential for forward recovery lies in the fact that there should be at least one correct process (thus, one valid checkpoint) during the normal run, since the lookahead execution from this valid checkpoint advances the computation without rollback. However, rollback may not be avoidable when the diagnosis checkpoint does not agree with either of the two uncommitted disagreeing checkpoints, since all lookahead runs may be incorrect.

2.5. Scheme Design Considerations

Our forward recovery strategy is a general approach to forward recovery using checkpointing for parallel and distributed systems. Many schemes can be derived from this general strategy. In this section, we examine the design parameters for a particular recovery scheme based on our forward recovery strategy.

2.5.1. Lookahead and rollback scheduling

With respect to lookahead and rollback scheduling, there are four parameters in designing a specific recovery scheme based on the approach described. The first is the number of replicated processes in the normal run, which we call *base (redundancy) size*. The larger the base size, the more potential there is for forward recovery, since it is likely to have an error-free checkpoint for successful lookaheads. The second is the *validation size* or the number of processes used for rollback validation; the third is the *validation depth* or the number of retries of the rollback validation process if a rollback validation fails to diagnose the disagreeing checkpoints. We can use either a larger validation size or a larger validation depth to increase the diagnosis success rate. In the case of a larger validation depth, the rollback validation success rate is increased by using time redundancy. The fourth is the *lookahead size* or the number of lookahead processes scheduled.

A forward recovery scheme is recursive if its validation depth is unlimited. In this case, the processes executed ahead may spawn their children of lookahead and validation tasks unboundedly, as the validation retries increase. This recursive scheme can maximally utilize the forward recovery capability of the lookahead execution, and rollback probability can potentially reach its lower bound. If the probability of multiple failures during a checkpoint period is very small, then it is unlikely that recursive validation and lookahead process spawning will be required. A nonrecursive scheme is an approximation of its recursive counterpart. In fact, it is the corresponding recursive scheme with all validation retries greater than the validation depth truncated.

If its base size is $2m+1$ for some integer m, a recovery scheme can also have forward recovery through error masking and majority voting. For non-$(2m+1)$ base sizes, lookahead size decides whether a scheme has forward recovery capability or not. If its lookahead size is equal to its base size, this scheme is forward recovery through optimistic execution. If the processor resource is limited, limiting the lookaheads scheduled (lookahead size < base size) leads to a graceful performance degradation with a limited forward recovery capability. At one extreme, the recovery scheme degenerates into a normal rollback scheme such as RAFT if the lookahead size is zero [27,28].

Table 2.1. Recovery Scheme Classification.

Base Size (b)	$b = 2m + 1$	forward recovery via error masking				
	$b = 2m$			Lookahead Size, l		
				$l = 0$	$0 < l < b$	$l = b$
		Validation Size	n	nonrecursive backward no lookahead	nonrecursive limited forward limited lookahead	nonrecursive forward full lookahead
			∞	recursive backward no lookahead	recursive limited forward limited lookahead	recursive forward full lookahead

The concepts of base size, validation size and depth, and lookahead size can be used to describe other recovery schemes as well. Table 2.1 presents a classification of nonerror-correction based recovery schemes. For example, a traditional triple module redundancy (TMR) scheme can be characterized with a base size of 3, rollback size and depth of zero, and a lookahead size of zero, since it schedules neither rollback validation nor lookaheads. It simply rolls back to the last committed checkpoint during recovery and uses error masking for forward recovery.

2.5.2. Test information

The test information available plays an important role in selecting the test mechanism. If the test is self-testable, we can eliminate the rollback validation in Figure 2.1 since the voter task can identify the correct process state by simply testing the checkpoint directly without using rollback validation. In addition, the lookahead execution scheduled from the erroneous checkpoint can also be eliminated. The resulting lookahead scheduling simply restarts a replicated process from the newly verified error-free checkpoint. If there is no knowledge about how to detect error based on the test portion of a checkpoint, a comparison test on the test portions of two checkpoints may be used. In this case, both rollback validation and lookahead executions are necessary.

2.5.3. State information

The state information in a checkpoint greatly affects the size of a checkpoint and thus the checkpointing overhead. The state information varies from computation to computation. If the state information spreads all over the process space, the identification and extraction of this state information can be very difficult or may be time-consuming. In this case, the complete run-time image of the process can be a good alternative as the state portion of a checkpoint. If the state information is stored only in several variables in the process space, we can extract these state variables as the state portion of a checkpoint. In recovery blocks and N-version programming, the input data can serve as the state portion of a checkpoint if we choose a recovery block or a version of algorithm implementation as a computation session. The input and output between recovery blocks or versions provide the natural boundary for checkpointing. In the algorithm-based fault-tolerant schemes, it is also possible to use the input data and intermediate results as the state information if we can decompose the whole algorithm into several subalgorithms and each subalgorithm is treated as a computation session.

2.5.4. Graceful degradation

In our recovery strategy, the lookahead scheduling may not be possible due to the limited processors available. In this case, a graceful degradation can be achieved in a natural manner in our strategy by selecting lookahead size < base size (Table 2.1). If there is only one available process for lookahead execution, we can randomly make a guess by selecting an uncommitted checkpoint for the lookahead scheduling. We will have a 50 percent chance to obtain correctly scheduled lookahead. Thus, we trade some forward recovery capability for process resource. If there is no process for lookahead, our scheme is naturally degraded into the common rollback schemes [27, 28] in which only the validation process is scheduled.

CHAPTER 3.

ANALYTICAL EVALUATION

3.1. Performance Metrics

In this thesis, we have considered two types of performance measures for a recovery scheme derived from our strategy: recovery time and recovery resource requirements. We have not considered the traditional fault-tolerant measures such as reliability and availability for two reasons. First, the recovery we have considered is per computation-based. Second, in our recovery schemes, the computation is guaranteed to finish given the available resources, such as processors.

Let T_e be the expected execution time of the computation task under consideration and T_0 the error-free execution time. The performance measures examined are

- *Number of Checkpoints*, N_c: the average number of checkpoints stored in the system or $\frac{1}{T_e}\int_0^{T_e} N_c(t)dt$. The maximal instantaneous $N_c(t)$ reflects the maximal storage requirement.

- *Number of Processors*, N_p: the average number of processors used by the system or $\frac{1}{T_e}\int_0^{T_e} N_p(t)dt$. It describes the processor redundancy required by a recovery scheme. The maximal instantaneous $N_p(t)$ reflects the maximal processor requirement.

- *Relative Execution Time*, R_e: the ratio of the expected execution time (T_e) over the error-free execution time (T_0). This measure normalizes the effect of the execution time of different computations. If R_e is close to one, the execution time will be close to the error-free execution time, demonstrating the effectiveness of forward recovery.

3.2. Basic Assumptions

In our analytical and experimental evaluation, three types of overhead are considered: checkpoint time (t_k), process restart time (t_r) and checkpoint testing time (t_t). For purposes of analysis, constant checkpoint intervals and overheads are used. Each processor has a constant probability of failure, p_f, during one computation session ($\Delta + t_k$) with or without restart and checkpoint test. This assumption implies two requirements. The first is a Poisson distribution for the failure distribution, while the second is $t_t \ll \Delta + t_k$ and $t_r \ll \Delta + t_k$ since the probability of failure over $[0, \Delta + t_k]$ is required to be equal to that over $[0, \Delta + t_k + t_t + t_r]$. The typical test time t_t and restart time t_r are in the order of a fraction of a second and the checkpoint interval Δ on the order of minutes or hours.

In order to consider the impact of the centralized file server that handles checkpoint files, we assume that t_k and t_r are approximately n-fold, when the n processes access their checkpoint files at the same time. This assumption enables us to study the impact of a file server by adjusting t_k and t_r, since both restart time t_r and checkpoint time t_k will be increased due to the file accesses to a single server. The increase in t_r and t_k may not be proportional to the number of processes that access the same file. However, a checkpoint file usually contains many blocks. A fair server policy guarantees that the n processes finish their access to the checkpoint file at approximately the same time. We also assume that checkpoint comparison is performed by a voter process on a host that can access the file system locally, and thus t_t is not changed.

3.3. Recovery Schemes Using Comparison Tests

3.3.1. Alternative recovery schemes

We examine two alternative schemes derived from our proposed recovery strategy and three other common schemes. These five recovery schemes are characterized in Table 3.1. The DMR-F-1 and DMR-F-2 are nonrecursive schemes derived from our forward recovery strategy. Their rollback validation is limited to one try with one or two rollback validation processes. The TMR-F is the common TMR forward recovery scheme using error masking and majority voting. It starts with three processes and votes on three checkpoints. If there is no matched pair, it schedules no rollback validation process and simply restarts from the last committed checkpoint.

The DMR-B-1 and DMR-B-2 are recursive rollback schemes modified from the RAFT algorithms [27,28]. Two processors are used for the normal execution. If the checkpoints

Table 3.1. Five Schemes Using Checkpoint Comparison Test.

DMR-F-1:	a nonrecursive forward recovery scheme with base size = 2, validation size = 1, validation depth = 1 and lookahead size = 2.
DMR-F-2:	a nonrecursive forward recovery scheme with a base size = 2 validation size = 2, validation depth = 1 and lookahead size = 2.
TMR-F:	a triple module redundancy forward recovery scheme using error-masking with base size = 3, validation size = 0, validation depth = 0 and lookahead size = 0.
DMR-B-1:	validation size = 1, validation depth = ∞ and lookahead size = 0.
DMR-B-2:	a recursive backward recovery scheme with base size = 2, validation size = 2, validation depth = ∞ and lookahead size = 0.

match after checkpointing, the execution advances to the next computation session. If there is no matched checkpoint pair, the computation rolls back repeatedly with one or two processes until a matched checkpoint pair is obtained. Our reason to compare our nonrecursive forward recovery schemes to the recursive rollback schemes is that the nonrecursive schemes are the first-order approximation of the corresponding recursive forward recovery ones. The recursive ones give the best performance among all their approximate derivations. That is, we compare our schemes to the corresponding rollback schemes with the best possible performance.

3.3.2. DMR-F-1: Forward recovery with one rollback validation

In this scheme, a lookahead is successful if the validation task is error free and there is a correct uncommitted checkpoint. Otherwise, a rollback is performed. The probability of a successful lookahead is

$$p_l = 2p_f(1 - p_f)^2.$$

The probability of rollback is

$$p_r \; = \; 2p_f^2(1 - p_f) \; + \; p_f^2.$$

Let there be an average of l successful lookaheads and r rollbacks in the task execution. Based on the assumption of the constant probability of failure in a computation session, we can expect

$$p_l \; = \; \frac{l}{(n - l) + l + r} \; = \; \frac{l}{n + r},$$

$$p_r \; = \; \frac{r}{(n - l) + l + r} \; = \; \frac{r}{n + r}$$

which lead to

$$l = \frac{np_l}{1-p_r},$$

$$r = \frac{np_r}{1-p_r}.$$

In a successful lookahead, the task execution includes one restart time (t_r) for scheduling lookahead tasks and 2.5 checkpoint comparisons for the checkpoint validation ($2.5t_t$). For a rollback, two session times (i.e., $2(\Delta + t_k)$) are wasted: one for the original execution pair and one for the lookahead period. Two restarts also result (one for the lookaheads and one for the rollback to the previously committed checkpoint). In addition, there are three checkpoint comparisons for the checkpoint validation. Thus, the expected execution time can be given as

$$\begin{aligned} T_e &= n(\Delta + t_k) + l(t_r + 2.5t_t) + r(\Delta + t_k + t_t + t_r) + r(\Delta + t_k + t_r + 2t_t) \\ &= n(\Delta + t_k)\left(1 + \frac{2p_r}{1-p_r}\right) + nt_r\frac{p_l + 2p_r}{1-p_r} + nt_t\frac{2.5p_l + 3p_r}{1-p_r}. \end{aligned}$$

Therefore,

$$R_e = \frac{T_e}{T_0} = 1 + \frac{2p_r}{1-p_r} + \frac{p_l + 2p_r}{1-p_r}\frac{t_r}{\Delta + t_k} + \frac{2.5p_l + 3p_r}{1-p_r}\frac{t_t}{\Delta + t_k}.$$

There is one (the committed) checkpoint during the normal execution run. Two additional (uncommitted) checkpoints are present during a lookahead/validation operation. At the end of the Δ for the rollback validation, there are eight checkpoints, one committed and seven uncommitted (one for the validation process, two for the normal process pair and four for the lookahead processes). Thus,

$$\begin{aligned}
\int_0^{T_e} N_c(t)dt &= n(\Delta + t_k) + 3l(t_r + t_t) + 8l(1.5t_t) \\
&\quad + r(\Delta + t_k + t_r) + 3r(\Delta + t_k + t_r + t_t) + 8r(2t_t) \\
&= T_e + 2T_0\frac{p_l + p_r}{1 - p_r} + 2nt_r\frac{p_l + p_r}{1 - p_r} + 2nt_t\frac{6.25p_l + 8p_r}{1 - p_r}.
\end{aligned}$$

Therefore,

$$N_c = 1 + 2\frac{p_l + p_r}{(1 - p_r)R_e} + 2\frac{p_l + p_r}{(1 - p_r)R_e}\frac{t_r}{\Delta + t_k} + 2\frac{6.25p_l + 8p_r}{(1 - p_r)R_e}\frac{t_t}{\Delta + t_k}.$$

Similarly,

$$\begin{aligned}
\int_0^{T_e} N_p(t)dt &= 2(n - l)(\Delta + t_k) + 2lt_t + 5l(\Delta + t_k + t_r + 1.5t_t) \\
&\quad + 2r(\Delta + t_k + t_r + t_t) + 5r(\Delta + t_k + t_r + 2t_t) \\
&= 2T_e + 3T_0\frac{p_l + p_r}{1 - p_r} + 3nt_r\frac{p_l + p_r}{1 - p_r} + 3nt_t\frac{1.5p_l + 2p_r}{1 - p_r},
\end{aligned}$$

$$N_p = 2 + 3\frac{p_l + p_r}{(1 - p_r)R_e} + 3\frac{p_l + p_r}{(1 - p_r)R_e}\frac{t_r}{\Delta + t_k} + 3\frac{1.5p_l + 2p_r}{(1 - p_r)R_e}\frac{t_t}{\Delta + t_k}.$$

If a centralized file server serializes file accesses, both the restart (t_r) and checkpoint times (t_k) will be increased because of the serialized file accesses of checkpointing and restart. According to our previous assumption about the centralized file server, the restart time will be threefold since one rollback validation and two lookahead processes read the last committed checkpoint file at the same time. The checkpoint time is $2t_k$ for the normal pair of task replications and $5t_k$ for the lookahead period (four for lookahead and one for rollback). Thus, the relative execution time with a file server system of limited speed can be shown as

$$R_e(fs) = 1 + \frac{2p_r}{1-p_r} + \frac{p_l + 5/3p_r}{1-p_r}\frac{3t_r}{\Delta + 2t_k} + \frac{2.5p_l + 3p_r}{1-p_r}\frac{t_t}{\Delta + 2t_k} + \frac{p_l + p_r}{1-p_r}\frac{3t_k}{\Delta + 2t_k}.$$

Similarly,

$$\begin{aligned} N_c(fs) &= 1 + 2\frac{p_l + p_r}{(1-p_r)R_e(fs)} + 2\frac{p_l + p_r}{(1-p_r)R_e(fs)}\frac{3t_r}{\Delta + 2t_k} \\ &\quad + 2\frac{6.25p_l + 8p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta + 2t_k} + 2\frac{p_l + p_r}{(1-p_r)R_e(fs)}\frac{3t_k}{\Delta + 2t_k}, \end{aligned}$$

$$\begin{aligned} N_p(fs) &= 2 + 3\frac{p_l + p_r}{(1-p_r)R_e} + 3\frac{p_l + p_r}{(1-p_r)R_e}\frac{3t_r}{\Delta + 2t_k} \\ &\quad + 3\frac{1.5p_l + 2p_r}{(1-p_r)R_e}\frac{t_t}{\Delta + 2t_k} + 3\frac{p_l + p_r}{(1-p_r)R_e}\frac{3t_k}{\Delta + 2t_k}. \end{aligned}$$

The above results are summarized in Table 3.2. Given p_f, R_e is a linear function of relative overhead factors: $\frac{t_r}{\Delta + t_k}$ and $\frac{t_t}{\Delta + t_k}$. These overhead coefficients reflect the contributions their corresponding overheads have to the scheme performance (degradation). The constant term in R_e is the corresponding performance for an error-free execution, while the second term reflects the inherent rollback in DMR-F-1. The smaller this term is, the more effective is the forward recovery. Except for an extra factor $(1/R_e)$ appearing in the overhead coefficients, N_p and N_c have a pattern similar to R_e. The impact of a centralized file server for checkpoint storage is reflected through the increased coefficients for t_r and the presence of the additional overhead term for t_k.

3.3.3. DMR-F-2: Forward recovery with two rollback validations

In DMR-F-2, the forward recovery is successful if one of the uncommitted checkpoints is correct and at least one of the validation checkpoints is correct. Unlike DMR-F-1, the

Table 3.2. Analytical Evaluation Summary: DMR-F-1.

T_e	$n(\Delta + t_k)\left(1 + \frac{2p_r}{1-p_r}\right) + nt_r\frac{p_l+2p_r}{1-p_r} + nt_t\frac{2.5p_l+3p_r}{1-p_r}$
R_e	$1 + \frac{2p_r}{1-p_r} + \frac{p_l+2p_r}{1-p_r}\frac{t_r}{\Delta+t_k} + \frac{2.5p_l+3p_r}{1-p_r}\frac{t_t}{\Delta+t_k}$
N_c	$1 + 2\frac{p_l+p_r}{(1-p_r)R_e} + 2\frac{p_l+p_r}{(1-p_r)R_e}\frac{t_r}{\Delta+t_k} + 2\frac{6.25p_l+8p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	8
N_p	$2 + 3\frac{p_l+p_r}{(1-p_r)R_e} + 3\frac{p_l+p_r}{(1-p_r)R_e}\frac{t_r}{\Delta+t_k} + 3\frac{1.5p_l+2p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_p)$	5
$R_e(fs)$	$1 + \frac{2p_r}{1-p_r} + \frac{p_l+5/3p_r}{1-p_r}\frac{3t_r}{\Delta+2t_k} + \frac{2.5p_l+3p_r}{1-p_r}\frac{t_t}{\Delta+2t_k} + \frac{p_l+p_r}{1-p_r}\frac{3t_k}{\Delta+2t_k}$
$N_c(fs)$	$1 + 2\frac{p_l+p_r}{(1-p_r)R_e(fs)} + 2\frac{p_l+p_r}{(1-p_r)R_e(fs)}\frac{3t_r}{\Delta+2t_k}$ $+2\frac{6.25p_l+8p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta+2t_k} + 2\frac{p_l+p_r}{(1-p_r)R_e(fs)}\frac{3t_k}{\Delta+2t_k}$
$N_p(fs)$	$2 + 3\frac{p_l+p_r}{(1-p_r)R_e} + 3\frac{p_l+p_r}{(1-p_r)R_e}\frac{3t_r}{\Delta+2t_k}$ $+3\frac{1.5p_l+2p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+2t_k} + 3\frac{p_l+p_r}{(1-p_r)R_e}\frac{3t_k}{\Delta+2t_k}$

rollback distance of DMR-F-2 varies if the rollback validation fails. It is one Δ if both of the original tasks fail and the validation pair succeeds. In this case, the checkpoints by the validation pair are correct and can be committed. The rollback can start from them instead of from the last committed checkpoint. Otherwise, the rollback distance is 2Δs. Let l, s, r be the average number of successful lookaheads, one-Δ rollbacks and 2-Δ rollbacks, respectively. Their corresponding probabilities are

$$
\begin{aligned}
p_l &= 2(1-p_f)p_f(1-p_f^2), \\
p_s &= p_f^2(1-p_f)^2, \\
p_r &= 2(1-p_f)p_f p_f^2 + p_f^2\left(2(1-p_f)p_f + p_f^2\right).
\end{aligned}
$$

Table 3.3. Analytical Evaluation Summary: DMR-F-2.

T_e	$n(\Delta + t_k)\left(1 + \frac{p_s+2p_r}{1-p_r}\right) + nt_r\frac{p_l+2p_s+2p_r}{1-p_r} + nt_t\frac{3.5p_l+5p_s+5p_r}{1-p_r}$
R_e	$1 + \frac{p_s+2p_r}{1-p_r} + \frac{p_l+2p_s+2p_r}{1-p_r}\frac{t_r}{\Delta+t_k} + \frac{3.5p_l+5p_s+5p_r}{1-p_r}\frac{t_t}{\Delta+t_k}$
N_c	$1 + 2\frac{p_l+p_s+p_r}{(1-p_r)R_e} + 2\frac{p_l+p_s+p_r}{(1-p_r)R_e}\frac{t_r}{\Delta+t_k} + 2\frac{11p_l+21.5p_s+21.5p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	9
N_p	$2 + 4\frac{p_l+p_s+p_r}{(1-p_r)R_e} + 4\frac{p_l+p_s+p_r}{(1-p_r)R_e}\frac{t_r}{\Delta+t_k} + 4\frac{2.5p_l+5.5p_s+5.5p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_p)$	6
$R_e(fs)$	$1 + \frac{p_s+2p_r}{1-p_r} + \frac{p_l+5/3p_s+5/3p_r}{1-p_r}\frac{3t_r}{\Delta+2t_k}$ $+\frac{3.5p_l+5p_s+5p_r}{1-p_r}\frac{t_t}{\Delta+2t_k} + \frac{p_l+p_s+p_r}{1-p_r}\frac{3t_k}{\Delta+2t_k}$
$N_c(fs)$	$1 + 2\frac{p_l+p_s+p_r}{(1-p_r)R_e(fs)} + 2\frac{p_l+p_s+p_r}{(1-p_r)R_e(fs)}\frac{3t_r}{\Delta+2t_k}$ $+2\frac{11p_l+21.5p_s+21.5p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta+2t_k} + 2\frac{p_l+p_s+p_r}{(1-p_r)R_e(fs)}\frac{3t_k}{\Delta+2t_k}$
$N_p(fs)$	$2 + 4\frac{p_l+p_s+p_r}{(1-p_r)R_e} + 4\frac{p_l+p_s+p_r}{(1-p_r)R_e}\frac{3t_r}{\Delta+2t_k}$ $+4\frac{2.5p_l+5.5p_s+5.5p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+2t_k} + 4\frac{p_l+p_s+p_r}{(1-p_r)R_e}\frac{3t_k}{\Delta+2t_k}$

Since the analysis of DMR-F-2 is very similar to that for DMR-F-1 except for DMR-F-2 using an extra process for rollback validation, we summarize the resulting formulas for DMR-F-2 in Table 3.3. The detailed analysis can be found in Appendix A. The results are similar to those of DMR-F-1.

3.3.4. TMR-F: Triple module redundancy

In TMR-F, the trio of replicated tasks continues when there is no erroneous checkpoint at the end of Δ. If there is a match in checkpoints, TMR-F performs a forward recovery via masking off the erroneous checkpoint. If all checkpoints are different, a rollback is scheduled. The rollback probability is given as

Table 3.4. Analytical Evaluation Summary: TMR-F.

T_e	$n(\Delta + t_k)\left(1 + \frac{p_r}{1-p_r}\right) + nt_r\frac{p_r}{1-p_r} + nt_t\frac{3p_r}{1-p_r}$
R_e	$1 + \frac{p_r}{1-p_r} + \frac{p_r}{1-p_r}\frac{3t_r}{\Delta+t_k} + \frac{3p_r}{1-p_r}\frac{t_t}{\Delta+t_k}$
N_c	$1 + \frac{9p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	4
N_p	3
$max(N_p)$	3
$R_e(fs)$	$1 + \frac{p_r}{1-p_r} + \frac{p_r}{1-p_r}\frac{3t_r}{\Delta+3t_k} + \frac{3p_r}{1-p_r}\frac{t_t}{\Delta+3t_k}$
$N_c(fs)$	$1 + \frac{9p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta+3t_k}$
$N_p(fs)$	3

$$p_r = 3p_f^2 - 2p_f^3.$$

The results of our analysis for TMR-F are summarized in Table 3.4. The detailed derivations are given in Appendix A. For TMR-F, R_e follows a pattern similar to DMR-F-1 and DMR-F-2. However, N_c has no inherent rollback term, since TMR-F always has one committed checkpoint during either normal computation sessions or rollback sessions, and three additional checkpoints at the end of each session for the duration of the checkpoint test time ($3t_t$).

3.3.5. DMR-B-1: Backward recovery with one rollback process

In this scheme, there is no forward recovery. If there is an error in either of the two original replicated processes, the computation session rolls back repeatedly until there is a

Table 3.5. Analytical Evaluation Summary: DMR-B-1.

T_e	$n(\Delta + t_k)\left(1 + \frac{p_1+2p_2}{1-p_f}\right) + nt_r\frac{p_1+2p_2}{1-p_f}$ $+nt_t\frac{p_f(10-15p_f+18p_f^2-7p_f^3)}{2(1-p_f)^2}\frac{t_t}{\Delta+t_k}$
R_e	$1 + \frac{p_1+2p_2}{1-p_f} + \frac{p_1+2p_2}{1-p_f}\frac{t_r}{\Delta+t_k} + \frac{p_f(10-15p_f+18p_f^2-7p_f^3)}{2(1-p_f)^2}\frac{t_t}{\Delta+t_k}$
N_c	$\left[1 + \frac{p_1+3p_2}{(1-p_f)^2} + \frac{2p_1+4p_2}{1-p_f}\right]\frac{1}{R_e} + \left[\frac{p_1+3p_2}{(1-p_f)^2} + \frac{2p_1+4p_2}{1-p_f}\right]\frac{t_r}{(\Delta+t_k)R_e}$ $+ \left[\frac{2p_1+8p_2}{(1-p_f)^3} + \frac{p_1+6p_2}{(1-p_f)^2} + \frac{3/2p_1+3p_2}{1-p_f} + \frac{9p_1+3p_2}{2}\right]\frac{t_t}{(\Delta+t_k)R_e}$
$max(N_c)$	∞
N_p	$1 + \frac{1}{R_e} + \frac{p_1+p_2}{R_e}\frac{t_k}{\Delta+t_k}$
$max(N_p)$	2
$R_e(fs)$	$\frac{\Delta+t_k}{\Delta+2t_k}R_e + \frac{t_k}{\Delta+2t_k}$
$N_c(fs)$	$\frac{\Delta+t_k}{\Delta+2t_k)}N_c + \frac{t_k}{(\Delta+2t_k)R_e(fs)}$
$N_p(fs)$	$\frac{\Delta+t_k}{\Delta+2t_k}N_p + \frac{2t_k}{(\Delta+2t_k)R_e(fs)}$

match in the uncommitted checkpoints produced by both the original process pair and the rollback processes [27,28]. Two situations can cause a rollback: (1) there is one error-free checkpoint produced by the original process pair and the rollback iterations need only to generate another correct checkpoint; (2) both checkpoints are erroneous for the original process pair run, and the rollback iterations need to produce two valid checkpoints. The corresponding probabilities are $p_1 = 2p_f(1-p_f)$ and $p_2 = p_f^2$, respectively. The analysis of DMR-B-1 is given in detail in Appendix A. The summary is presented in Table 3.5.

3.3.6. DMR-B-2: Backward recovery with two rollback processes

Like DMR-B-1, this scheme employs the recursive rollback to find a pair of matched checkpoints. However, they differ in that DMR-B-2 uses two rollback tasks whereas DMR-B-1 uses one. For the detailed analysis, refer to Appendix A. The summary is given in Table 3.6.

3.3.7. Discussion

According to Tables 3.2–3.6, R_e, N_p or N_c can be generally expressed in terms of the relative overhead factors as shown below:

$$m = c + \alpha + \beta \frac{t_r}{\Delta + t'_k} + \gamma \frac{t_t}{\Delta + t'_k} + \delta \frac{t'_k}{\Delta + t'_k},$$

where c is a constant, $m \in \{R_e, N_c, N_p\}$ and t'_k is either $2t_k$ or $3t_k$. The constant c reflects the error-free performance, while α is the performance degradation due to rollbacks in the schemes we considered. The smaller α is for a forward recovery scheme, the more effective the scheme is in terms of reducing the execution time degradation. In this thesis, α is called the coefficient of the overhead due to rollback. This rollback overhead can not be eliminated and depends only on the failure probability, p_f. The expression $c + \alpha$ represents the inherent performance of a particular scheme, since all eliminatable overheads are removed (i.e., the zero overhead situation). The factors of β, γ, and δ are the overhead coefficients for process restart, checkpoint comparison test and checkpointing.

For R_e, the overhead coefficients, α, β, γ and δ are related only to p_f. For N_p and N_c, the coefficients also include a factor of $\frac{1}{R_e}$. Normally, the relative overheads are very small, and we can approximate R_e with the zero-overhead R_e. This approximation, in fact, gives

Table 3.6. Analytical Evaluation Summary: DMR-B-2.

T_e	$n(\Delta + t_k)\left[1 + \frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}\right] + nt_r\frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}$ $+nt_t\left[4p_f^2\frac{4p_f+3}{(1+p_f)^3(1-p_f)^2} + p_f\frac{6+2p_f-p_f^2}{(1+p_f)^2}\right]$
R_e	$1 + \frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2} + + \frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}\frac{t_r}{\Delta+t_k}$ $+\left[4p_f^2\frac{4p_f+3}{(1+p_f)^3(1-p_f)^2} + p_f\frac{6+2p_f-p_f^2}{(1+p_f)^2}\right]\frac{t_t}{\Delta+t_k}$
N_c	$\left[1 + \frac{p_f(6+5p_f+3p_f^2-3p_f^3+p_f^4)}{(1+p_f^3)(1-p_f)^2}\right]\frac{1}{R_e} + \frac{p_f(6+5p_f+3p_f^2-3p_f^3+p_f^4)}{(1+p_f^3)(1-p_f)^2R_e}\frac{t_r}{\Delta+t_k}$ $+\frac{p_f(137+10p_f-16p_f^2-22p_f^3+19p_f^4+4p_f^5-4p_f^6}{(1-p_f)^3(1+p_f)^4}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	∞
N_p	2
$max(N_p)$	2

$R_e(fs)$	$1 + \frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2} + \frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}\frac{2t_r}{\Delta+2t_k}$ $+\left[4p_f^2\frac{4p_f+3}{(1+p_f)^3(1-p_f)^2} + p_f\frac{6+2p_f-p_f^2}{(1+p_f)^2}\right]\frac{t_t}{\Delta+2t_k}$
$N_c(fs)$	$\left[1 + \frac{p_f(6+5p_f+3p_f^2-3p_f^3+p_f^4)}{(1+p_f^3)(1-p_f)^2}\right]\frac{1}{R_e} + \frac{p_f(6+5p_f+3p_f^2-3p_f^3+p_f^4)}{(1+p_f^3)(1-p_f)^2R_e}\frac{t_r}{\Delta+t_k}$ $+\frac{p_f(137+10p_f-16p_f^2-22p_f^3+19p_f^4+4p_f^5-4p_f^6}{(1-p_f)^3(1+p_f)^4}\frac{t_t}{\Delta+t_k}$
$N_p(fs)$	2

the upper bound for α, β, γ and δ in N_p and N_c, since the presence of t_r, t_t and t_k increases R_e. Therefore, N_p and N_c are approximately a linear function of overhead factors. The overhead coefficients represent the contribution of their corresponding overhead factors to the performance degradation. The larger the coefficient for an overhead factor, the more important this overhead factor is with respect to performance degradation.

For the noncentralized file server situation, δ is zero. The checkpoint time, t_k, does not appear as an overhead factor because an error-free execution time that includes checkpoint time, $n(\Delta + t_k)$, is used as the base for our performance measures. In fact, the overhead coefficient for the checkpoint time is $c + \alpha$ if the checkpointless error-free execution time is used as the base for our performance measures. For example, R_e can be redefined as the ratio of the expected execution time over the checkpointless error-free execution time, $\frac{T_e}{n\Delta}$, instead of $\frac{T_e}{\Delta+t_k}$. That is,

$$R'_e = \frac{T_e}{n\Delta} = \frac{T_e}{\Delta + t_k}\frac{\Delta + t_k}{\Delta} = R_e\frac{\Delta + t_k}{\Delta} = c + \alpha + (c + \alpha)\frac{t_k}{\Delta} + \beta\frac{t_r}{\Delta} + \gamma\frac{t_t}{\Delta}.$$

Checkpointing overhead is inherent in any checkpoint-based scheme since checkpoint time is always included in the execution whether a fault occurs or not. For this reason, we did not use the checkpointless error-free time as the base for our performance measure. To minimize the impact of checkpoint overhead, the checkpoint interval or frequency should be determined optimally.

The performance degradation due to a centralized file server is reflected in two ways. The first is the increased overhead caused by the file access serialization. For example, an approximate factor of 3 appears for the restart overhead term, $\frac{t_r}{\Delta+t_k}$ in $R_e(fs)$. The second

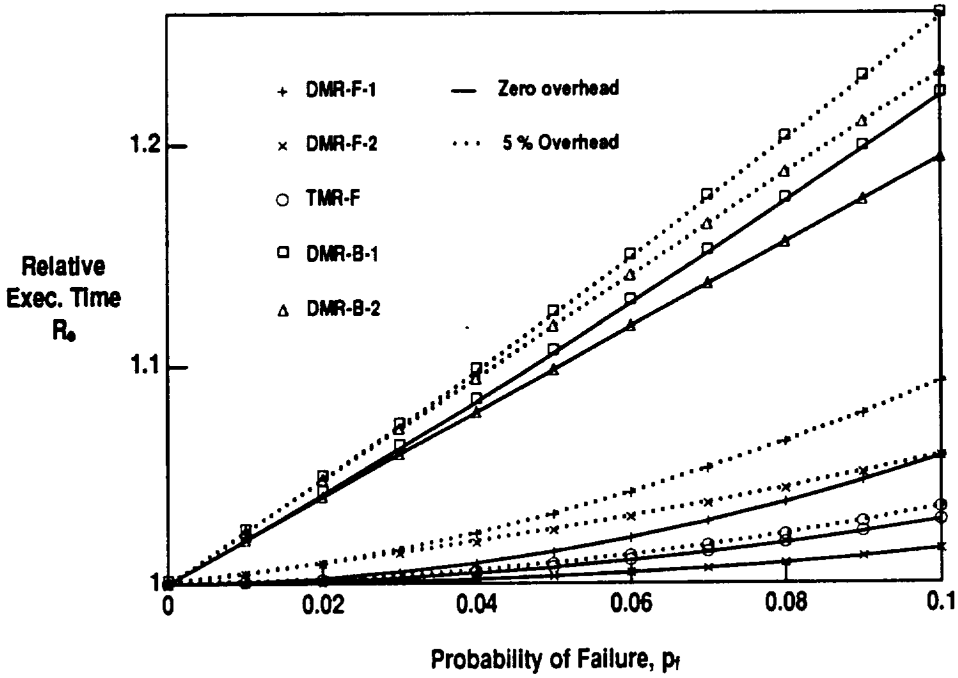

Figure 3.1. Comparison: Relative Execution Time.

is the nonzero overhead coefficient for checkpoint time (δ) because of the extra checkpointing activities by the lookahead and rollback validation processes during recovery.

3.3.8. Comparison

In order to compare the five schemes we described above, R_e, N_p, and N_c are plotted in Figures 3.1, 3.2 and 3.3. The solid curves depict the zero-overhead case (i.e., the inherent performance, $c + \alpha$), whereas the dotted curves depict the case with 5% overheads (e.g., t_k, t_r and t_t are 5% of $\Delta + t_k$, respectively). The range of failure probability considered is limited within [0, 0.1], since typical environments are unlikely to have high failure rates.

In Figure 3.1, the expected execution time for DMR-F-1 and DMR-F-2 is comparable to that for TMR-F. In fact, their execution times are nearly the same as the error-free execution time. The execution times for the rollback schemes (DMR-B-1 and DMR-B-2)

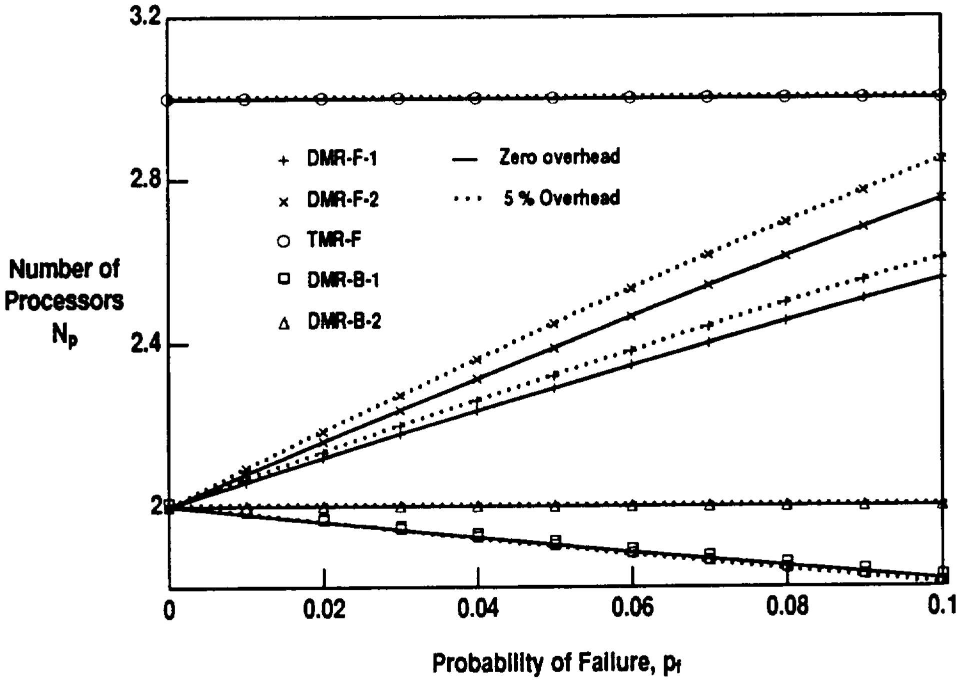

Figure 3.2. Comparison: Number of Processors.

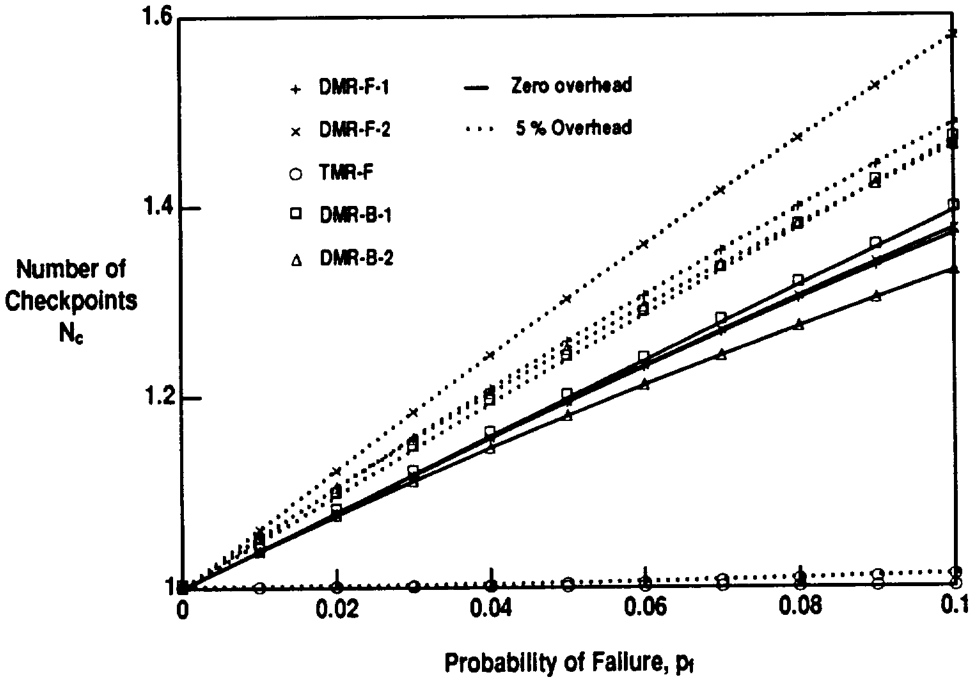

Figure 3.3. Comparison: Number of Checkpoints.

can be as high as 20% more than the error-free execution time. The increase in R_e with p_f shows that rollback is still possible in TMR-F, DMR-F-1 and DMR-F-2, even though these schemes can perform forward recovery. The relative execution time, R_e, for DMR-F-2, is larger than that for DMR-F-2 because there are more rollback validation failures in DMR-F-1.

The average number of processors used for DMR-F-1 and DMR-F-2 is less than that of TMR (Figure 3.2). Using more than three processors dynamically for the infrequent error situation enables DMR-F-1 and DMR-F-2 to reduce the overall processor redundancy. As expected, the rollback schemes, DMR-B-1 and DMR-B-2, use on average fewer processors than the others. For DMR-B-1, N_p decreases with p_f because only one processor is used during recovery.

The number of checkpoints increases with p_f for all schemes except TMR-F. For TMR-F, N_c is close to one; N_c for DMR-F-1 and DMR-F-2 is slightly higher than that for DMR-B-1 and DMR-B-2. It seems contradictory to the fact that more checkpoints would be accumulated during recovery for DMR-B-1 and DMR-B-2. However, DMR-B-1 and DMR-B-2 do have a smaller N_c than DMR-F-1 and DMR-F-2 because they have a longer execution time than DMR-F-1 and DMR-F-2 due to rollbacks. The difference in N_c may be insignificant, since most modern systems usually have a large secondary storage for the checkpoint files.

As expected, the presence of overhead increases R_e. Both DMR-F-1 and DMR-F-2 still have an execution time close to the error-free execution time (within 5% for DMR-F-2 and 10% for DMR-F-1). For DMR-F-1 and DMR-F-2, N_p is increased less than 1% because the extra processors are used only during recovery. For TMR-F and DMR-B-2, N_p is constant, since both schemes always use three and two processors, respectively, during both normal execution and recovery.

3.3.9. Overhead impact

As discussed in Section 3.3.7, the overhead coefficients reflect the importance of the corresponding overheads with respect to performance degradation. The impact of checkpoint overhead is determined by $c + \alpha$ and depicted in Figures 3.1, 3.2 and 3.3 as the zero-overhead curves. The impact of checkpoint overhead on R_e for DMR-F-1, DMR-F-2 and TMR-F is smaller than that for DMR-B-1 and DMR-B-2. This is because the rollback reduction in DMR-F-1, DMR-F-2 and TMR-F leads to fewer checkpointing sessions in computation (Figure 3.1). For DMR-F-1 and DMR-F-2, N_p, is more sensitive to the checkpoint overhead than that for TMR-F, DMR-B-1 and DMR-B-2 as indicated by a positive slope in Figure 3.2. The static redundancy employed in TMR-F and DMR-B-2 is reflected by the flat slopes in Figure 3.2. Except for TMR-F, the sensitivity of N_c to the checkpoint overhead is reflected by the relatively steep slopes in Figure 3.3.

Figures 3.4, 3.5 and 3.6 compare the overhead coefficients of restart time and checkpoint comparison time. The solid curves represent the impact of t_r; the dotted ones depict the impact of t_t. For N_p and N_c, the overhead coefficients such as α, β and γ may not be independent of t_k, t_r and t_t due to the presence of R_e in these coefficients. In Figures 3.5 and 3.6, we used the ideal zero-overhead result for R_e conservatively since R_e with overhead is bigger than without overhead.

The impact of the comparison time t_t is more than twice of that of the restart time t_r. This suggests that any reduction in comparison time will result in a bigger gain in

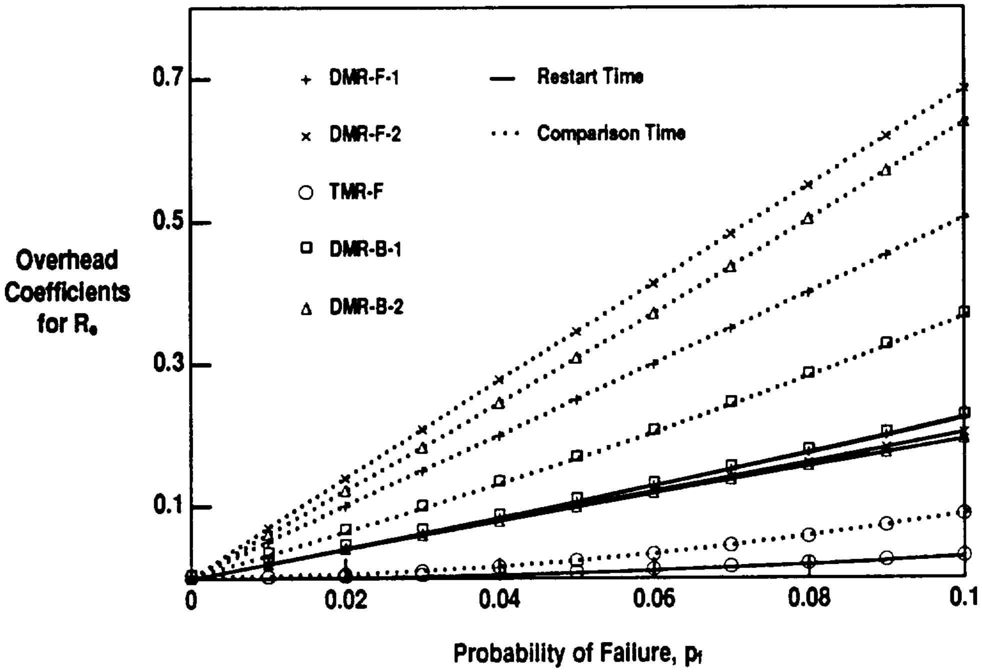

Figure 3.4. Overhead Impact on Execution Time.

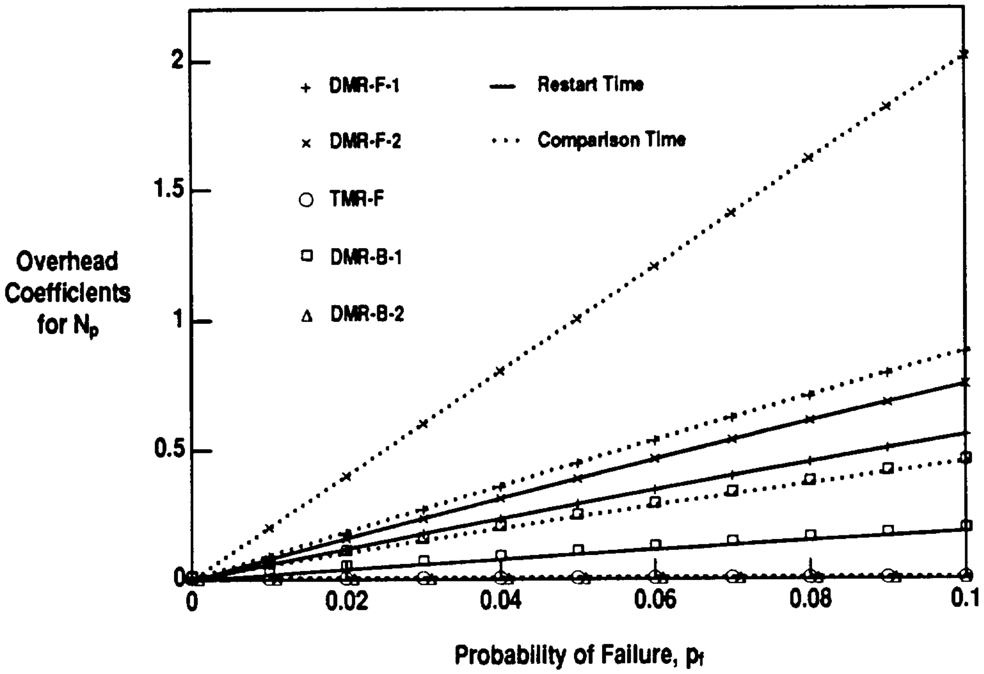

Figure 3.5. Overhead Impact on Number of Processors.

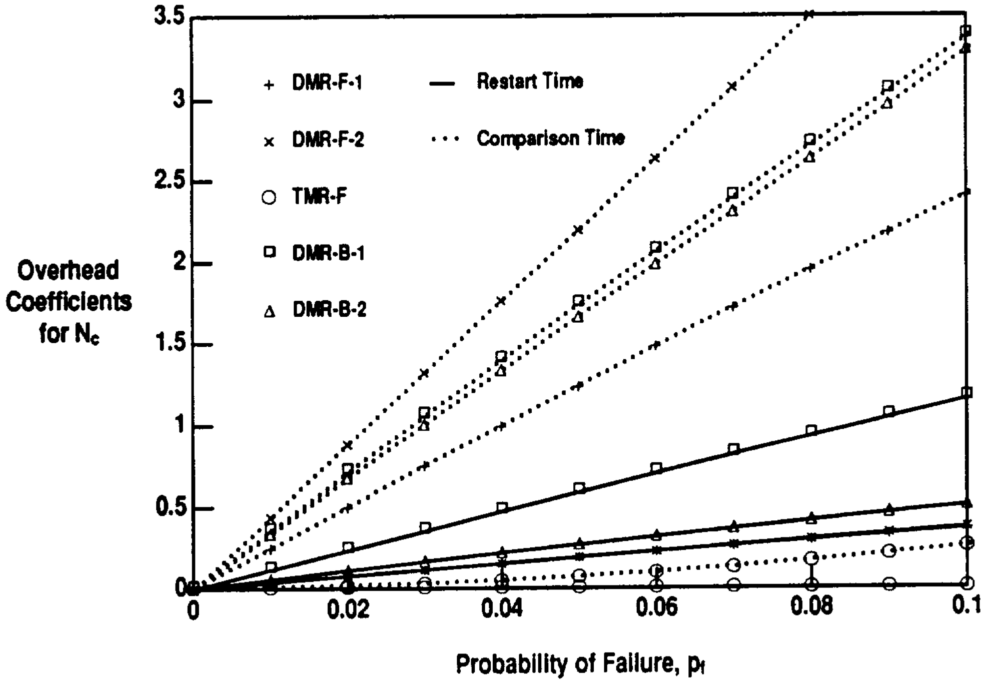

Figure 3.6. Overhead Impact on Number of Checkpoints.

performance improvement than will an equal reduction in restart time. In Figure 3.4, t_r and t_t affect R_e for DMR-F-2 and DMR-B-2 more than R_e for other schemes. The TMR-F scheme is insensitive to both t_r and t_t. Figure 3.5 shows that N_p is affected by t_r and t_t more for DMR-F-1 and DMR-F-2 than for other schemes because they employ extra processors during recovery. The number of checkpoints for all schemes except TMR-F is sensitive to the overheads (Figure 3.6).

3.3.10. File server impact

The impact of a centralized file server is depicted in Figure 3.7 for a case with 5 % relative overhead. The solid curves are for the centralized server case, while the dotted ones are for the noncentralized server case. The impact of a centralized file server for TMR-F, DMR-B-1, and DMR-B-2 is not as significant as that for DMR-F-1 and DMR-F-2, since there are additional checkpoint operations and restarts by the lookahead and

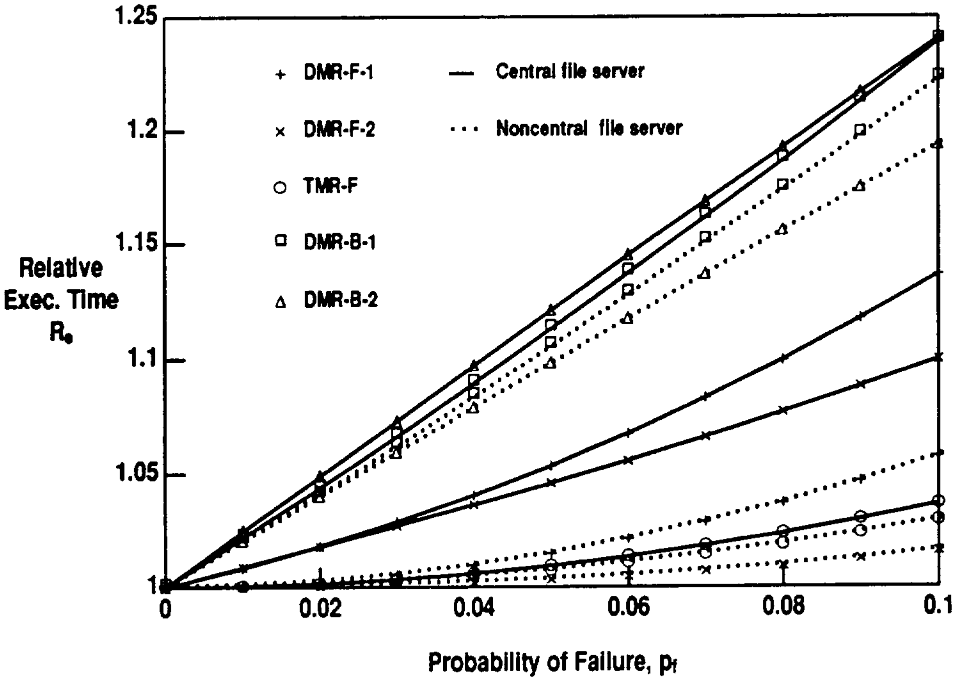

Figure 3.7. Impact of a Central File Server on Execution Time.

rollback validation processes during recovery for DMR-F-1 and DMR-F-2. Restart time and checkpoint time increase as a result of the file access serialization by the centralized server. Meanwhile, an extra overhead term for t_k appears in R_e, N_p and N_c (Tables 3.2 and 3.3).

3.3.11. Optimal checkpoint placement

The formulas for T_e in Tables 3.2 and 3.3 can be used to minimize the impact of checkpoint time on execution time by selecting the proper checkpoint interval or frequency. If the checkpoint interval is too long, the execution loss due to reprocessing increases the execution time, while the checkpointing overhead increases the execution time with frequent checkpointing. Figure 3.8 shows the expected execution time under different failure rates and overhead costs for DMR-F-1. The optimal checkpoint frequency can be obtained

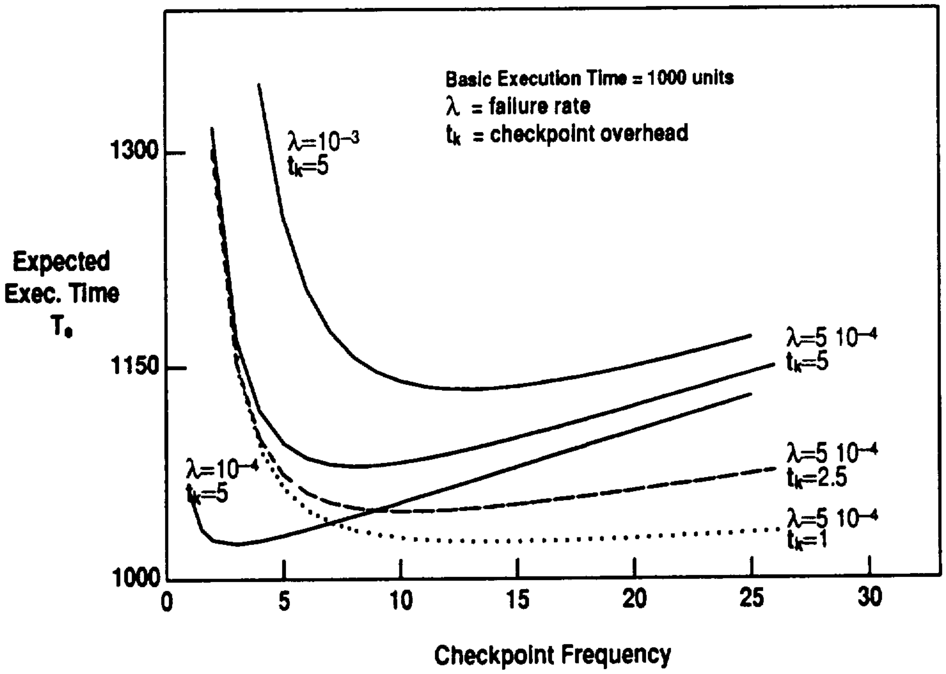

Figure 3.8. Optimal Checkpoint Placement.

by either numerical or graphical means, given a failure rate, task computation time, and overhead costs such as checkpoint time, restart time, and comparison time.

Note in Figure 3.8 that for a low checkpointing overhead, the execution time curve near the bottom is rather flat. This suggests that an accurate checkpoint interval is not necessary since a few additional checkpoints still give a near optimal solution. For small failure rates, the checkpoint interval is usually large or checkpoint frequency is small. This observation agrees with the previous studies on optimal checkpoint placement for other recovery schemes [1–4].

3.4. Self-Testable Scheme

If a checkpoint is self-testable, errors in the checkpoint can be detected by using this checkpoint alone. During recovery, the lookahead scheduling can be reduced to only scheduling a replicated process from the correct checkpoint identified by the self-testable

Table 3.7. Analytical Evaluation Summary: Self-Testable Scheme.

T_e	$n(\Delta + t_k)\left(1 + \frac{p_r}{1-p_r}\right) + nt_r\frac{p_l+p_r}{1-p_r} + nt_t\frac{p_l+2p_r}{1-p_r}$
R_e	$1 + \frac{p_r}{1-p_r} + \frac{p_l+p_r}{1-p_r}\frac{t_r}{\Delta+t_k} + \frac{p_l+2p_r}{1-p_r}\frac{t_t}{\Delta+t_k}$
N_c	$1 + 2\frac{p_l}{(1-p_r)R_e}\frac{t_r}{\Delta+t_k} + 2\frac{2p_r}{(1-p_r)R_e}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	3
N_p	2
$max(N_p)$	2

test. This leads to the elimination of both the rollback validation process and the lookahead processes from the erroneous checkpoint. For the purpose of comparison, we consider a self-test scheme with a base size of two in the following.

This scheme uses two replicated processes for the computation sessions. At the end of a checkpoint session, the test portions from the two newly produced checkpoints are tested. If one checkpoint is tested as erroneous, a new replicated process is restarted from the error-free checkpoint and the original erroneous process is terminated. The correct checkpoint is committed. If both checkpoint are tested as erroneous, a rollback is performed. Therefore, the probabilities of a successful lookahead and of rollback are, respectively,

$$\begin{aligned} p_l &= 2p_f(1-p_f) \\ p_r &= p_f^2. \end{aligned}$$

In Table 3.7, R_e, N_p and N_c are summarized. (For details see Appendix A) For the self-testable scheme, R_e is similar to that for DMR-F-1 and DMR-F-2 (Figure 3.9). This scheme also compares favorably to TMR-F (Figure 3.9): it has a shorter execution time than TMR-F. With 5% overhead, the self-testable scheme is still comparable to TMR-F,

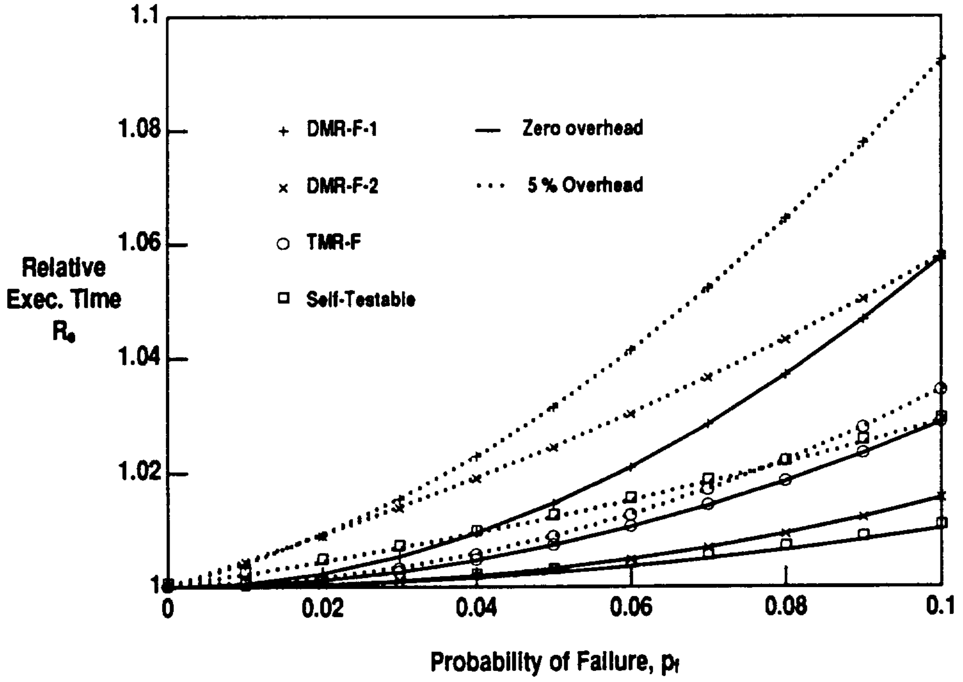

Figure 3.9. Self-Testable Scheme: R_e Comparison.

even though it is more sensitive to the overhead than TMR-F. The self-testable scheme uses two static processors during both normal computation and recovery. However, N_c does not have the term for inherent rollback. This is because the number of checkpoints is always one for either normal execution or recovery except for the short period during the checkpoint testing when it is three. This explains why N_c for the self-testable scheme is insensitive to the overhead, similar to that for TMR-F (Figure 3.10).

3.5. Graceful Performance Degradation Scheme

Compared to TMR-F, the extra processors required during recovery by our strategy may not always be available. A graceful degradation in forward recovery can be achieved by using the limited lookahead scheduling. For example, we can schedule one lookahead process instead of two. Although this compromises the performance, it does not render our strategy a useless one.

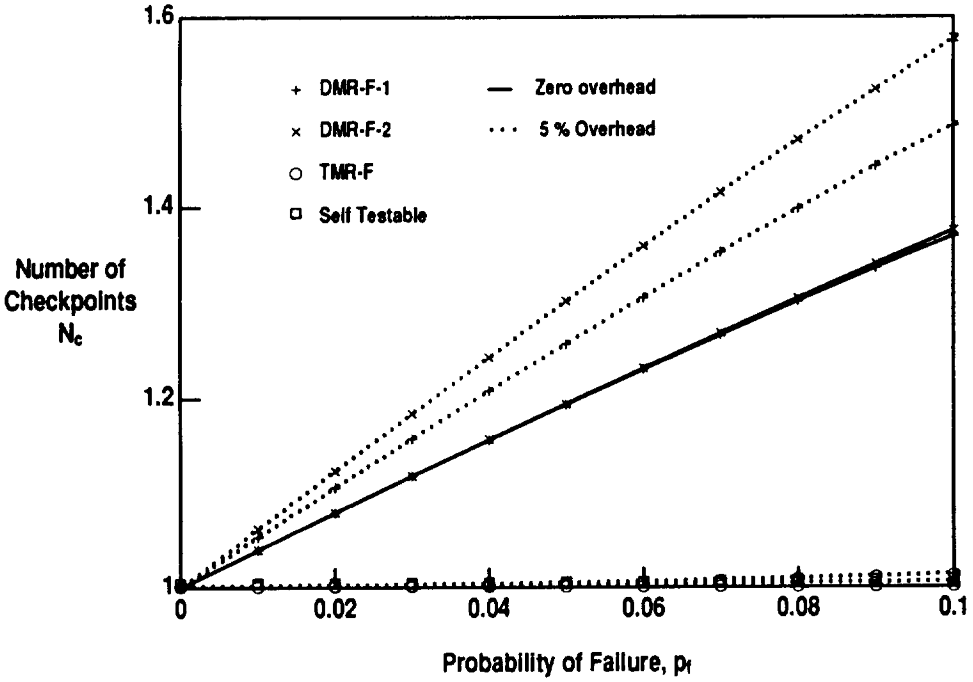

Figure 3.10. Self-Testable Scheme: N_c Comparison.

In the following, we demonstrate the graceful degradation caused by the limited lookahead scheduling with DMR-F-1. We assume that the lookahead scheduling is to select randomly one lookahead process for one of the two uncommitted checkpoints. The analysis for this degradation scheme is similar to that of DMR-F-1. In fact, they are the same except that half of the successful lookahead executions in DMR-F-1 become unsuccessful and result in rollbacks due to the misscheduled lookahead execution in this degradation scheme. Therefore, the probabilities of a successful lookahead and a rollback are, respectively,

$$p_{l'} = \frac{1}{2}p_l,$$

$$p_{r'} = p_r + \frac{1}{2}p_l,$$

where p_l and p_r are the probabilities of successful lookahead and rollback for DMR-F-1, respectively (Section 3.3.2). This reduction in successful lookahead and increase in rollback

Table 3.8. Analytical Evaluation Summary: Graceful Degradation Scheme.

T_e	$n(\Delta + t_k)\left(1 + \frac{2p_{r'}}{1-p_{r'}}\right) + nt_r\frac{p_{l'}+2p_{r'}}{1-p_{r'}} + nt_t\frac{2.5p_{l'}+3p_{r'}}{1-p_{r'}}$
R_e	$1 + \frac{2p_{r'}}{1-p_{r'}} + \frac{p_{l'}+2p_{r'}}{1-p_{r'}}\frac{t_r}{\Delta+t_k} + \frac{2.5p_{l'}+3p_{r'}}{1-p_{r'}}\frac{t_t}{\Delta+t_k}$
N_c	$1 + 2\frac{p_{l'}+p_{r'}}{(1-p_{r'})R_e} + 2\frac{p_{l'}+p_{r'}}{(1-p_{r'})R_e}\frac{t_r}{\Delta+t_k} + 2\frac{4.75p_{l'}+6p_{r'}}{(1-p_{r'})R_e}\frac{t_t}{\Delta+t_k}$
$max(N_c)$	6
N_p	$2 + \frac{p_{l'}+p_{r'}}{(1-p_{r'})R_e} + \frac{p_{l'}+p_{r'}}{(1-p_{r'})R_e}\frac{t_r}{\Delta+t_k} + \frac{1.5p_{l'}+2p_{r'}}{(1-p_{r'})R_e}\frac{t_t}{\Delta+t_k}$
$max(N_p)$	3

cause a reduction in performance, compared to DMR-F-1. Table 3.8 summarizes the results of our analysis.

Figure 3.11 compares this degraded scheme with DMR-F-1, DMR-B-1 and a nonrecursive version of DMR-B-1. The DMR-B-1 scheme is a recursive rollback scheme that gives the best possible performance among all of its nonrecursive derivations. Both DMR-F-1 and its degraded scheme are nonrecursive. The nonrecursive DMR-B-1 is simply a DMR-F-1 without lookahead scheduling. Clearly, this degraded DMR-F-1 gives a longer execution time than DMR-B-1, although its degraded performance is still between those of DMR-F-1 and the nonrecursive DMR-B-1. This suggests that DMR-F-1 should be switched to either DMR-B-1 or the degraded recursive version of DMR-F-1, if the processor available is limited at the time of recovery. The improvement in N_p and N_c can be seen in Figures 3.12 and 3.13. In fact, the performance of the degraded scheme lies between those of DMR-F-1 and DMR-B-2.

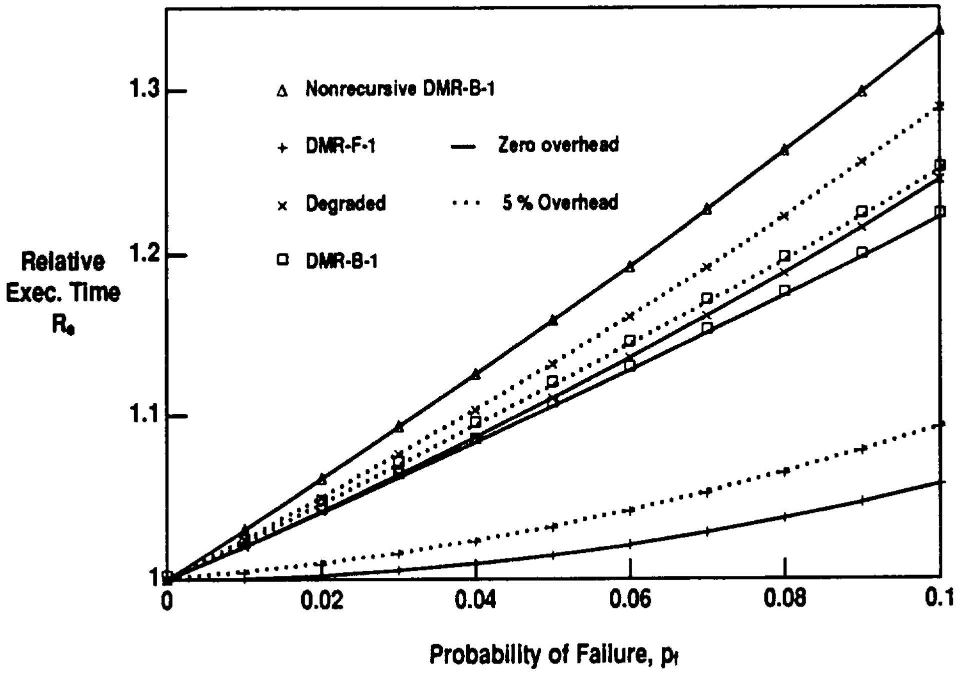

Figure 3.11. Degraded Scheme: R_e Comparison.

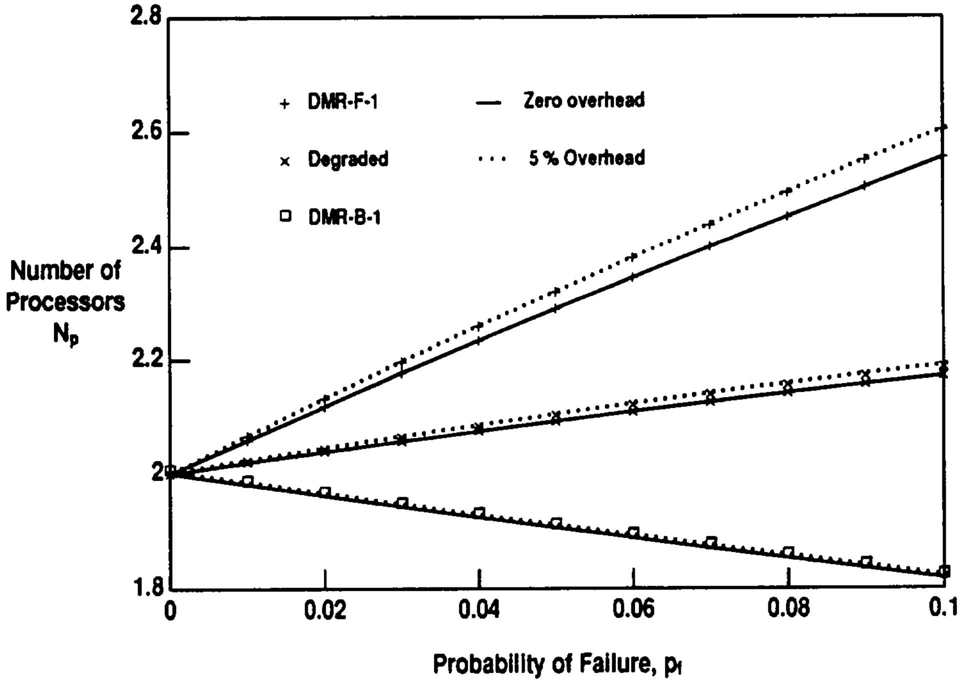

Figure 3.12. Degraded Scheme: N_p Comparison.

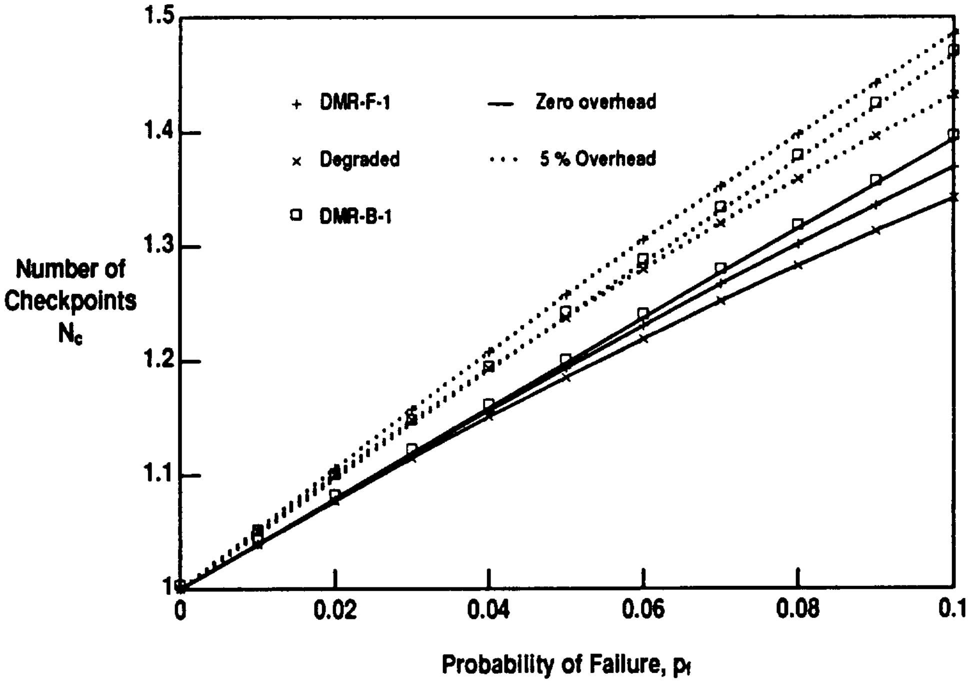

Figure 3.13. Degraded Scheme: N_c Comparison.

3.6. Summary

In this chapter, we have shown the following analytical results:

- Our forward recovery schemes (DMR-F-1 and DMR-F-2) can achieve a nearly error-free execution time with an average redundancy less than three.
- Checkpoint time is the inherent overhead in the five schemes we considered as in any checkpoint-based schemes. It is proportional to the sum of the error-free result and the inherent rollback term (α). It can be minimized by placing checkpoints optimally.
- Checkpoint test time is more important than restart time ($\gamma > \beta$). Any improvement due to a reduced test time gains more in performance than that due to a reduced restart time.
- The presence of a centralized file server increases restart time and checkpoint time. It also results in an additional degradation in performance.

- If a self-testable test is available, our forward recovery scheme can achieve a shorter execution time and a lower static processor redundancy than TMR-F.

- With limited processor resources, our schemes can obtain a reduced forward recovery gracefully. Furthermore, the performance of our recovery schemes can be naturally degraded to those of rollback ones.

CHAPTER 4.

EXPERIMENTAL EVALUATION

4.1. Introduction

In this chapter, we discuss our DMR-F-1 implementation for a distributed system. The objective is to investigate the feasibility of DMR-F-1 for distributed systems and to measure the performance overheads in an actual implementation. We selected DMR-F-1 for three reasons: (1) DMR-F-1 is representative for our recovery strategy; (2) DMR-F-1 is very sensitive to overhead (Section 3.3.9); and (3) DMR-F-1 can be easily extended into other schemes such as DMR-F-2. Our distributed implementation utilizes the ease of availability of workstations, and it can be generalized to the message passing-based parallel systems such as hypercubes and connection machines. The memory shared parallel systems may have a common mode of failure with the shared memory. If the processes that share the memory space are treated as a computation entity, our implementation may be applied in this case as well.

In our implementation, a checkpoint is the running image of a process. The test and state portions of a checkpoint are not separate entities. The test mechanism is simply a comparison of two checkpoint files. This selection of checkpoint structure and test gives potentially maximal checkpoint and test times. It also requires no knowledge about error and may handle possibly a wide range of failures.

4.2. Host Environment

Our implementation environment consists of a Sun 3/280 server and a pool of 12 Sun 3/50 diskless workstations. The server provides a Sun NFS transparent access to remote file systems under SunOS 4.0. A voter task for the checkpoint comparison and recovery initiation is also run on this server. All checkpoints are kept by the server. The Sun 3/50 workstations are used as the processing units. This setting makes it possible to evaluate the impact of the centralized file server on DMR-F-1. This is an entirely user level implementation with no kernel modifications required.

4.3. Basic Problems

Two problems have to be overcome for any recovery schemes that use checkpoint comparison: the remote restartability and comparability of a checkpoint. That is, a task must be able to be restarted from a checkpoint produced on other nodes, and a checkpoint produced on a node must be identical to any checkpoint from any other nodes if both are correct and for the same computation. The former is required for process replication (lookahead execution), while the latter is needed for checkpoint validation.

Due to different workloads at each node, the processing speed may vary on each node. In our recovery scheme, the task execution time is determined by the slowest process in the replicated process pair. The mismatch in processing speed (or workload) prolongs the completion of the task computation. It also causes the problem of uncommitted checkpoints accumulating in the file system.

4.4. Checkpoint Construction

4.4.1. Checkpoint structure

A checkpoint used in our implementation is a snapshot of a process *run-time image* at the time of checkpointing. There has been considerable research concerning checkpoint construction in UNIX [11,38–40]. Smith implemented a mechanism for checkpoint construction in UNIX for the purpose of process migration [38]. His checkpoint is an executable file generated by a checkpoint operation. It contains the text segment, the data segment, as well as the stack segment of the process state. The stack segment is treated as a part of the data segment. The processor state (e.g., registers) is saved by a *setjmp()* system call. The restart of the checkpointed process is simply the reexecution of this executable file on another processor. Li and Fuchs developed a checkpointing scheme for their compiler-assisted checkpoint insertion techniques [11]. Their checkpoint is a data file that contains the data segment and partial stack segment of the checkpointed process. The checkpoint is intended for use in the same shell process on the same machine.

Our checkpoint structure is a superset of that of Li and Fuchs. In addition to having the complete stack and data segments, our checkpoint also contains a segment for the file I/O output data during that checkpoint interval. The inclusion of the file output as a part of a checkpoint makes checkpoint comparison effective for error detection (described later). The process registers are saved as a part of the stack. The omission of the text segment is possible because the original executable file is already available through NFS. There is no need to transfer the executable file to perform a remote restart.

4.4.2. Checkpoint operations

The checkpoint/restart operations include three routines: *_checkpoint()*, *_restart()*, and *_terminate()*. They are all user level and can be placed into user applications either manually or automatically by a compiler [11]. The routines are described as follows:

- *_checkpoint()* is placed in user application programs. When executed, it saves the processor state on the stack, stores both data and stack segments in a data file and signals to the voter that a new checkpoint has been generated.
- *_restart()* is inserted in *main()* as the first thing to execute. It checks with the voter if there is a checkpoint from which to start. If there is a checkpoint, it reads in and restores the data and stack segments, and resumes from the checkpoint; otherwise, it does a normal return.
- *_terminate()* is inserted before every *exit()*. When executed, it signals to the voter that the task has terminated.

4.4.3. Restartability

The virtual and uniform memory layout of UNIX in homogeneous machines provides the basis for the restart of a checkpointed process on a remote node. However, some user process information is usually kept in kernel for efficiency. A checkpoint without this information may not be restartable even for the same kernel. One example is the file I/O information in the file descriptor table in the kernel. When a process terminates or aborts, this information is cleared by the kernel. Restarting a process from a checkpoint without reestablishing this information in another kernel makes a local file descriptor in a user program meaningless.

To make a checkpoint remotely restartable, the user information kept in the kernel has to be extracted during checkpointing and reestablished to the new kernel at restart [38,39]. A set of library routines was developed for file I/O operations. The library keeps extra data as a part of the checkpoint, such as file name, access mode, and file position, associated with the opened files. During checkpointing, all file buffers are flushed for opened files, and the file positions are updated and stored in the checkpoint. During a restart, those files are reopened and repositioned according to the previously saved information in the checkpoint. In this manner, the attributes of file I/O can be saved and restored easily across the network. These file I/O routines together with *_checkpoint()*, *_restart()*, *_terminate()* comprise the checkpoint library. Using compiler-assisted techniques, these file I/O routines can be substituted transparently for their corresponding calls to the standard I/O library in user programs [11].

Even with the complete information of a user process state, the checkpoint may still not be restartable. In UNIX, some state attributes are kernel-dependent. They cannot be saved and carried across kernels (i.e., nodes) in a sensible fashion [38,39]. Examples are *process group*, *signal* received, the value of the real-time clock, and any children the process may have spawned with *fork()*. Similar to CONDOR and Smith, our current implementation assumes that for restartability a program may not use or depend on those kernel-dependent attributes that have partial information internal to the operating system other than file I/O.

4.4.4. Comparability

The kernel-dependent attributes also cause checkpoints to be incomparable, even if these checkpoints are all valid. For example, the value of the real-time clock for different kernels may be different, since these clocks are seldom synchronized. The valid checkpoints from the same execution on different nodes may not be the same if the program has these attributes as a part of its memory space.

For those kernel-dependent attributes, we enforce the following restrictions to make the checkpoint comparable: we can eliminate the use of variables to store such kernel-specific attributes, or carefully place them in local variables (on the stack) whose scope does not include a checkpoint operation, or clear these variables before checkpointing. Fortunately, most numeric applications seldom use kernel-dependent values except file I/O, and thus meet the restrictions we put on checkpoint restartability and comparability.

4.5. Voter and Recovery Management

In our DMR-F-1 implementation, the checkpoint comparison and recovery initiation are managed by a voter process running on a Sun NFS server. Our current implementation assumes that the voter is reliable. During a checkpoint operation, the communication between the voter and the tasks being executed uses the Internet *sockets*. At each processing node, there is a simple RPC-based (Remote Procedure Call) daemon process that schedules or terminates a task in that processing node on behalf of the voter. Originally, a **rsh** call from the voter was used for scheduling and killing a task on a remote node. However, we found the performance overhead of a **rsh** call to be unacceptable. Therefore, the voter was designed in such a way that it can schedule and terminate a remote task, compare checkpoints, and initiate a restart across a network.

The voter is invoked with the name of the task program to be executed and its arguments. The voter then schedules two replicated processes for this program and waits for messages from the scheduled tasks. When a task process is initiated, the call to *_restart()* sends a *register_msg* to the voter. Upon receiving it, the voter sends back a checkpoint file name if recovery is needed. Otherwise, the voter replies with no checkpoint and the task does a normal start. When a task makes a checkpoint with *_checkpoint()*, the voter receives a *checkpoint_msg*. The voter either advances to the next checkpoint interval or does a lookahead/validation operation, depending on whether the checkpoint comparison fails or not. When a task is terminated, the voter receives an *exit_msg*. If all replicated tasks have exited, the voter terminates.

In a distributed environment, the processing speed of processing nodes may vary due to differences in hardware and workload. This mismatch in processing speed causes the replicated tasks to lag behind one another. Therefore, the task completion time is prolonged since it is a function of the slowest of the replicated pair. In addition, the uncommitted checkpoints produced by the faster task can accumulate in the file system. In a distributed environment, a checkpoint may be a natural place for migrating processes and redistributing workload. We added a simple mechanism in the voter algorithm to adjust the performance of the replicated task executions. If the voter detects a growth by two in the checkpoint count for a task, the two replicated tasks are switched to other nodes.

4.6. Experiments

4.6.1. Benchmark programs

Two criteria for selecting programs are adopted in our experiments: (1) representativeness across different checkpoint sizes and (2) ease for checkpoint placements. Checkpoint size is very important since it determines the overheads such as checkpoint time, restart time and comparison time. The structure of most scientific programs (an obvious main loop) can give a nearly equal checkpoint spacing by a simple manual technique (see Section 4.6.2). Four scientific and two SPEC benchmark programs with different checkpoint sizes were selected for our experiments [41]. They are described as follows:

convlv: is the FFT algorithm for finding the convolution of 1024 signals with one response. The length of each signal was 256 bytes. The length of the response was 99 bytes. The size of the entire data set was over 1 megabyte (M) but the size of the memory-resident data set was only a few kilobytes (K).

ludcmp: is an LU decomposition algorithm that decomposes 100 randomly generated matrices with size uniformly distributed from 50 to 60. Although it has a larger data set (2.4 M) than that of **rsimp**, this program occupies less main memory because memory is reused, i.e., a memory block is allocated before a new matrix is read in and is deallocated after the result is written out.

matrix300: is a SPEC bench program. It performs various matrix multiplications, including transposes using Linpack routines SGEMV, SGEMM and SAXPY, on matrices of order 300. It produces no output during computation and has large resident data in memory (2.2 M).

nasa7: is a modified version of NASA Ames FORTRAN kernels from SPEC. It consists of seven floating-point intensive modules. The original version uses a large memory and generates heavy paging activities on a NFS server that cause lengthier execution on diskless workstations than it would be on a single machine (44 hours vs. 4 hours). We changed some array dimensions so that paging would not delay our experiments (250 K data and about 2 hour execution). There is no data output during execution.

rkf: is the Runge-Kutta-Fehlberg method for solving the ordinary differential equation $y' = x + y$, $y(0) = 2$ with step size 0.25 and error tolerance 5×10^{-7}. A table of function values was generated for $x = 0$ to 1.5 every 0.0001. This is a computation-intensive program with a small data set.

rsimp: is the revised Simplex method for solving the linear optimization problem for the *BRANDY* set from the Argonne National Laboratory. One characteristic of this program is its large memory-resident data set (about 1 M). There is no file output during execution.

4.6.2. Checkpoint placement

The checkpoint validation test using checkpoint comparison in DMR-F-1 requires that the checkpoints compared must be produced from the same computation. That is, the checkpoints inserted have to have fixed locations in the program execution. The checkpoints inserted by the Li and Fuchs' CATCH may be changed for different execution runs of the same computation, since their approach is based on polling the real-time clock. For more details, see Chapter 5. For now we use a simple polling method based on a selected loop count to insert checkpoints in user programs. Since this method does not depend on the real-time clock, it can insert checkpoints that are fixed in the program execution. This method is depicted in Figure 4.1. The overhead caused by maintaining and testing loop count is likely to be negligible, since the loop we selected is often the significant loop in a program. During execution, the number of loop iterations is usually small; thus the insertion overhead is also small.

With the selected loop count method, *_checkpoint()* is inserted manually to maintain roughly a constant checkpoint interval. Table 4.1 shows that this insertion scheme gives an adjustable checkpoint interval by choosing a proper threshold with a small variance. Our observation on the optimal checkpoint placement in Section 3.3.11 suggested that the optimal checkpoint interval is typically very large and insensitive to small displacement for

```
main()
{
        int threshold = certain_value;
        int count = 0;
        _restart();

        while (expr) {
             // major work here
             if ((count = (count+1) % threshold) == 0)
                 _checkpoint();
        }
        _terminate();
}
```

Figure 4.1. Checkpoint Placement Using Selected Loop Count.

Table 4.1. Overhead Measurements.

Programs Name	# ckp (per run)	ckp_size (data/stack/file) (in bytes)	ckp_time (std. dev.) (in sec)	cmp_time (std. dev.) (in sec)	ckp_interval (std. dev.) (in sec)	exec_time (w/o. ckp) (in sec)
convlv	128	75950 (66196/1554/8200)	0.2172 (0.3411)	0.1608 (8.6302e-3)	13.917 (0.90787)	1809.22 (1781.42)
ludcmp	50	121510 (71708/1550/48252)	0.2408 (3.428e-2)	0.2030 (1.8224e-2)	20.626 (2.1092)	1043.38 (1031.34)
matrix300	150	2219446 (2217652/1794/0)	5.8714 (0.6949)	8.6157 (0.2338)	239.777 (26.729)	37092.88 (36206.30)
nasa7	49	351614 (349788/1826/0)	0.7672 (0.1347)	0.9660 (5.683e-2)	131.46 (28.22)	6611.44 (6573.00)
rkf	88	51777 (46972/1734/3071)	0.1477 (2.563e-2)	0.1492 (7.2498e-3)	29.7202 (1.0840)	2638.58 (2625.58)
rsimp	59	995314 (991676/3638/0)	2.411 (0.3767)	3.8286 (0.21893)	42.8063 (8.6359)	2713.04 (2568.38)

DMR-F-1. Therefore, this small variance in checkpoint interval due to the selected loop count insertion still gives the optimal or near optimal solution.

4.6.3. Error injection

In our experiments, errors are injected into checkpoints to study the recovery behavior of the programs. An error is injected into a checkpoint by randomly flipping bits in the data or stack segment of the checkpoint. This type of error is intended to model the changes in variables due to possible errors in memory and in data manipulation (ALU). We avoid the injection of run-time errors because it may result in an incomparable checkpoint.

The probability of a node failure during a checkpoint interval in our experiments, p_f, is constant as we assumed in Chapter 3. The range of p_f is selected as [0, 0.1]. Each program is run five times for each failure probability to obtain the average measures.

4.6.4. Program characteristics

The overhead measures we considered consist of the checkpoint size (ckp_size), checkpointing time (ckp_time), and checkpoint comparison time (cmp_time). Other measures of interest are the checkpoint interval (ckp_int) and the error-free execution time with and without checkpointing. The restart time of a task is of about the same order as the checkpointing time in magnitude and is not listed explicitly. Table 4.1 summarizes these overhead measures for each program. The checkpoint size consists of three parts: data segment, stack segment and the file output during the checkpoint interval. Programs **Rsimp** and **matrix300** give examples of a large checkpoint. Most applications we examined have checkpoints of size (64-350 K). The stack size is small in all six programs. This is not unexpected for scientific applications in which the calling depth is rather limited. The file output size can be large in some applications (e.g., **convlv**).

In Table 4.1, both *ckp_time* and *cmp_time* do not include the processing time for the file output portion of the checkpoint. For *ckp_time*, the file output portion is already written to disk during execution; thus, it is not necessary to rewrite this portion to the checkpoint. Three variables in a checkpoint are enough to locate this file output portion (file name, starting position and length for each output file). The small standard deviation in the checkpoint interval indicates that the selected loop count insertion of *_checkpoint()* has produced a nearly constant checkpoint placement. We have found that checkpoint time, comparison time and restart time are highly correlated. Since file I/O operations are the major part of checkpointing (write), checkpoint comparison and restart (read), the overhead costs such as checkpoint time, comparison time and restart time can be expected to be proportional to the size of the checkpoint files.

4.6.5. Error detection by checkpoint comparison

The effectiveness of checkpoint comparison is studied for the six selected programs. To avoid the interference of run-time error injection with checkpoint comparability, a random bit or word error is injected in the previous checkpoint to model a transient error occurrence during its subsequent checkpoint interval. Then one task is started from this erroneous checkpoint and another task from the error-free checkpoint. The checkpoints produced by the two tasks after one checkpoint interval are compared. A mismatch indicates a detected error. Table 4.2 summarizes the results for 101 injected random errors. The number of errors detected is categorized by where the error is detected: the data, stack and the file output

Table 4.2. Error Detection Through Checkpoint Comparison.

Program	bit-wise errors					word-wise errors				
	# Errors Detected				#	# Errors Detected				#
	data	stack	file	abort	Missed	data	stack	file	abort	Missed
convlv	68	3	30	0	0	71	0	30	0	0
ludcmp	43	0	58	0	0	37	3	59	2	0
matrix300	101	0	-	0	0	100	0	-	1	0
nasa7	87	0	-	0	14	87	0	-	0	14
rkf	78	1	22	0	0	76	3	22	0	0
rsimp	99	0	-	2	0	98	0	-	2	1

segments of the checkpoints. The abortion of the task due to an error in the checkpoint can be treated as a special case of error detection by sending an abortion signal to the voter explicitly.

The errors detected by checkpoint comparison account for the majority of injected errors that occurred (about 98%) for all programs except **nasa7**. If the file output during the checkpoint interval is not included in the checkpoint structure, 22 to 59% of the errors would not be detected (**rkf**, **convlv** and **ludcmp**). Some errors were missed in our experiments. In this case, we have a valid file output during execution and a valid checkpoint at the end; the missed errors are actually masked off and cause no problems with respect to correct executions. This case occurs when an error is in a dead variable and this variable is reinitialized later. A close look at the checkpoint placement for **nasa7** reveals that a new array of about 11% of the total checkpoint size is computed during the checkpoint interval. The 14 missed errors were probably inserted into the new array space and were overwritten during the computation. In sum, the checkpoint structure provided an effective error detection tool for the programs we studied.

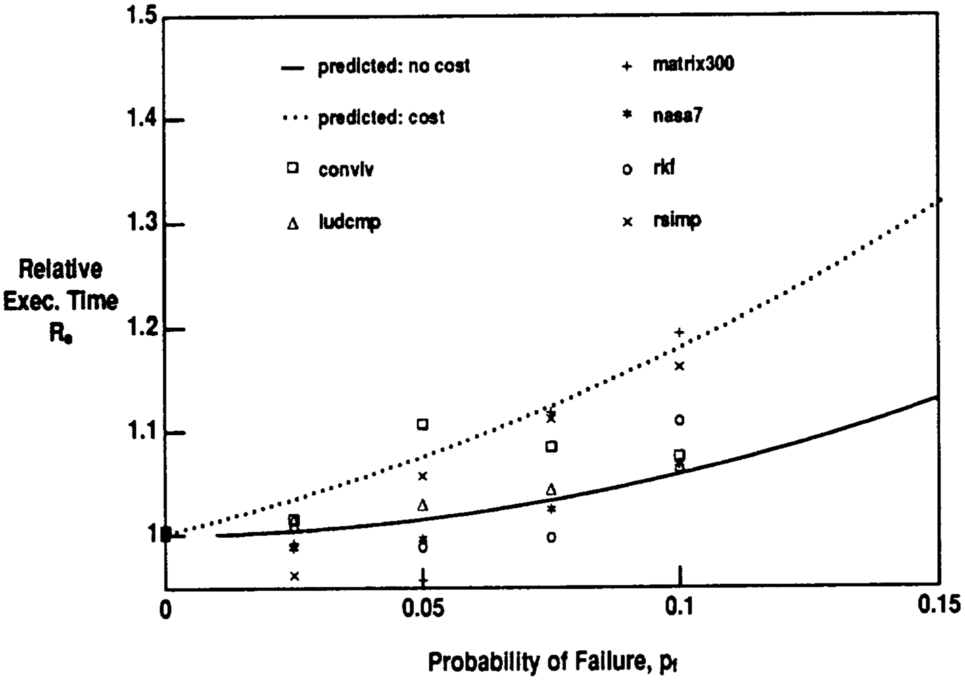

Figure 4.2. Relative Execution Time During the Day.

4.6.6. Performance results

Each program was run five times for each p_f in order to obtain the average measures. The execution time in our experiments is actually the program response time. It includes the system, user and blocking times. The analytical predictions for the relative execution time, number of processors, and number of checkpoints are also included in Figures 4.2, 4.3 and 4.4 to compare against our experimental results (Chapter 3). The voter processing overhead is the time spent in the voter program minus the checkpoint comparison time and divided by the checkpoint interval.

The data were collected for two workload conditions: daytime (10 AM to 6 PM) and nighttime (10 PM to 8 AM). During the day, the workload among our workstations was uneven and the NFS server was busy. During the night, our workstations and the NFS server were lightly loaded.

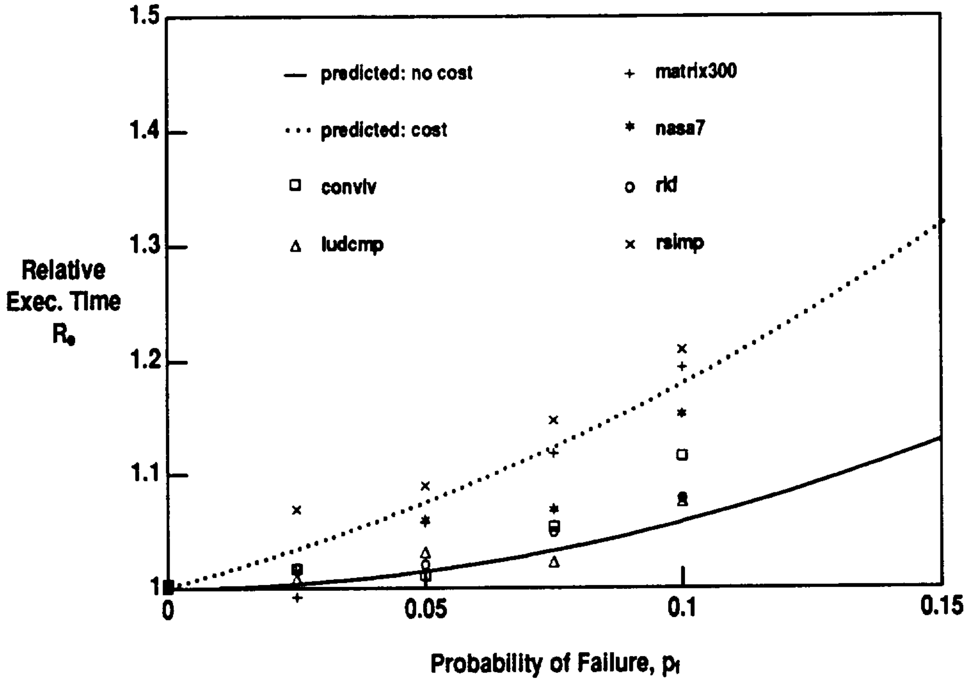

Figure 4.3. Relative Execution Time During the Night.

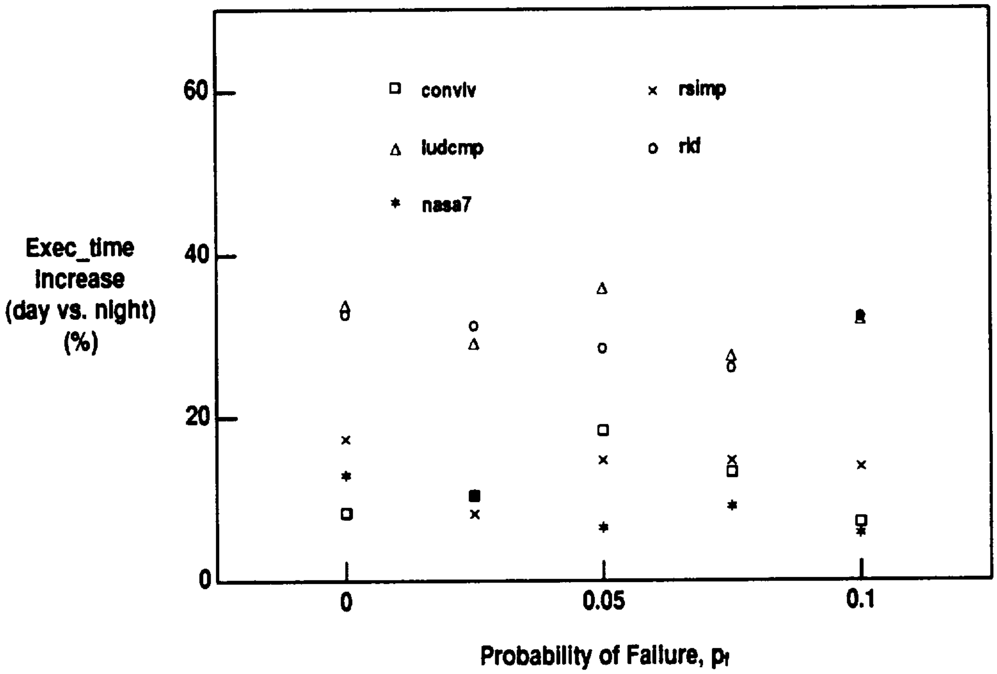

Figure 4.4. Difference in Execution Time.

In Figures 4.2 and 4.3, the relative execution time is plotted for each of the six programs under two different workload conditions. The relative execution time for the programs with a moderate checkpoint size (**ludcmp**, **convlv**, **nasa7** and **rkf**) is close to the analytical zero-overhead prediction (solid curve), since the overheads for those program are very small compared to their checkpoint intervals. The relative execution time for the program with large checkpoints (**matrix300** and **rsimp**) fits well with the analytical prediction under a centralized file server (the dotted curve, assuming an overhead level of **rsimp**). This increase in execution time for large checkpoints can be explained by the fact that **matrix300** and **rsimp** are likely to be blocked due to its large file I/O operations during checkpointing and comparison. In fact, the limited speed of the NFS file handling and our use of the file server for managing checkpoints centrally resulted in a performance bottleneck. The paging activities from the replicated processes also contribute to the increase in execution time. The relative execution time increases significantly for high error rates due to the heavy file server activities during checkpointing and comparison of checkpoints. This suggests that a reduction in checkpoint size, an increase in file system speed, or other noncentralized server implementations may improve the relative execution over that of our current implementation. In addition, a reduction in comparison time may significantly reduce the execution time of DMR-F-1 (Section 3.3.9).

The relative execution time fluctuates more for the daytime condition than that for the night. The execution time is longer in the day runs than in the night (Figure 4.4). This reflects the fact that the workload is heavier and more likely to change during the day than during the night.

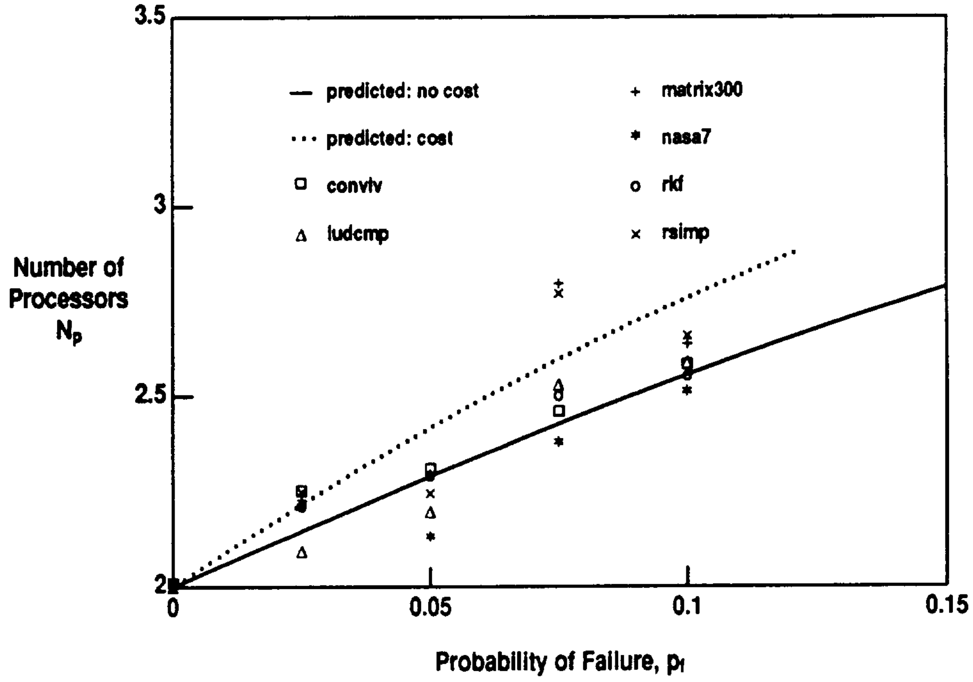

Figure 4.5. Number of Processors During the Day.

For the p_f we considered, the number of processors used, N_p, is less than the three that TMR requires, although DMR-F-1 uses two more processors momentarily during lookahead/validation operations. As anticipated in Section 3.3.8, N_p is quite insensitive to the workload distribution conditions and checkpoint size (Figures 4.5 and 4.6).

The number of checkpoints, N_c, is highly sensitive to the workload and checkpoint size, as a result of the checkpoint accumulation in the file system due to uneven processor speed, especially for the programs with large checkpoint sizes (Figures 4.7 and 4.8). Without switching the task executions on the nodes that have different processing speed, N_c averaged 6.95 in five runs for **convlv**. In one of these runs, N_c reached 18.45. With the switch mechanism mentioned in the previous section, N_c is limited to about 2 or 3. The simple switch rule we used in the voter limits N_c by redistributing the workload.

In Figure 4.9, the average processing overhead for the voter is plotted. The overhead is small compared to the checkpoint interval, and increases as the failure probability increases.

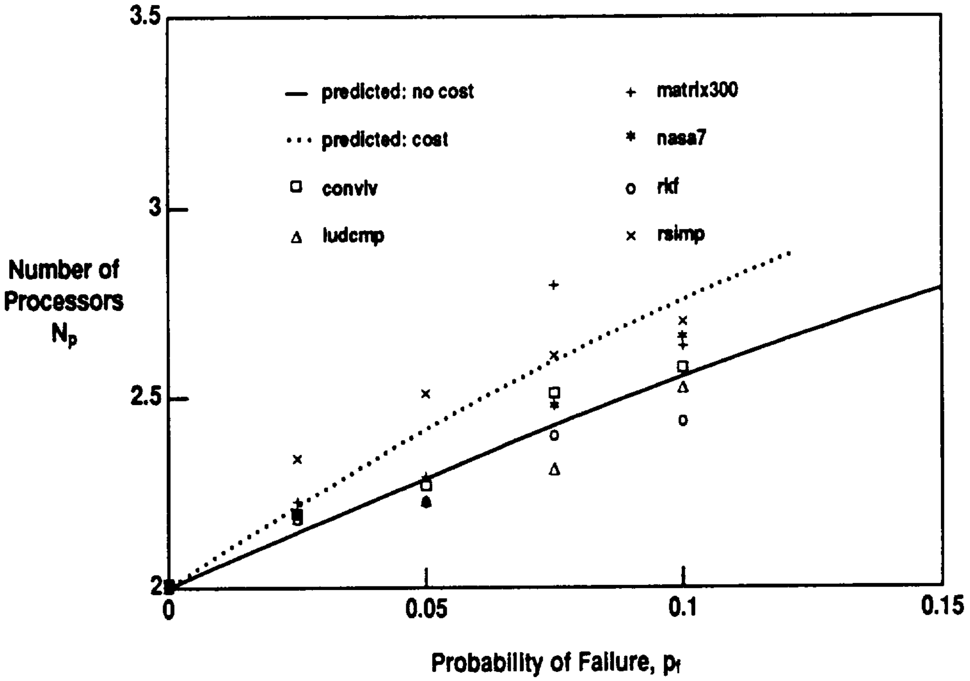

Figure 4.6. Number of Processors During the Night.

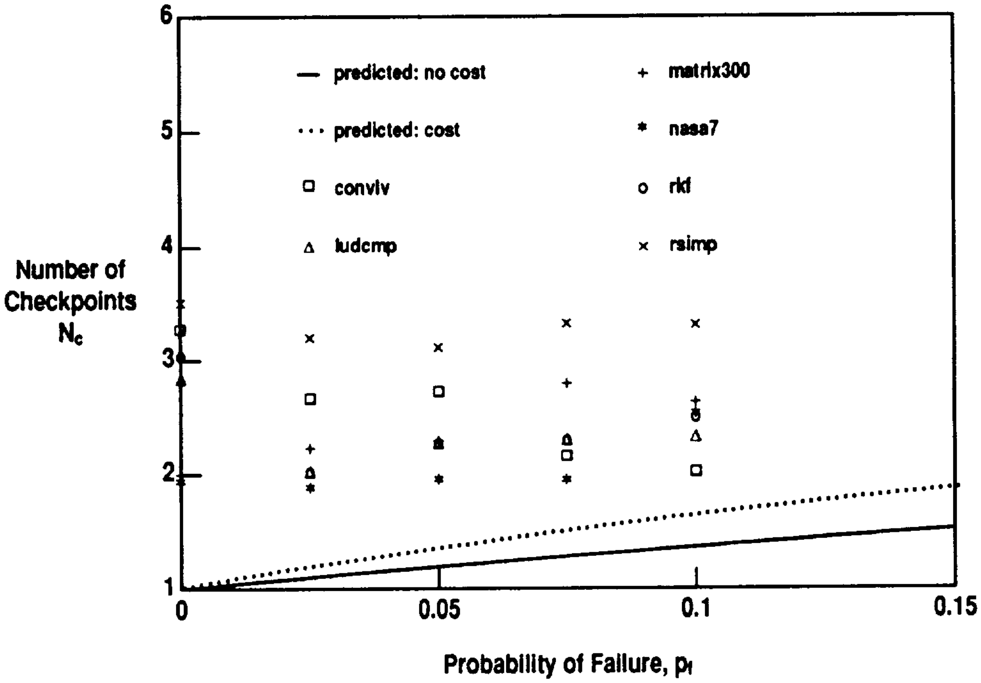

Figure 4.7. Number of Checkpoints During the Day.

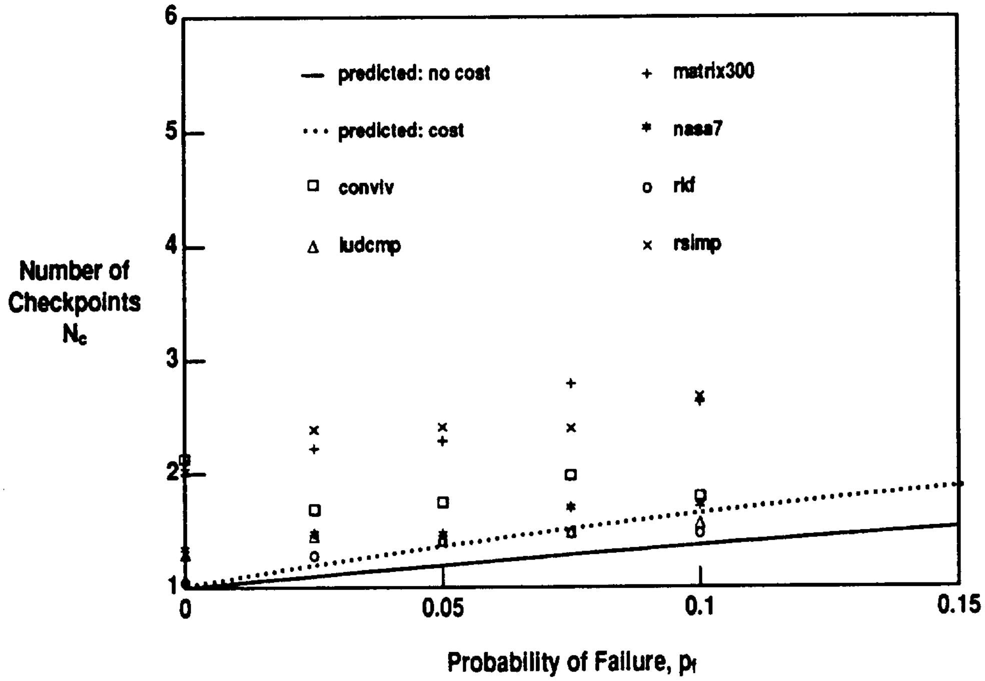

Figure 4.8. Number of Checkpoints During the Night.

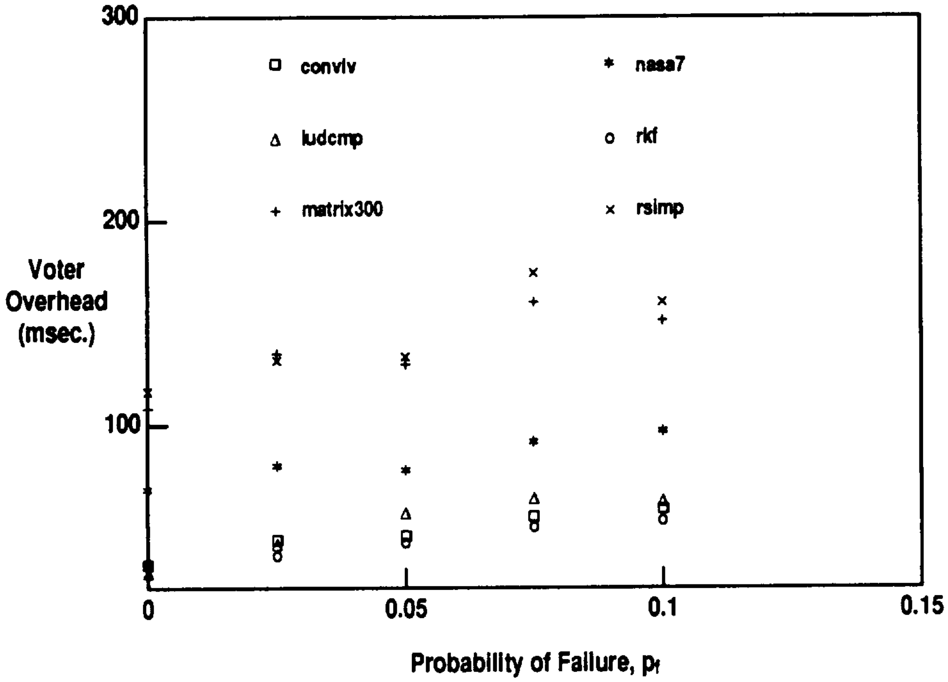

Figure 4.9. Voter Processing Overhead.

Large checkpoints also increase this overhead due to the waiting time for file I/O during checkpoint comparison.

4.7. Summary

In this chapter, we have described a distributed implementation of DMR-F-1. There is no universal way in UNIX to obtain a restartable and comparable checkpoint structure because of either the user information kept in the kernel for efficiency or the machine-dependent information. Our DMR-F-1 implementation was evaluated with six benchmark programs. The experiments have shown:

- Checkpoint comparison is effective means of error detection and checkpoint validation in our experiments. The file output during a computation session should be included as a part of the checkpoint for this computation session.

- The overheads, such as checkpoint time, restart time and comparison time, are highly correlated to the checkpoint file size.

- The experimental results agree with the analytical predictions given in Section 3.3.2. For moderate checkpoint sizes (64-350 K), the experimental results are close to that of the deal zero-overhead case; the average execution time is reasonably close to the error-free execution time. For large checkpoint sizes, the experimental results are close to what the analytical model predicts.

- For moderate checkpoint sizes, the checkpoint and comparison times are small, compared to the checkpoint interval. For large checkpoint sizes, these overheads have an important impact on performance due to the centralized file server used in our implementation.

- The workload also has an important impact on the number of checkpoints and the execution time. The former resulted from the uneven distribution of the workload and the latter from both workload distribution and workload level.

- The average number of processors used in our forward recovery scheme is less than TMR.

CHAPTER 5.

COMPILER-ASSISTED STATIC CHECKPOINT INSERTION

5.1. Introduction

Error recovery using checkpointing and rollback is a common strategy in fault-tolerant systems because it can handle unanticipated errors [7]. Considerable theoretical research has been devoted to determining optimal checkpoint intervals [1–6]. A practical problem in implementing checkpointing and rollback recovery is the maintenance of the desirable checkpoint interval. Checkpoints may be static in the sense that they are at fixed locations in a program or they may be dynamic such that their locations in a program may vary, as a function of time or system behavior. Although dynamic checkpoints can be implemented with existing hardware interrupt support (system clock), they are not reproducible. Static checkpoints must rely on either the insertion of checkpoints before program execution or the monitoring of program behavior during execution. Reproducible checkpoint intervals, as obtained with static checkpoints, can be used for debugging [42–45] or error detection by means of checkpoint comparison with replicated processes [46,47].

Chandy and Ramamoorthy have developed a scheme for application-level checkpoint insertion, given a computation sequence, execution time, checkpoint time and restart time [48]. Their scheme is a graph-theoretic method to determine the optimal locations for checkpoint placement. Toueg and Balaoglu, and Upadhyaya and Saluja followed a similar approach [2,49,50]. Li and Fuchs have studied techniques for checkpoint placement at the compiler level (CATCH) [11]. Checkpoint subroutines are transparently inserted in the user program by the compiler. Li's CATCH is a dynamic checkpoint insertion scheme. To maintain the desirable checkpoint interval, the real-time clock is polled to decide if a checkpointing call is due. This polling code is called a potential checkpoint in their paper. To reduce excessive polling at these potential checkpoints, a leverage count is inserted in loop bodies. The potential checkpoint is activated once the leverage count is greater than a threshold value. Polling the real-time clock can result in different checkpoint locations for different execution runs of the same computation due to the clock granularity (one second in Unix) and the workload on the system.

This chapter presents a compiler-assisted approach for *static* checkpoint insertion. Instead of fixing the checkpoint locations before program execution, a compiler-enhanced polling mechanism is utilized to maintain both the desired checkpoint intervals and reproducible checkpoint locations. Instruction-based time measures are used to track the computation progress and thus checkpoint intervals. These measures produce static checkpoints by eliminating the real-time clock. This approach has been implemented in a GNU CC compiler for Sun 3 and Sun 4 (SPARC) processors [51]. Experiments demonstrate that our approach provides for scalable checkpoint intervals and reproducible checkpoint placements with a performance overhead that is less than that of the previously presented compiler-assisted dynamic scheme (CATCH).

Section 5.2 describes the problem of compiler-assisted checkpoint insertion. Section 5.3 presents the concept of instruction-based time measure and four static insertion schemes. Section 5.4 discusses our implementation and experimental results.

5.2. Checkpoint Insertion Problem

There are two basic problems in inserting checkpoints in a user program: how to design a set of operations to accomplish checkpointing and recovery, and how to maintain a desired checkpoint interval. Optimal checkpoint intervals may be the desired checkpoint spacing [1–6].

5.2.1. Checkpoint operations

Compiler-assisted checkpoint insertion requires a set of checkpoint operations to accommodate checkpointing, to restart from a checkpoint and to perform checkpoint cleanup at the end of the computation. In Chapter 4, we have discussed a checkpoint library that provides such a set of checkpoint operations for UNIX. These operations (subroutines) can be divided into two categories.

The operations that perform recovery and checkpointing are the basic operations, since they are required no matter what the operating system may be. The compiler actually inserts the basic operations into the user program. In the checkpoint library developed in Chapter 4, *_restart()* provides a recovery entry point in the user program. It is usually inserted in the beginning of the user program. If a recovery from a checkpoint is needed, it restores the system state and resumes execution from this state. The *_checkpoint()* call creates a checkpoint when executed; *_terminate()* clears the checkpoints at the end of the computation as the last operation in the user program. In UNIX, *_restart()* may be inserted as the first line in the *main()* routine, while *_terminate()* is inserted before every *exit()* call. A *_checkpoint()* is inserted wherever a checkpoint is desired in the user program.

The interface operations are operating system specific, and they often assist the basic operations to accomplish checkpointing and recovery. The interface operations are usually substituted for some routines in the user program by the compiler. For example, *_fopen()* in the file I/O interface routines described in Chapter 4 is used to separate the process state from the operating system kernel to obtain a remotely startable checkpoint. It takes the same arguments as *fopen()*, stores these arguments in the user space and then calls the true *fopen()*. During recovery, these retained arguments can be used to reopen the file without asking the kernel to roll back. During compilation, the compiler simply substitutes *_fopen()* for each occurrence of *fopen()* in the user programs.

In the following, we will not consider the problem of checkpoint operations further, since we already studied this problem in detail in Chapter 4.

5.2.2. Checkpoint interval maintenance

Maintaining desirable checkpoint intervals by a compiler requires a time measure for describing checkpoint intervals and a mechanism to insert a checkpoint operation in the user program. Using the elapsed time of a computation as the time measure leads to dynamic checkpoints (CATCH). This is because the elapsed time for a computation often varies from execution to execution due to resource sharing with other computations. Static checkpoint insertion needs a time measure that is independent of the real-time clock and that describes checkpoint interval in terms of computation progress. In Section 5.3.1, we will introduce instruction-based measures that satisfy both requirements.

Ideally, the compiler should insert checkpoint operations into the user program prior to the program execution. There are three difficulties in doing so. First, the exhaustive search for all possible execution paths in a program to determine checkpoint locations is computationally intractable in general. Second, to calculate the execution length along a particular path may be impossible due to the lack of information at compilation time (e.g., data sensitive loop bounds) [52]. Third, each path may require a different set of checkpoint locations. It is possible to have many insertions on a common path, but this leads to redundant insertions.

As a result, the checkpoint interval is maintained by either an interrupt driven mechanism or a polling mechanism. Figure 5.1 presents an interrupt driven scheme for UNIX-based systems. The system interrupt initiates checkpointing requests, based on the system's real-time clock. Upon receiving a time-out interrupt, a checkpointing operation is executed as a part of the interrupt service. Using this interrupt driven mechanism, the compiler-assisted insertion is trivial. The compiler needs only to insert *restart()* at the beginning of the *main()* and *cleanup()* before every *exit()*. No explicit insertion of *checkpoint()* is necessary. This interrupt driven insertion can maintain desirable checkpoint intervals at a low cost, since the hardware interrupts can initiate checkpointing requests efficiently. However, the interrupt mechanism based on the real-time clock generally cannot guarantee a specific checkpoint location in a program, and thus it is a dynamic insertion.

The polling mechanism, when activated, tests whether a checkpoint is needed given the desired checkpoint interval. If a checkpoint is needed, the checkpointing operation (routine) is executed. This is called a polling point in this thesis. Using the polling mechanism, the compiler has to deal with two problems: the polling point placement and the management of the execution time since the last checkpoint (Figure 5.2). For a polling mechanism to be successful, the overhead resulting from the time measure maintenance and checkpoint

```
#include <signal.h>

int checkpoint()
{
        /* make a checkpoint here */

        /* schedule the next checkpoint operation */
        signal(SIGALRM, checkpoint);
        alarm(the_desired_checkpoint_interval);
}

int restart()
{
        /* Check if there is a checkpoint to restart. If yes,      */
        /* read in checkpoint and restore system state. Otherwise, */
        /* do nothing.                                             */

        /* schedule the next checkpoint operation                  */
        signal(SIGALRM, checkpoint);
        alarm(the_desired_checkpoint_interval);
}

main()
{
        restart();  /* recovery entry point & start checkpoint     */
                    /* interrupt. No explicit checkpoint() is      */
                    /* needed.                                     */

        ...

        cleanup();  /* clean up checkpoints                        */
        exit(0);
}
```

Figure 5.1. Interrupt Driven Checkpoint Insertion in UNIX.

polling should be reasonably small. In CATCH, the real-time clock is used for the polling measure and loop iteration is used for the polling location. Using the real-time clock as the polling measure leads to a dynamic insertion, since the workload in the environment and the blocked operations in user programs can affect the polling decision in different execution runs.

5.3. Static Checkpoint Insertion

5.3.1. Instruction-based time measure

The essence of checkpoint and rollback recovery is to save the previously finished computation and to avoid restart from the beginning every time under the presence of faults. The elapsed time for a computation is not suitable to describe the progress of the computation precisely, since the computation time usually changes with the system load because of the resource sharing between processes in the system. Therefore, maintaining checkpoint intervals using the elapsed time of the computation leads to dynamic checkpoints, such as in CATCH. For static checkpoint insertion, the computation progress measure should be independent of the real-time clock and depend only on the computation itself. The instruction-based time measure introduced below is such a measure.

The sequence of instructions that has been executed by a computation is called a trace in this thesis. Let T be a trace of a program execution such that T=$\{I_1, I_2, \ldots, I_j, \ldots\}$ where I_j is the j-th instruction executed in this trace. A subsequence s of a trace T is called a subtrace of trace T (denoted as $s \subseteq T$). A measure m on a trace T is said to be a instruction-based time measure if $m(s_1) \leq m(s_2)$, given $s_1 \subseteq s_2$, and $s_1, s_2 \subseteq T$.

```
#include <signal.h>
/* on-fly time measure accumulation */
void update_time_measure(measure)
    int measure;
{
        /* measure is calculated by the compiler and is a constant */
        measure_since_last_checkpoint += measure;
}
void polling_point()
{
        if (measure_since_last_checkpoint >= measure_threshold) {
                checkpoint(); /* explicit checkpoint() call is needed */
        }
}

void checkpoint()
{
        /* make a checkpoint here */
}

void restart()
{
        /* Check if there is a checkpoint to restart. If yes,      */
        /* read in checkpoint and restore system state. Otherwise, */
        /* do nothing.                                             */
}

main()
{
        restart(); /* recovery entry point & start checkpoint      */
                   /* interrupt.                                   */

        ...

        update_time_measure(measure);
        polling_point(); /* polling point inserted here.           */

        ...

        cleanup(); /* clean up checkpoints                         */
        exit(0);
}
```

Figure 5.2. Polling-based Checkpoint Insertion in UNIX.

There are two properties associated with the definition of the instruction-based measure. First, it describes the progress of computation in terms of instruction traces. That is, an instruction-based time measure is a monotonously nondecreasing measure with respect to the instruction sequence in a computation trace, given a common starting instruction. If both subtraces start from the same instruction, a large value of m implies a longer computation than does a small m. Second, there is a time elapse for executing a subtrace $s \subseteq T$ such that $m(s) = m$. This time elapse is called the execution time of traces with a measure of m and is denoted as $t(m(s))$ or simply $t(m)$. In this chapter, the value of $m(s)$ is sometimes called the trace size of s for short.

However, the execution time of a trace size, m, may not be the same even when the execution is run on the same machine. For example, subtrace $s_1 = \{I_i, I_{i+1}, \ldots, I_{i+k}\}$ and subtrace $s_2 = \{I_i, I_{i+1}, \ldots, I_{i+k}, I_{i+k+1}\}$ may have the same measure by the definition above even with $s_1 \subseteq s_2$. Generally, s_1 and s_2 will result in a different execution time. We can model the execution time of a trace size with $t(m) = t_\mu + T_\sigma$, where t_μ is a constant for the expected execution time and T_σ is a random variable for the variation in execution time. The constant t_μ can be used as the time measure for computation progress, while the random T_σ can be viewed as the accuracy of the instruction-based measure with respect to elapsed time.

There are many possible instruction-based time measures. We will now describe four of them. They generate a time measure for checkpoint interval maintenance with potentially different accuracies.

- *Instruction cycle count*: The instruction cycle count (ICC) is defined as the total number of cycles for individual instructions in a trace. It is an instruction-based time measure. In fact, for any subtraces of $s_1 = \{I_i, I_{i+1}, \ldots, I_{i+k}\}$ and $s_2 = \{I_i, I_{i+1}, \ldots, I_{i+k}, I_{i+k+1}, \ldots, I_{i+k+j}\}$, we have

$$ICC(s_2) = ICC(s_1) + \sum_{n=1}^{j} cycles(I_{i+k+n}) > ICC(s_1).$$

The ICC measure can generate an accurate execution time. As a matter of fact, the deviation in execution time for ICC will be within the number of cycles for the longest instruction in a trace. If ICC is used for the checkpoint interval measurement, the resulting interval can be accurate within one instruction if a checkpoint can be inserted at any instruction boundary. However, this measure is architecture dependent as an instruction may have different execution cycles in different architecture.

- *Instruction count*: The instruction count (IC) is the number of instructions in a trace. Clearly the instruction count is an instruction-based time measure. Given an IC value, the trace execution time varies with the set of instructions in a trace. If the trace is large enough to have a representative mix of different instructions, the variance in execution time may be small. For a RISC machine, IC may be a very accurate time measure because of the uniform instruction cycle count (e.g., one instruction per cycle). Compared to the typical checkpoint interval, a few extra instructions in a trace will not affect the trace execution time in a noticeable way. In this case, IC can serve as a very good time measure for checkpoint intervals if a checkpoint can be inserted at the instruction boundary.

- *Loop/function count*: The loop/function count (LFC) is the number of loop iterations and functions in a trace. It can be shown that LFC is also an instruction-based time measure. The LFC measure changes only at a loop iteration or a function entry/exit point, and it does not distinguish between the individual instructions within a loop iteration. The execution time, given an LFC value, is potentially less accurate than that for ICC or IC. Different programs may have different loop sizes and thus different execution times. As will be shown, the number of instructions within one loop iteration is typically small. Compared to the checkpoint interval, the variation in the trace execution time caused by ignoring the instructions in a loop iteration is negligible. For LFC, a checkpoint may be inserted at a loop iteration boundary, a function entry point or exit point. Both IC and LFC are architecture independent since they do not depend on the execution cycles of particular instructions.

- *Selected loop/function count*: The selected loop/function count (SLFC) is the number of iterations for some loops selected. The selection of a particular SLFC affects the accuracy of the instruction-based measure with respect to execution time. If the selected loops spread over the whole execution trace evenly, the SLFC may generate a stable execution time (small T_σ). Otherwise, T_σ may be very large. The major loops in scientific programs may be good candidates for SLFC, since they are likely to be executed throughout the computation [47].

Ideally, ICC provides the most accurate time measure for a checkpoint interval. However, it may require the knowledge of the instruction set of a particular architecture and may consume more time in obtaining the ICC values. Moreover, the large size of an optimal checkpoint interval implies that the accuracy produced by the instruction cycle count is unnecessary since it results in very little changes in the realized checkpoint interval. Although LFC and SLFC are potentially less accurate than ICC and IC with respect to the trace execution time, they can be calculated with low cost. The accuracy may still be acceptable if the checkpoint interval contains many loop iterations so that a stable mix of instructions is executed in each checkpoint interval.

5.3.2. Checkpoint insertion schemes

We use a polling mechanism with instruction-based time measures to accomplish the static checkpoint insertion. The compiler calculates the instruction-based time measure along an execution path. These statically calculated values for the time measure are accumulated in a counter during the program execution on the fly. The accumulated counter gives the time measure since the last checkpoint. Based on the location of the time measure accumulation and polling points, the four schemes we have implemented are described in Table 5.1.

5.3.3. SLFC determination

In order to implement the SL-SL scheme, a method for selecting loops for SLFC was developed. Our approach is profile-based. Probe routines are placed into a program by the compiler. These probes collect the trace information during program profiling. The information collected is used to aid the loop selection for the SLFC measure. Once SLFC is determined, the compiler places static checkpoints in the program according to the SLFC measure.

Table 5.1. Four Static Checkpoint Insertion Schemes.

B-B:	This scheme measures the instruction count (IC). The code for both the time measure accumulation and polling is inserted in each basic block in the program. A basic block is a sequence of consecutive instructions in which the program control enters at the top and leaves from the bottom with no branches or halts inside. Basic blocks in this thesis are described in terms of RTL instructions.
B-L:	In this scheme, the time measure is the instruction count. The time measure accumulation code is inserted in each basic block, while that for polling is placed in each loop.
L-L:	Is scheme uses the loop/function count as the time measure. The code for time measure accumulation and polling is inserted in each loop and function.
SL-SL:	In this scheme, the time measure is the selected loop/function count (SLFC). The code for the time measure accumulation and polling is inserted only in the selected loops/functions.

There are two problems involved in selecting an SLFC measure: (1) to identify a set of loops that tend to appear throughout the execution trace, and (2) to determine a threshold value for each selected loop. This threshold value is important as the on-the-fly accumulated SLFC value is compared against this threshold value at each polling point in order to make a checkpoint decision. During profiling execution, each probe records the loop/function ID and calculates the frequency of occurrences of this loop in a checkpoint interval. If a set of loops can be found such that every checkpoint interval contains at least one loop from this loop set, this loop set may be a candidate for SLFC. The frequency associated with each loop for a checkpoint interval can be used as the threshold value for the loop.

Given a program and its profile data, the SLFC selection can be formulated as a cover set problem in a weighted bipartite graph. The checkpoint intervals and loop/function IDs are two sets of vertices. If a loop appears in a checkpoint interval, there is an edge between the checkpoint interval vertex and the loop vertex. The frequency of the loop occurrences in the checkpoint interval is the weight for this edge. The cover range of a loop vertex is the set of all of the checkpoint interval vertices that are connected to the loop vertex. An SLFC cover set is a set of the loops such that their cover range contains all of the checkpoint interval vertices.

There are four criteria for selecting a good SLFC cover set that gives a stable checkpoint interval with a small polling overhead:

- *Minimal overlapping*: The overlapping of cover ranges for two selected loops may result in unstable checkpoint intervals due to the interference of their possibly different threshold values.

- *Minimal cover set*: The size of an SLFC cover set is directly related to the code size overhead as the code inserted is proportional to the size of the cover set. Given that code size is not a problem for most applications, this criterion may be discounted during the selection of an SLFC cover set.

- *Minimal average frequency*: The average frequency for a loop in the SLFC cover set is used as the threshold value, for this loop, in our current implementation. A higher frequency leads to more frequent execution of the inserted checkpoint polling code for this loop and thus a higher run-time overhead.

- *Uniform Frequency*: This calls for a small variance in the frequencies for a loop in the SLFC cover set. As checkpointing is delayed for small frequency edges and is too frequent for large frequency edges, large variance in frequency weights results in a more unstable checkpoint interval.

Although finding a minimal cover set is NP-complete, finding a cover set with minimal and uniform frequency can be mapped into the problem of finding a minimal total weight cover set. In the current implementation, a heuristic algorithm is used to combine all of these criteria for SLFC selection (Figure 5.3). This heuristic is a greedy algorithm with different priorities for cover range, frequency average, and frequency variance. It selects loop vertices with large cover ranges and small frequencies under constraints of small relative frequency variance and little overlap for the selected loops.

5.4. Implementation and Experimental Evaluation

5.4.1. Base compiler and insertion filter

The base compiler that was selected to implement our static checkpoint insertion is the GNU CC compiler version 1.40 for Sun 3 and Sun SPARC. We used an implementation similar to CATCH. A register transfer language (RTL) level filter is placed between parsing and object code generation (Figure 5.4). The advantage of a checkpoint insertion filter is that the insertion is an independent module that is added to the base compiler and does not change the base compiler if the checkpoint insertion is not required. The choice of a register transfer level insertion is aimed at achieving an architecture independent implementation.

5.4.2. Benchmark programs

Six benchmark programs were used to examine our static insertion technique. Our objective was to study the effectiveness of the checkpoint interval maintenance in terms of

```
select
   [1] the number of checkpoint intervals that a loop
       covers as the primary key (in decreasing order);
   [2] the average frequency of a loop as the secondary
       key (in increasing order); and
   [3] the relative standard deviation in frequency
       for a loop (std. dev./average) as the third
       key (in increasing order).
sort the vertices according to the above keys.

cover_set = NULL;

/* set for no overlapping cover range */
overlapping_size = 0;

while (size(cover_set) < desired_coverage) do
{
     for each vertex v in the sorted loop_set do
     {
          /* select a v with uniform frequency */
          if (freq_variance(v) > threshold) continue;

          if (size(cover_range(v) and cover_set) <=  overlapping_size)
               add v to cover_set;

          if (size(cover_set) >= desired_coverage) break;
     }
     /* relax the overlapping constraint a little */
     overlapping_size++;

     if (no changes in cover_set) break;
}
```

Figure 5.3. Heuristic SLFC Selection Algorithm.

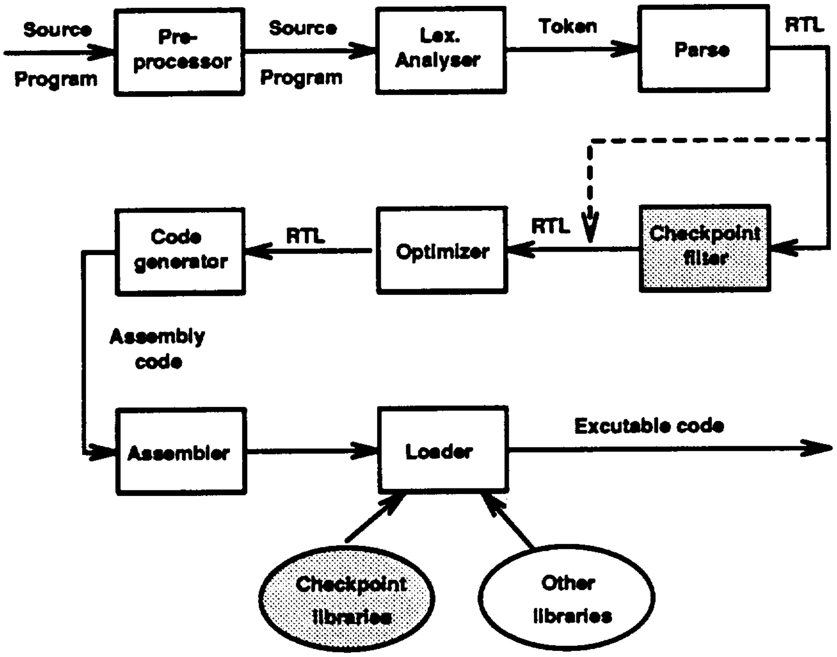

Figure 5.4. Base Compiler and Its Checkpoint Insertion Filter.

1. The average checkpoint interval and its variance (t_μ and T_σ in Section 5.3.1). This gives the effectiveness of an instruction-based time measure for checkpoint interval maintenance. A small variance implies that the instruction-based measure is accurate with respect to execution time.

2. Scalability of the checkpoint interval with respect to the instruction-based time measure threshold, for checkpoint polling tests. Linearity in the checkpoint interval with respect to the polling threshold allows for accurate prediction of the desired threshold.

3. The overhead for checkpoint interval maintenance due to the compiler-assisted technique. This overhead results from the time measure accumulation and checkpoint decision making at polling points. The time measure accumulation overhead depends upon the frequency of updating the time measure counter, and the polling overhead depends upon the frequency of executing the polling points. If the time measure

counter is updated sparsely or very few polling points are inserted, the checkpoint interval may become inaccurate. On the other hand, a high frequency of checkpoint polling and time measure accumulation leads to a high run-time overhead. Compilation time is ignored in this thesis since it is incurred only once and it is usually very small. Other overhead factors such as checkpoint time, that are not related to checkpoint insertions, are not considered in this chapter. In fact, checkpoint time is more related to the checkpoint size than to the checkpoint interval maintenance [11,47].

4. Code size. This reflects the space overhead due to code insertions.

Of the six benchmark programs examined, four are scientific applications in which loops are large and the calling depth is small. The other two programs contains a number of small loops and a large calling depth. The six benchmark programs are as follows:

convlv: is an FFT algorithm that finds the convolution of 1024 signals with one response [11,47].

espresso: is a SPEC integer program for Boolean function minimization, developed at the University of California at Berkeley [41]. It contains short loops and recursive functions.

li: is a Lisp interpreter solving the 8-queen problem. It is a SPEC integer program developed by Sun Microsystems [41].

ludcmp: is an LU decomposition algorithm that decomposes 100 randomly generated matrices of uniformly distributed size between 50 and 60 [11,47].

rkf: uses the Runge-Kutta-Fehlberg method for solving the ordinary differential equation $y' = x+y$, $y(0) = 2$ with step size 0.25 and error tolerance 5×10^{-7}. This is a floating-point intensive program with large loop bodies [11,47].

rsimp: is the revised Simplex method, for solving the linear optimization problem for the *BRANDY* set, from the Argonne National Laboratory [11,47].

Table 5.2. Benchmark Characteristics.

Program	Static Basic Block		Dynamic Basic Block	
	Total number	Avg. size (ins./block)	Total number (10^6)	Avg. size (ins./block)
convlv	128	5.9	13.5	9.9
rkf	33	4.7	4.3	7.6
ludcmp	96	3.5	20.9	5.0
rsimp	185	3.1	73.0	4.6
espresso	9018	3.1	108.6	2.9
li	3077	2.4	149.3	2.3

Table 5.2 describes the structure of the six programs in terms of the basic blocks. The block size is the number of the RTL instructions in a basic block. The static program information is collected from the program during compilation, while the dynamic information is collected from profiling during execution. The fact that **convlv** and **rkf** have large loop bodies is reflected in their large dynamic basic block sizes. Similarly, **espresso** and **li** have small loop bodies (and thus small dynamic basic blocks). The basic block size has an important impact on the performance overhead required for checkpoint interval maintenance. Smaller basic blocks result in a higher maintenance cost in B-B and B-L as the ratio of the inserted code to the basic block size is high.

5.4.3. Checkpoint intervals

Table 5.3 summarizes the checkpoint intervals generated on a Sun 3/50 diskless workstation. The threshold value, **L**, is the number of RTL instructions that are executed before the next checkpoint for B-B and B-L, and the number of loop iterations for L-L and SL-SL.

For all six programs, the checkpoint interval generated is linearly scalable. The **L** value is program specific due to the different block structures in different programs. For the same L, the floating-point programs (e.g., **rfk.**) generate longer checkpoint intervals than do

Table 5.3. Checkpoint Interval Maintenance (Sun 3).

Program	Scheme	L	Interval Average (sec)			Standard Deviation (sec)		
			L	5L	10L	L	5L	10L
convlv	B-B	500,000	1.5	7.7	15.4	0.0209	0.0355	0.0528
	B-L	500,000	1.4	7.2	14.7	0.0347	0.0768	0.1040
	L-L	50,000	7.8	23.8	47.7	0.0654	0.1168	0.1535
	SL-SL	50,000	4.8	23.7	47.7	0.0960	0.2287	0.3427
rkf	B-B	500,000	5.1	25.3	50.7	0.5713	2.4046	4.3954
	B-L	500,000	5.0	24.7	49.7	0.5684	2.3988	4.3822
	L-L	50,000	8.2	40.7	81.4	1.0644	3.9855	6.9957
	SL-SL	50,000	8.1	39.6	76.8	0.6678	1.4292	4.6366
ludcmp	B-B	500,000	1.4	7.2	14.4	0.1734	0.1552	0.1976
	B-L	500,000	1.3	6.7	13.5	0.1642	0.1556	0.2017
	L-L	50,000	2.0	10.1	20.1	0.1146	0.1287	0.1969
	SL-SL	50,000	1.9	9.5	18.7	0.1825	0.4040	0.3159
rsimp	B-B	500,000	1.3	6.3	12.6	0.0228	0.0820	0.1601
	B-L	500,000	1.2	5.9	12.1	0.0204	0.0651	0.2833
	L-L	500,000	15.3	76.7	154.4	0.2102	1.0541	1.8698
	SL-SL	500,000	15.2	76.0	151.8	0.3683	0.7133	0.2399
espresso	B-B	500,000	0.8	4.2	8.3	0.0043	0.0474	0.1492
	B-L	500,000	0.8	4.0	7.9	0.0218	0.2284	0.6066
	L-L	500,000	4.9	24.0	48.1	1.6159	6.0640	10.7311
	SL-SL	500,000	3.9	17.8	37.3	3.5543	8.5735	13.9470
li	B-B	500,000	1.2	5.9	11.8	0.0002	0.0007	0.0007
	B-L	500,000	0.9	4.6	9.5	0.0031	0.0005	0.5168
	L-L	500,000	7.3	37.0	72.5	0.0187	0.3404	0.0344
	SL-SL	500,000	6.1	29.1	58.3	0.3896	0.1070	0.1564

the integer benchmarks (**espresso** and **li**). The linear scalability of the checkpoint interval makes it possible to produce a consistent checkpoint interval across different programs. For example, the first few polling points can compare the targeted checkpoint interval with those generated under the initial **L**. If they disagree, **L** can be adjusted according to this linearly scalable relationship to obtain the desired checkpoint interval.

The standard deviation in the checkpoint interval reflects the accuracy of the interval as maintained by the instruction-based measure. Table 5.3 compares the standard deviations of all the four schemes. Generally, the standard deviations are less than one third of their corresponding checkpoint intervals. Statistically, the actual checkpoint interval would most likely be within two or three standard deviations of the average interval. As mentioned previously, small changes in checkpoint frequency from the optimal frequency have little effect on the performance of the optimal solution [1–6]. Therefore, this small variation in the generated checkpoint interval will still ensure a near optimal interval. Using the loop iteration count in L-L and SL-SL does not noticeably decrease the checkpoint interval accuracy. This may result from the large threshold **L** value, since the large number of loop iterations between checkpoints likely leads to a stable mixture of instructions for each checkpoint interval. As a comparison, Table 5.4 shows a program-independent checkpoint interval as maintained by the dynamic interrupt scheme described in Section 5.2.

The results for L-L and SL-SL on a Sun 4 SPARC IPC are given in Table 5.5. The checkpoint interval for the programs with many floating-point operations and large loop bodies (**rkf** and **convlv**) is significantly larger than for those with smaller loop bodies. The integer programs, especially **espresso** and **li**, generated comparable intervals. This suggests that **L** is less program specific for integer programs in a RISC machine than in a

Table 5.4. Interrupt Driven Dynamic Scheme (Sun 3).

Program	**Threshold value** (sec)	**Average number of checkpoints**	**Average interval** (sec)	**Standard deviation** (sec)	**Exec. time overhead** (%)
convlv	5	64.6	4.9	0.069	0.2
rkf	5	81.0	5.0	0.048	0.1
ludcmp	5	51.0	4.9	0.107	0.2
rsimp	5	146.2	5.0	0.056	0.1
espresso	5	41.2	4.9	0.089	0.0
li	5	672.5	5.0	0.022	0.2

Table 5.5. Checkpoint Interval Maintenance (Sun 4).

Program	**Scheme**	**L**	**Interval Average (sec)**			**Standard Deviation (sec)**		
			L	5L	10L	L	5L	10L
convlv	L-L	50,000	0.4	2.1	4.2	0.0139	0.0412	0.0709
	SL-SL	50,000	0.4	2.1	4.2	0.0119	0.0258	0.0363
rkf	L-L	50,000	1.1	5.5	10.9	0.2189	0.8943	1.6498
	SL-SL	50,000	1.1	5.4	10.8	0.2323	0.9276	1.6498
ludcmp	L-L	50,000	0.3	1.3	2.5	0.0256	0.0289	0.3707
	SL-SL	50,000	0.2	1.2	2.3	0.0290	0.0515	0.0449
rsimp	L-L	500,000	1.9	9.5	19.0	0.0227	0.0575	0.0918
	SL-SL	500,000	1.8	8.9	17.8	0.0628	0.1975	0.3587
espresso	L-L	500,000	1.1	5.3	10.5	0.2490	0.9836	1.8720
	SL-SL	500,000	0.9	4.0	8.3	0.7575	1.8839	2.9514
li	L-L	500,000	1.9	9.7	19.4	0.0074	0.0090	0.0297
	SL-SL	500,000	1.7	8.3	16.5	0.0182	0.0319	0.0463

CISC machine, as the frequency of almost one instruction-per-cycle improves the accuracy of instruction count or loop count as a measure of execution time. However, the SUN SPARC checkpoint intervals for the integer benchmarks (**espresso** and **li**) are in the same order of magnitude as the floating-point programs with comparable loop sizes, while the SUN 3 checkpoint intervals for the same integer programs are one order of magnitude smaller. The increased checkpoint intervals for **espresso** and **li** on SUN SPARC can be explained by the lack of support for integer multiplication and division on SUN SPARC [53]. In fact, integer multiplication and division are implemented through software traps, and are frequently used for address manipulations in the integer benchmarks we examined. The discrepancies in checkpoint intervals between programs with intensive floating-point operations and those with intensive integer operations still exist for SUN SPARC, since the SPARC IPC implementation supports the floating-point through an off-chip floating-point unit.

5.4.4. Checkpoint interval maintenance overhead

In Table 5.6, the execution overhead in B-B and B-L is generally around 20% for programs with moderate basic block size (**convlv**, **ludcmp**, **rkf** and **rsimp**) and more than doubles the execution time for programs with small basic block size (< 3 for **espresso** and **li**). This is expected since the instruction-based measure is updated in each basic block. A smaller basic block results in a larger updating code with respect to the block size, and thus larger insertion overhead. In B-B, the checkpoint polling point is also inserted in each basic block. The B-B scheme has roughly twice as much overhead as the B-L scheme. The large value for the polling threshold **L** and small block size imply that the polling

Table 5.6. Checkpoint Interval Maintenance Overhead (Sun 3).

Program	Scheme	Execution time		# of RTL insns.		Executable size		Text seg. size	
		(sec)	(%)		(%)	(K)	(%)	(K)	(%)
hline	original	360.3		790		32		16	
	B-B	414.4	15.0	1274	61.3	40	25	24	50
convlv	B-L	388.8	7.9	959	21.4	40	25	24	50
	L-L	367.5	2.0	848	7.3	40	25	24	50
	SL-SL	363.6	0.9	811	2.7	40	25	24	50
rkf	original	416.4		188		24		8	
	B-B	434.7	4.4	331	76.1	24	0	8	0
	B-L	430.6	3.4	235	25.0	24	0	8	0
	L-L	424.4	1.9	202	7.5	24	0	8	0
	SL-SL	417.8	0.3	198	5.6	24	0	8	0
ludcmp	original	245.2		414		24		8	
	B-B	317.4	29.5	809	95.4	32	33	16	100
	B-L	297.1	21.2	560	35.3	32	33	16	100
	L-L	265.5	8.3	477	15.2	24	0	8	0
	SL-SL	245.2	0.0	437	5.6	24	0	8	0
rsimp	original	678.2		724		24		8	
	B-B	843.4	24.4	1488	105.5	32	33	16	100
	B-L	796.7	17.5	1011	39.6	32	33	16	100
	L-L	732.0	7.9	852	17.7	32	33	16	100
	SL-SL	678.5	0.0	764	5.5	24	0	16	100
espresso	original	217.5		35621		176		152	
	B-B	517.7	138.0	71611	101.0	440	150	408	168
	B-L	418.6	92.4	47005	32.0	328	86	296	95
	L-L	312.5	43.7	38708	8.7	208	18	176	16
	SL-SL	218.7	0.5	36340	2.0	184	5	160	5
li	original	3330.2		10459		104		80	
	B-B	8152.0	144.8	22860	118.6	200	92	168	110
	B-L	6481.2	94.6	14736	40.9	160	54	128	60
	L-L	4429.3	33.0	11763	12.5	120	15	88	10
	SL-SL	3343.7	0.4	10595	1.3	104	0	80	0

at each basic block is unnecessary if a fine-grained checkpoint interval is not targeted. If additional hardware is available, an interrupt-driven mechanism can be used to eliminate the high overhead in B-B and B-L. In fact, a hardware instruction (cycle) count register can be added as part of the process context. It can be decremented whenever an instruction is executed. Once it reaches zero, an interrupt for checkpointing can obtain a static checkpoint without any polling overhead.

The execution overhead for L-L is relatively small for programs with large loop sizes. However, L-L may still result in high polling overhead for programs with small loops (**espresso** and **li**). The profile-based SL-SL produces the smallest execution overhead of the four schemes, by polling only at the selected loops. In fact, the overhead is less than one percent of the execution time.

The increase in program size on a SUN 3 due to code insertion is presented in Table 5.6. The executable file size and text segment in the executable file are aligned at an 8K page boundary. Thus, the increases in the executable and text segment sizes may not reflect the true code overhead accurately. The number of RTL instructions in a program may be a better indicator for describing the code size overhead. The space overhead follows the general pattern in the execution time overhead. The L-L scheme typically has a code overhead of 20% on a Sun 3/50, while SL-SL has a mere 5% code size overhead.

Similar results for L-L and SL-SL on a Sun SPARC IPC are given in Table 5.7. The execution overhead is reduced (by almost a half) for integer benchmark programs (**espresso** and **li**) and increased for the floating-point programs for L-L. This reflects that SUN SPARC RISC provides an effective support for most integer operations and the off-chip floating-point operations. The execution time overhead for SL-SL is again less than 1% of the total execution time. The space overhead for L-L on a Sun SPARC IPC is slightly increased due to the relatively large RISC code size compared to the non-RISC code size. The space overhead for SL-SL is less than 4% of program size.

Table 5.7. Checkpoint Interval Maintenance Overhead (Sun 4).

Program	Scheme	Execution time		# of RTL insn.		Executable size		Text seg. size	
		(sec)	(%)		(%)	(K)	(%)	(K)	(%)
convlv	original	28.6		1297		40		24	
	L-L	29.8	4.5	1401	8.0	40	0	24	0
	SL-SL	28.6	0.1	1308	0.9	40	0	24	0
rkf	original	54.5		312		24		8	
	L-L	55.6	2.1	337	8.0	24	0	8	0
	SL-SL	54.8	0.6	323	3.5	24	0	8	0
ludcmp	original	31.1		638		24		8	
	L-L	34.0	9.2	742	16.3	32	33	16	100
	SL-SL	31.3	0.5	649	1.7	32	33	16	100
rsimp	original	83.7		1114		32		16	
	L-L	94.1	12.1	1309	17.5	32	0	16	0
	SL-SL	83.9	0.0	1125	1.0	32	0	16	0
espresso	original	44.7		46810		256		232	
	L-L	56.0	25.1	51572	10.2	304	19	272	16
	SL-SL	44.8	0.1	46821	0.0	272	6	240	5
li	original	939.2		13796		144		112	
	L-L	1087.2	15.8	16137	17.0	168	17	128	14
	SL-SL	944.0	0.5	13807	0.1	152	6	112	0

5.4.5. Profiling and SLFC selection

In our profiling experiments, the minimal coverage that was selected for the SLFC selection algorithm was 90%. Table 5.8 indicates that our algorithm identifies only one loop/function polling point for each of the six programs we considered. Tables 5.3 and 5.6 have shown that this SLFC selection is effective in reducing overhead and producing stable checkpoint intervals. The coverage factor for **espresso** is less than 100%. This is because our algorithm stops after the resulting coverage is greater than the minimal coverage (90%). The 100% coverage is still possible but our algorithm did not continue any further.

The key to a successful profiling is to use a representative data set during profiling. There are four sets of data for **espresso**. We used the first set (**bca.in**) as the profile data.

Table 5.8. SL-SL Profiling Summary.

Program	Loop set	Cover set	Threshold set	Coverage (%)	Analysis time (sec)	
					Sun 3	Sun 4
convlv	{0-14}	{14}	{15}	100	1.9	0.8
rkf	{0-2}	{1}	{7100}	100	0.4	0.1
ludcmp	{0-14}	{4}	{39}	100	3.5	1.5
rsimp	{0-29}	{20}	{10}	100	1.4	0.6
espresso	{0-783}	{621}	{910}	94.2	10.1	4.3
li	{0-388}	{156}	{13}	100	72.8	32.9

Table 5.9. SL-SL Results for Nonprofiled Data Sets.

Data set	Scheme	Sun 3		Sun 4	
		Interval 10L	Exec. time (sec)	Interval 10L	Exec. time (sec)
bca.in	original		217.5		44.7
	SL-SL	37.3	218.7	8.3	44.8
cps.in	original		269.1		57.7
	SL-SL	17.7	269.5	3.8	57.7
ti.in	original		323.9		69.9
	SL-SL	10.1	324.3	2.2	70.0
tial.in	original		554.6		113.9
	SL-SL	26.1	555.5	5.3	114.4

Table 5.9 compares the results for the program profiled on **bca.in** and run with three nonprofiled data sets. The execution overhead for SL-SL is still less than 1%. The checkpoint interval for the profiled data set is 37.3 sec on Sun 3. However, the checkpoint interval ranges from 26.0 to 10.1 sec for the nonprofiled data sets. This indicates that **bca.in** may not be the representative data set for the four data sets, and it highlights the need for representative profiling data in using the profile-based SLFC selection.

5.4.6. Comparison with CATCH

With respect to overhead, the L-L scheme is very close to the basic CATCH [11]. The L-L run-time overhead is essentially the same as that for maintaining the potential

Table 5.10. Run-Time Overhead (%) Comparison: Static vs. Dynamic Schemes.

Program	**Static Insertion**		**Dynamic Insertion**		
	L-L	**SL-SL**	**CATCH**		**Interrupt Driven**
			Basic	**Trained**	
convlv	2.0	0.9	4.8	1.4	0.2
rkf	1.9	0.3	2.7	0.8	0.1
ludcmp	6.8	0.0	8.2	3.8	0.2
rsimp	7.9	0.0	13.2	5.2	0.1
espresso	43.7	0.5	54.7	9.9	0.0
li	33.0	0.4	34.8	6.1	0.2

checkpoint leverage in CATCH. The extra overhead for CATCH is in polling the real-time clock. The results for SL-SL are comparable to those for the trained CATCH, as both use the profile-based approach. In the trained CATCH, the cover set is selected based on coverage and checkpoint size with no regard to the threshold value determination and non-overlapping of cover ranges. Table 5.10 compares L-L and SL-SL with their corresponding CATCH schemes. The interrupt-driven dynamic scheme is also presented. Generally, the overhead for our static schemes (L-L and SL-SL) is less than that for the dynamic CATCH. The overhead for SL-SL is comparable to that for the interrupt-driven dynamic approach, without using extra hardware support.

5.5. Summary

In this chapter, a compiler-assisted approach for static checkpoint insertion has been presented. This approach uses an instruction-based measure to describe checkpoint intervals in terms of computation progress. The instruction-based measure is independent of the real-time clock, although it has a time attribute related to the program execution. This relationship between computation progress and execution time makes it possible to use an instruction-based measure for checkpoint interval maintenance.

Four different schemes, based on this approach, have been implemented and evaluated. Experiments show that our static method can generate a stable and scalable checkpoint interval. The overhead for the basic block-based schemes, such as B-B and B-L, is very high without hardware support. The loop iteration count scheme (L-L) can obtain a comparable checkpoint interval as B-B and B-L, with a reasonable overhead. The block size of a program has a significant impact on insertion overhead for our schemes. The profile-based SL-SL scheme can effectively reduce both the run-time overhead as well as the space overhead. In fact, this scheme can produce scalable and stable checkpoint intervals with an overhead comparable to that for the hardware interrupt scheme. This requires only a representative data set for accurate prediction of program run-time behavior. Overall, our experiments recommend the loop iteration count schemes (L-L and SL-SL) as reasonable choices for static checkpoint insertion. Both have a smaller run-time overhead than the corresponding CATCH schemes.

CHAPTER 6.

EVOLUTIONARY CONCURRENT CHECKPOINTING

6.1. Introduction

6.1.1. Inconsistent recovery line and rollback propagation

In parallel and distributed computations, there are many concurrent processes that communicate with each other. A recovery line in this case is a set of local checkpoints, one for each process. If a process prior to its local checkpoint communicates with a process that already took its local checkpoint, this communication is said to cross the corresponding recovery line. Communication across a recovery line leads to rollback propagations [7, 12, 13]. In Figure 6.1, message M1 crosses the recovery line, {C12, C13}, and M2 crosses the recovery line {C13, C23}. Process 1 rolls back to C13 when an error is encountered. However, process 2 has to roll back to C22 since process 1 after its rollback needs m2 from process 2. The phenomenon that rollback to a recovery line causes the system to roll back to another recovery line is called *rollback propagation*. In its extreme, rollback propagation can force the system to roll back in a domino fashion [7, 12, 13]. In this thesis, a recovery line is said to be consistent if there is no communication across the recovery line. Therefore, a consistent recovery line can eliminate rollback propagation by guaranteeing that there is no communication across the recovery line.

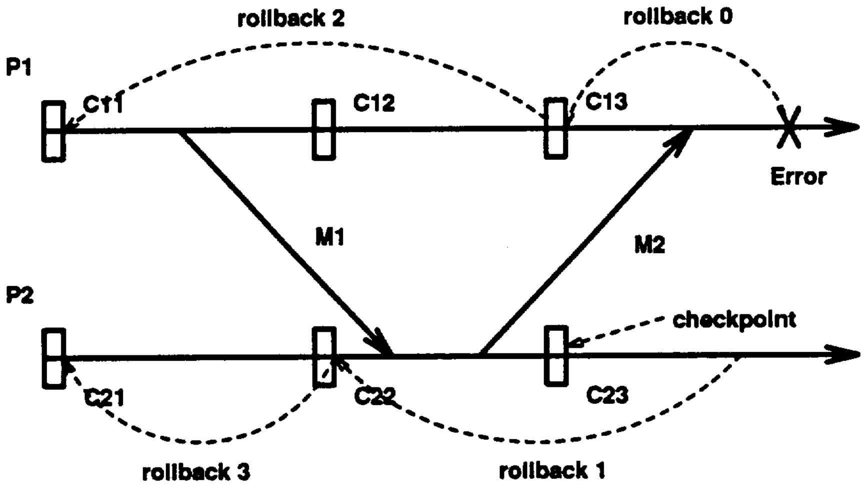

Figure 6.1. Rollback Propagation.

6.1.2. Previous approaches

Independent checkpointing allows individual processes to take their local checkpoint without any coordination between processes [54]. This approach cannot guarantee a consistent recovery line and often requires keeping multiple recovery lines. Rollback propagations are allowed. In contrast, coordinated checkpointing tries to obtain a consistent recovery line and eliminate rollback propagation.

One approach to obtaining a consistent recovery line is to stop computation and synchronize the concurrent processes at an agreed upon point in time [13,14,17]. In some tightly coupled systems it is possible to synchronize processors instantaneously [17]. However, typically this global synchronization requires rounds of message exchanges. An alternative is to synchronize checkpointing with communication [15,17,19,55]. Whenever two processes communicate, checkpointing can be invoked in both processes. The recovery line is always consistent since there is no communication across the corresponding checkpoints.

During recovery, only the individual process encountering the error rolls back, because the faulty process has not communicated since its last checkpoint. In the communication-synchronized approach, checkpointing frequency is fixed and is dependent on communication patterns.

Message logging is often used to reduce the cost of checkpoint operations [19–21]. Instead of resending past messages during recovery, message logs are replayed to produce them. Optimistic logging [22–26], which can be viewed as the communication-synchronized checkpointing with deferred logging operations, is often used. Deferred logging often requires complex methods to keep message dependence information for uncommitted message logs and to manage interleaving message retries and message replays during recovery.

Chandy and Lamport have shown that the global state of a distributed system consists of both states of individual processes and communication channels [56]. They proposed to save individual process states locally by checkpointing and to save the channel states by logging the messages sent before the sending processes save their process states. A special marker message is broadcast to all other processes after the local process makes its checkpoint. Provided a FIFO channel, all messages before the marker message from the process are the ones that need to be logged. The number of broadcast messages required is N. This approach has been applied to concurrent checkpointing in distributed systems [57–59]. With a bounded communication latency and loosely synchronized clocks, the special marker messages can be eliminated [60, 61]. However, restoring the original message order after rollback often requires a mechanism to determine when to replay from message logs and when to retry messages due to the interleaving of logged messages and normal messages during checkpointing.

This chapter describes an evolutionary approach to concurrent checkpointing. In this approach computation periodically enters a checkpoint session, where a consistent recovery line evolves. A checkpoint session can be initiated at any computation point. Upon receiving notification of the start of a checkpoint session, each process independently takes a local checkpoint. The initial recovery line, made up of the local checkpoints, may be inconsistent since no attempt has been made to prevent communication across it. As computation progresses, the local checkpoints are updated whenever there is a communication between processes, as in the communication-synchronized approach. This local checkpoint updating causes the recovery line to evolve into a consistent recovery line. At the end of the checkpoint session a consistent recovery line is guaranteed and its checkpoints can be committed. The resulting global recovery line requires that all processes roll back to their previous checkpoint if an error occurs. The frequency of checkpoint sessions can be controlled, depending on the performance and reliability requirements of the system.

Our approach does not specify the mechanism by which individual checkpoints are taken. It attempts to reduce the overhead in coming to an agreement about a consistent recovery line. Therefore, it is useful only in systems where the overhead of synchronization between processors dominates the overhead of taking individual checkpoints. Other limitations to our approach are the requirements that communication is synchronized between processors and that communication latency is bounded. Many systems conform to these requirements, and ones that do not can usually be modified to conform.

The following section describes the assumptions, key ideas, and key techniques of our evolutionary checkpointing algorithm. The subsequent two sections discuss the correctness and performance considerations of our algorithm. Section 6.5 describes the application of our algorithm to rollback error recovery in both shared-memory and distributed memory computer systems.

6.2. Evolutionary Checkpointing

6.2.1. Computation model

The computation considered in this chapter consists of a number of concurrent processes that communicate through messages over a network. This model is extended later to a cache-based shared-memory system by viewing a memory access to nonlocal data as a message from the source processor that provides the data to the destination processor that receives the data.

In our communication model, messages are assumed to be synchronized: the sender is blocked until an acknowledge message is received from the receiver. Most lower layers of network models naturally provide and implement acknowledge mechanisms (e.g., Ethernet). Reliable communication requires acknowledge messages even at high levels. In distributed-memory systems and network file servers, the read/write requests are in fact implemented with remote procedure calls (*RPC*) or synchronized messages [62]. Multiprocessor systems also meet this assumption since read/write accesses are atomic and synchronized. The assumption provides two advantages. First, checkpointing of a message sender can be requested by the message receiver during a checkpoint session if necessary. Second, this checkpointing request can be piggybacked on the acknowledgment at low additional cost.

Christian, and Tong et al. use a bounded communication latency to remove the special checkpointing marker messages in Chandy's checkpointing scheme [56,60,61]. In this thesis, we use a similar bounded communication latency for our evolutionary checkpoint scheme to determine a consistent recovery line without exchanging extra messages. We denote the communication upper bound as Δ in this chapter.

In general, communication latency is nondeterministic at the user level due to message size, processes that are not ready to communicate, and underlying network characteristics. A two-layer approach can be used to achieve a bounded communication latency. A message server can be inserted below the user-level process. The user process sends and receives messages only through its message server. The user-level messages can be asynchronous and unbounded in communication latencies. However, the message server divides messages into packets to remove the uncertainty in communication latency due to message size. In many networks, proper techniques such as priority preemptive scheduling can guarantee a deterministic communication response for the message server [63,64]. In some systems, the message server is a natural component such as the cache controller in shared-memory multiprocessors, and the pager in distributed-memory systems [15,65]. Another approach that can be used to obtain a bounded communication latency is the time-out mechanism. Even if the communication latency is unbounded, messages are delivered within a small threshold with a high probability [61]. The messages with a communication delay larger than the time-out threshold can be detected and treated as a *performance failure* [61].

A computation is divided into alternating checkpoint-free and checkpoint sessions. A checkpoint is a snapshot of the process state at the time of checkpointing. The operation of our scheme does not depend on the manner in which the checkpoints are taken, as long as the computation state at the checkpoint can be restored. Since in a checkpoint session only the last checkpoints taken on each processor are guaranteed to form a consistent recovery line, the intermediate checkpoints can be generated in local memory and do not have to written out to a stable or backup storage. Thus, checkpoint updating can be accomplished quickly by marking the process state unmutable [15,17,45]. The final checkpoints still need to be copied to stable storage. If the overhead of waiting for this copying to occur is too high, another process can be scheduled to do the copy, without blocking the computation [45]. We therefore assume the checkpoint operation time during a checkpoint session to be negligible compared to the communication delay upper bound.

A checkpointing coordinator broadcasts a *ckp_start* message to initiate a checkpoint session and a *ckp_end* message to terminate this checkpoint session. This checkpointing coordinator can be one of the participating concurrent processes. Our recovery algorithm can handle errors that cause missing ckp_start or ckp_end messages. The need to broadcast the ckp_end messages can be eliminated by a local timer at each process. If the local clocks are loosely synchronized with a small shift, using local clocks to signal ckp_start and ckp_end is possible, similar to other schemes in the literature [60,61].

The point of time at which a process enters a checkpoint session is its entry point to the checkpoint session. Similarly, the time at which a process exits a checkpoint session is its exit point to the checkpoint session. The set of the entry points for a checkpoint session form the checkpoint session entry line and the set of exit points for a checkpoint session form the checkpoint session exit line. The reception points of ckp_start and ckp_end form the initial entry line and exit line for the checkpoint session.

6.2.2. Approach

In our approach, rollback propagation is eliminated by obtaining a consistent recovery line every time a checkpoint is taken. To obtain a consistent recovery line our approach guarantees the following conditions:

- There is at least one local checkpoint for each process, and thus a recovery line, during a checkpoint session.
- This recovery line converges to a consistent one as the computation progresses.
- There are no messages exchanged across the entry line or exit line. Inside the checkpoint session, messages do not cross the current potentially consistent recovery line.

To fulfill these requirements, our checkpointing schemes makes use of the following techniques:

- Upon entering a checkpoint session, every process immediately takes a checkpoint. This guarantees that there always exists a recovery line from the beginning of the session.
- To eliminate messages crossing the checkpoint session entry line, the initial entry points are adjusted to *include* crossing messages in the checkpoint session.
- To remove messages crossing the exit line, the initial exit points are adjusted to *exclude* crossing messages from the checkpoint session.
- Inside the checkpoint session, checkpointing is synchronized with communication. Both the receiver and sender of a message take a new local checkpoint immediately after the communication. This communication synchronized checkpoint updating leaves

```
-------------------------------------------------------------------
local variables and operation for each node:

    ckp_num:       checkpoint number (time stamp);
    ckp_session:   checkpointing in session flag
                      0 - not in a checkpointing session
                      >0 - ckp_num currently in session
    checkpoint(n): make a local checkpoint with checkpoint number n
    enter_ckp_session()  // enter a checkpoint session
        {    ckp_num++; ckp_session = ckp_num;
             checkpoint(ckp_num);
        }

Augmented message format:

    message : <ckp_num, ckp_session, normal message>;
    ack:       acknowledge: <ckp_num, normal acknowledge>
                   ckp_num - 0 : no need to checkpoint
                            >0: makes a checkpoint with
                                checkpoint number ckp_num.

-------------------------------------------------------------------
```

Figure 6.2. Local Variables and Operations at Each Process Node.

the exchanged message behind the recovery line and makes the recovery line evolve towards a consistent line.

- To avoid messages completely bypassing the checkpoint session, ckp_end is signalled 2 Δ after ckp_start.

6.2.3. Detailed description

Figure 6.2 describes the local data structures needed to implement evolutionary checkpointing [1]. Figures 6.3 and 6.4 describe the detailed algorithms for the message sender and receiver, respectively, for our checkpointing scheme.

[1] The appended checkpointing information such as the checkpoint number can be eliminated if the delivery of ckp_start and ckp_end is reliable, since using mismatches in checkpoint numbers to detect the missing ckp_start and ckp_end messages is not necessary.

```
ack = send_message(msg);
if (ack.ckp_num > 0) { // need to make a local checkpoint
    if (ckp_num + 1 == ack.ckp_num) {
        // receiver already passed the entry line
        // advance local checkpoint entry point to now
        enter_ckp_session();
    } else if (ckp_num == ack.ckp_num) {
        // both sender and receiver in checkpointing session
        checkpoint(ckp_num);
    } else { // detect performance fault or missing ckp_start
             // or ckp_end msgs (Lemma 3).
        error();
    }
} else if (ack.ckp_num == 0) { // no need for local checkpointing
    if (ckp_session != 0) {
        // receiver already exits its session
        // adjust its checkpoint exit point to now.
        ckp_session = 0;
    }
} else  // impossible by the ack format
    error();
```

Figure 6.3. Sender Algorithm.

```
When a ckp_start is received,

    if (ckp_start.ckp_num == ckp_num+1 && ckp_session == 0)
        enter_ckp_session();  // a new ckp_session
    else if (ckp_start.ckp_num == ckp_num && ckp_session == 1)
        // ignore it; its entry point has been adjusted before.
    else error(); // detect missing ckp_start/end msgs.

When a ckp_end is received,

    if (ckp_end.ckp_num == ckp_num) {
        if (ckp_session == 1) ckp_session = 0; // exit the session
        // else ignore it; its exit point has been adjusted before.
    } else error(); // detect missing ckp_start/end msgs.

when a message is received,

    if (ckp_session) { // checkpointing in session
        if (msg.ckp_num + 1 == ckp_num) {
            // sender yet to enter checkpointing session;
            ack_back(ckp_num); // ask sender to checkpoint
            checkpoint(ckp_num);
        } else if (msg.ckp_num == ckp_num) {
            // both sender and receiver in the ckp session;
            if (msg.ckp_session == ckp_num) {
                // sender still in checkpointing session;
                // both update their local checkpoints.
                ack_back(ckp_num);
                checkpoint(ckp_num);
            } else {
                // sender exited the session; no checkpoint update.
                ack_back(0);
            }
        } else // detects missing ckp_start/end msgs
            error();
    } else { // out of the checkpointing session
        if (ckp_num == msg.ckp_num) {
            // both sender and receiver out of the session;
            // no local checkpointing asked for the sender.
            ack_back(0);
        } else if (ckp_num+1 == msg.ckp_num) {
            // receiver yet to enter the checkpointing session;
            // advance the local entry point to now.
            ckp_num++;
            ckp_session = ckp_num;
            ack_back(ckp_num);
            checkpoint(ckp_num);
        } else // performance fault or ckp_start/end missing:
               // msg crosses the ckp session (lemma 3).
            error();
    }
```

Figure 6.4. Receiver Algorithm.

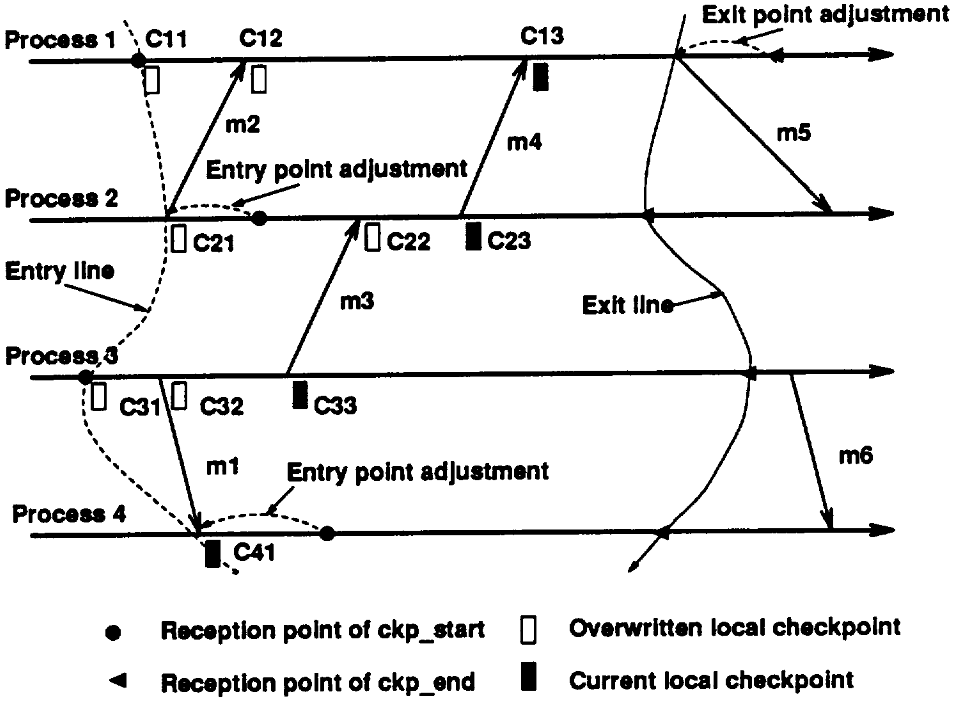

Figure 6.5. Checkpoint Session and Recovery Lines.

6.2.3.1. Entering a checkpoint session

Upon receiving the ckp_start signal from the checkpointing coordinator, a process enters the checkpoint session and takes a local checkpoint by saving its process state. The different session entry points of the processes form the session entry line. The initial set of local checkpoints provides a potentially inconsistent initial recovery line. For example, the initial recovery line {C11, C21, C31, C41} in Figure 6.5 is not consistent since if process 2 is restarted from C21, it will not resend message m2, while if process 1 is restarted from C11, it will wait for this message.

6.2.3.2. Adjusting the entry points

If a process that has not entered the checkpoint session exchanges a message with a process already in the checkpoint session, it will not wait for the ckp_start to enter the checkpoint session. Instead it marks its entry point as if it has received the ckp_start before the message exchange and takes its initial checkpoint right after the exchange. When the ckp_start is subsequently received, it is ignored. The adjustment of an entry point from the ckp_start reception point is demonstrated in Figure 6.5 where message m1 crosses the original entry line (the ckp_start reception line). By moving the entry point of process 4 to the point of communication, m1 is included in the checkpoint session. Therefore, process 4 makes its initial checkpoint C41 while process 3 updates its local checkpoint C31 with C32 at the request piggybacked on the acknowledge from process 4.

6.2.3.3. Updating local checkpoints

If a message is exchanged between two processes inside a checkpoint session, the receiver updates its local checkpoint to the current state. Meanwhile, it also piggybacks a request to the sender to update its local checkpoint on the acknowledge of the message. In Figure 6.5, the message m3 between processes 2 and 3 leads to the updating of C21 and C32, to C22 and C33, respectively. This checkpoint updating makes the recovery line evolve to a consistent one by including the exchanged message in the checkpointed state. For example, when process 2 updates C21 with C22 and process 3 updates C32 with C33, they include m3 in the checkpointed state. The new recovery line {C12, C22, C33, C41} is consistent.

In our scheme, a sender takes a local checkpoint only when the acknowledge from the receiver requires it to. During a checkpoint session, local checkpointing is synchronized with the computation, as in the communication-synchronized schemes [15–18]. Our approach can be viewed as a scheme that samples, during checkpoint sessions, a small fraction of checkpoints made by the communication-synchronized schemes.

6.2.3.4. Adjusting the exit points

When a process is in a checkpoint session and receives a ckp_end, it exits the checkpoint session. If the process exchanges a message with a process that has exited the checkpoint session, the process marks its exit point immediately and ignores the subsequently received ckp_end. No checkpoint updating is performed. In this manner, the message exchange is *excluded* from the checkpoint session. It can be proven that when the processes reach the exit line, the current local checkpoints form a consistent recovery line. Message m5 in Figure 6.5 illustrates a case of exit point adjustment. Message m5 crosses the original exit line (the ckp_end reception line). When process 1 finds out that the receiver of m5 is already outside the checkpoint session, it immediately moves its exit point to the point of communication and exits the checkpoint session. In this manner, m5 is excluded from the checkpoint session. When the last process leaves the checkpoint session, the exit line is complete, and the set of current local checkpoints ({C13, C23, C33, C41} in the example) comprises a consistent recovery line.

6.2.3.5. Avoiding bypassing messages

Provided that communication delay is bounded by Δ, broadcasting ckp_end 2Δ after the ckp_start broadcast guarantees that no messages bypass the checkpoint session. That is, there is no message that originates before a checkpoint session and is received after the checkpoint session, such as message m4 in Figure 6.6. If a message were allowed to bypass

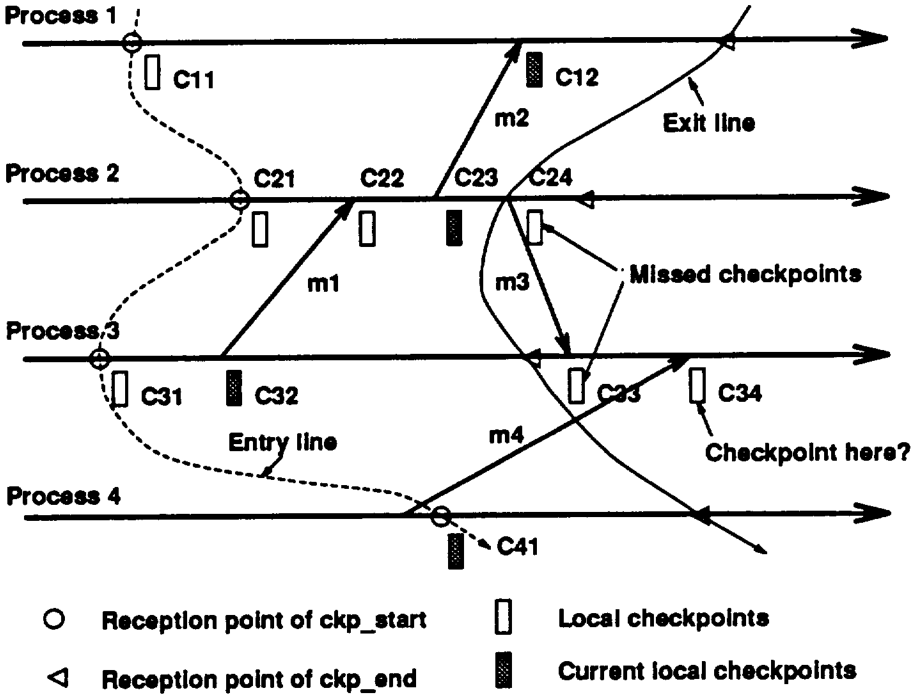

Figure 6.6. Example of a Message Bypassing a Checkpoint Session.

the checkpoint session, some checkpoints of the resulting recovery line might be missing. For example, process 3 interacts with process 2 after passing its exit point but before receiving m4 from process 4. Process 3 has already exited the checkpoint session, thus the exit point of process 2 is adjusted and the local checkpoints are not updated to C24 and C33. Even if we let process 3 update its local checkpoint after receiving m4, the missing checkpoints (C24, C33) make the current recovery line ({C12, C23, C34, C41}) inconsistent, since there is a message exchange across the exit line (m3).

6.2.3.6. Handling Missing Checkpointing Messages

Missing ckp_start and ckp_end messages can be detected by the evolutionary algorithms in Figures 6.3 and 6.4. Suppose process j missed a ckp_start message for a checkpoint session. If it communicates with another process already in the checkpoint session, this missing ckp_start does not affect the checkpointing algorithm, since this message is ignored due to the entry point adjustment. If process j receives a ckp_end message, its local checkpoint number is mismatched with the checkpoint number in the ckp_end message. This detects a missing ckp_start. In general, mismatches in the local checkpoint number and the checkpoint number in messages detects errors in message delivery in the evolutionary scheme.

6.3. Correctness

There exist an entry line and an exit line for each checkpoint session. Every process receives a ckp_start message for each checkpoint session. The only time the entry point is not the reception point of the ckp_start is when the entry point has been adjusted to an earlier point due to a message exchange between a process that is yet to enter the checkpoint session with another process already in the checkpoint session. Thus there is always an entry line at or before the ckp_start reception line. Similarly, there is always a session exit line at or before the ckp_end reception line. Since the ckp_end is broadcast 2Δ after the ckp_start, the ckp_end will be received by each process after the ckp_start. If the ckp_end reception point is the exit point for a process, the exit point is behind its corresponding ckp_start reception point and thus its entry point. If the exit point has been adjusted, the process must be in the checkpoint session when the adjustment occurs, and the exit point will not be adjusted ahead its entry point. Thus, the exit line is always behind the corresponding entry line. Therefore,

Lemma 1: Given the algorithm in Figures 6.3 and 6.4, there is an entry line followed by an exit line for each checkpoint session.

We will show that there is a recovery line after an entry line. That is, every process will have a local checkpoint after this line. Upon receiving a ckp_start, a process either makes a local checkpoint or ignores this ckp_start. According to the algorithm, the process ignores a ckp_start only when its entry point has been adjusted to an earlier time than the ckp_start reception point. As a part of its entry point adjustment, a local checkpoint is made for this process. This proves the following lemma.

Lemma 2: Given the algorithm in Figures 6.3 and 6.4, there is a recovery line after the last process passes the entry line of a checkpoint session.

Lemma 1 and Lemma 2 imply that there is a recovery line when the last process passes the exit line. Before we show that this recovery line is consistent, we first prove a lemma which assures that the minimum time difference between the entry point for one process and the exit point for any other process is at least one Δ. This condition assures that no messages bypass the checkpoint session. That is, a message originated before a checkpoint session will not be received after the checkpoint session and vice versa. Let (s_i, e_j) be the pair of the entry time for process i and the exit time for process j for the same checkpoint session.

Lemma 3: Given the algorithm in Figures 6.3 and 6.4, $e_j - s_i > \Delta$ for any (s_i, e_j) and $i \neq j$.

Proof: Let (s_i, e_j) be the pair with the minimum difference among all the possible pairs. There are only two cases possible. (1) e_j is the time of ckp_end reception by process j. According to Lemma 1, s_i is either the reception time of the ckp_start at process i or an earlier point than the reception time due to an entry point adjustment. Therefore, we need only to prove that the time difference between e_j and the reception time of the ckp_start at process i is greater than Δ. Since any message will be delivered within Δ and the ckp_end is broadcast 2Δ after the ckp_start, no process will receive a ckp_start later than one Δ after the broadcast of ckp_start, and no process will receive a ckp_end before 2Δ after the broadcast of ckp_start. Therefore, $e_j - s_i > \Delta$. (2) e_j is not the time of ckp_end reception by process j. This case occurs only when process j receives a message from a process (e.g., process k) that passed the exit line when process j was still in the checkpoint session (i.e., it is yet to receive the ckp_end). According to the algorithm, j will adjust its exit point from its ckp_end reception point. This case is impossible. Otherwise, e_j cannot be in the minimum pair of (s_i, e_j), since process k has passed the exit line before process j $(e_k < e_j)$. This contradicts that (s_i, e_j) is the smallest pair of all the possible pairs that include (s_i, e_k).

□

A recovery line is consistent if all messages sent before (after) a consistent recovery line are received before (after) this line. That is, there are no message exchanges across a consistent recovery line. This guarantees that any rollback will not need to cross this line and thus eliminates the domino effect of rollback propagation.

Theorem 1: Given the algorithms in Figures 6.3 and 6.4, the set of the current local checkpoints forms a consistent recovery line when the last process exits a checkpoint session.

Proof: According to Lemmas 1 and 2, there is a recovery line when the last process exits a checkpoint session. Suppose a process receives a message after this exit line. The sender cannot be in the state prior to the checkpoint session, since the sender has yet to pass its entry point and thus its exit point. This implies that the exit line is still incomplete. The sender cannot be in the checkpoint session either; otherwise, the algorithm requires the receiver to ask the sender to adjust its exit point to exclude the message exchange from this checkpoint session. Therefore, the sender must be after its exit point. Suppose a process sends a message after the exit line. The receiver cannot be in the state prior to the checkpoint session; otherwise, this gives an incomplete exit line. The receiver cannot be in the checkpoint session either, since the algorithm will adjust the receiver's exit point to exclude the message from this checkpoint session. Thus, the receiver must have passed the exit line. Therefore, there is no message exchange across the exit line, and the recovery line after the exit line is consistent. Since there is no local checkpoint updating after the exit line, this consistent line remains until the next checkpoint session. □

6.4. Performance Considerations

6.4.1. Convergence time

We define the convergence time of our evolutionary checkpointing scheme as the time for a potentially inconsistent recovery line to evolve into a consistent recovery line. This parameter determines the minimum length of a checkpointing session. More importantly, it also affects the overhead involved in our scheme since the longer the convergence time, the more local checkpoint updating is likely. The following theorem gives an upper bound on the convergence time of our algorithm.

Theorem 2: Given the algorithm in Figures 6.3 and 6.4, the convergence time of the recovery line during a checkpointing session is less than $3\ \Delta$.

Proof: According to the algorithm, the first process enters the checkpointing session upon receiving a ckp_start, which occurs no earlier than the ckp_start broadcasting time. The last process to receive a ckp_end will receive it no later than $3\ \Delta$ after the ckp_start broadcast since ckp_end is broadcast 2 Δ after ckp_start, and ckp_end will be delivered to every process within Δ. According to the proof of Lemma 1, the exit line forms before the ckp_end reception line because the exit point is either the reception point of a ckp_end or at an earlier time than the reception point due to the exit point adjustment. Theorem 1 guarantees a consistent recovery line after all processes pass the exit line. Therefore, there is a consistent recovery line no later than 3 Δ after the ckp_start is broadcast. Thus the convergence time is less than 3 Δ. □

6.4.2. Run-time overhead

The expected run-time overhead (C_k) can be simply expressed in terms of the frequency of checkpoint sessions (n), checkpointing time per session (C_s), rollback probability (p_r) and recovery overhead (C_r) as

$$C_k = nC_s + np_rC_r$$

$$C_s = C_{init} + N_{update}C_{update}$$

where C_{init} is the checkpoint cost of the initial checkpoint made at the entry of a checkpoint session; N_{update} and C_{update} are the frequency and the overhead of local checkpoint updating respectively. The first term, nC_s, in C_k represents the checkpointing overhead, while the second term, np_rC_r, is the recovery overhead.

Given the frequency (n) and length (convergence time) of checkpoint sessions, the checkpointing overhead, C_s, depends on the frequency and overhead of local checkpoint updating. The number of times that a local checkpoint is updated is computation specific. Every time a message is sent or received inside a checkpoint session, the local checkpoint has to be updated. Given the limited convergence time, the number of updates is likely to be limited.

To determine the number of checkpoint updates that can be expected in a distributed memory system we traced the communication patterns of eight parallel programs on an 8-node Intel IPSC/2 hypercube (Table 6.1). We took random snapshots of the computation with lengths varying from 10 to 500 msec. On the IPSC/2, message latency averages about 1 msec/K [66]. Our snapshot lengths therefore represent conservative estimates of the session lengths that could be chosen for the IPSC/2. For every program and snapshot length we performed 1000 random trials.

Table 6.2 shows the frequency of messages (which corresponds to the number of checkpoint updates) for different session lengths. For the numerical programs (**fft**, **mult**, **gauss**, **qr** and **navier**), the average number of messages transmitted or received is less than 3. However, the number of messages in a particular checkpoint session can be as high as 244 (**qr**). Typically messages in the hypercube occur in bursts when data are distributed to and collected from the nodes. If this is the case, compiler-assisted techniques that detect

Table 6.1. Hypercube Program Traces.

Program	**Description**	**Execution**	**Message**		
		Time	# recvs.	# sends	avg. size
		(msec)			(bytes)
fft	fast Fourier transform	51363	110	60	97.7K
mult	matrix multiplication	4160	48	43	18.5K
gauss	Gauss elimination	47222	6764	2706	622.8
qr	QR factorization	3590	4105	4098	508.9
navier	fluid flow simulator	21315	118	118	22.7
tester	circuit test generator	123339	13215	10786	264.3
cell	circuit cell placement	50645	42619	42764	31.7
router	VLSI channel router	435648	371700	371650	18.9

Table 6.2. Communication Characteristics of Hypercube Traces.

	Session	Messages			Session	Messages	
Trace	**length**	max	average	**Trace**	**length**	max	average
	(msec)	#	#		(msec)	#	#
fft	10	1	0.12	**navier**	10	4	0.11
	50	2	0.12		50	4	0.11
	100	2	0.12		100	5	0.12
	500	3	0.13		500	10	0.13
mult	10	4	0.05	**tester**	10	15	0.34
	50	4	0.06		50	44	0.98
	100	4	0.06		100	67	1.57
	500	4	0.07		500	100	4.16
gauss	10	10	0.33	**cell**	10	32	2.05
	50	40	0.81		50	104	7.76
	100	80	1.05		100	149	13.48
	500	243	2.24		500	451	28.21
qr	10	7	0.90	**router**	10	30	2.56
	50	30	1.92		50	123	10.58
	100	58	1.85		100	213	21.10
	500	244	2.34		500	505	100.55

communication bursts in programs and plan checkpoint session accordingly could be used to decrease the number of checkpoints in a session [11,67].

The overhead of updating a local checkpoint varies with the checkpointing mechanism used. If a new complete state is saved as the checkpoint update, C_{update} is the same as the initial checkpoint cost, C_{init} [11,47]. If only the change in state since the last checkpoint is needed to update the checkpoint (e.g., flushing dirty pages in a virtual memory system), C_{update} is likely to be smaller than C_{init} [15]. If local checkpoint updating is implemented with logging messages, C_{update} is the cost of message logging. In the above hypercube example, message logging may be appropriate for high message density programs such as **router**.

Recovery overhead, C_r, is related to the reprocessing time after recovery, which on the average is one-half of the checkpoint interval. Studies on checkpoint placement have shown that the rollback probability, p_r, is typically small enough to ensure low recovery overhead (np_rC_r) compared to checkpointing overhead (nC_s), even when the checkpoint interval and/or recovery cost are large. Schemes with an inherent high checkpoint frequency fail to take advantage of the benefits of making checkpoint intervals large. In the evolutionary approach, the checkpoint interval can be chosen as large as necessary to reduce checkpointing overhead [3–6].

6.4.3. Memory overhead

The storage requirement for the evolutionary checkpointing is two global checkpoints, one for the last committed checkpoint and one for the current working buffer for the uncommitted checkpoint. For virtual memory-based systems, the working buffer can be set copy-on-write to the committed checkpoint. The working space is split with the committed checkpoint only when a modification is needed. After the current checkpoint is committed, the space for the old committed checkpoint can be switched to the working space.

6.5. Applications to Shared-Memory Systems

Recently there has been an active research interest in recoverable shared-memory and shared virtual memory computer systems [15–18,68–70]. Both globally synchronized and communication-synchronized approaches have been applied to these systems. The main drawback of these schemes is uncontrollable checkpointing [18]. In this section we will demonstrate how evolutionary checkpointing can be adopted to these situations.

6.5.1. Recovery in cache-based multiprocessor systems

In cache-based systems, cache-based rollback error recovery can be used to recover from transient processor errors [71]. In this recovery scheme, the checkpoint state is kept in the main memory, those dirty cache blocks that have not been modified since the last checkpoint, and the processor registers. A processor takes a checkpoint whenever it is necessary to replace a dirty block in its cache. At a checkpoint, the processor registers are saved, and all dirty cache blocks are marked unchangeable. Unchangeable lines may be read, but have to be written back to memory before being written. Rollback is accomplished by simply invalidating all cache lines except the unchangeable lines, restoring the processor registers, and restarting the computation.

Wu et al. proposed a cache-based recovery method for shared-memory multiprocessor systems using the communication-synchronized approach [15]. A communication is an access to a dirty cache block from the private cache of another processor. Communication between processors induces a checkpoint on the source processor. The destination processor does not need to be checkpointed, since if it rolls back it can always acquire a new copy of the transmitted data from the source processor. The effect is similar to message logging, in that the data received are available again after an eventual rollback. Ahmed et al. have proposed a globally synchronized checkpointing strategy for cache-based error recovery in multiprocessors [17]. They assume that a checkpoint operation can be synchronized among all processors and takes only one cycle.

These cache-based schemes have the disadvantage that the frequency of unavoidable checkpoints, due to replacement of dirty lines, is high [18]. However, the overhead in taking a checkpoint is very low. Therefore cache-based recovery is applicable to updating the checkpoints during the checkpoint session in our evolutionary scheme in which checkpointing activities are for only a very short period.

To apply our approach to cache-based recovery, we first map our system model to the shared-memory multiprocessor model. The cache controllers serve as the message servers of our model. Caches behave as the normal caches for checkpoint free computations, and as Wu's caches during checkpoint sessions. A communication is a *read* or a *write* access to a nonlocal cache. Communication in multiprocessors is synchronized since the processor is blocked until data are accessed. The memory access time, and therefore the communication time, is also bounded.

A global interrupt can be used as the mechanism to generate the ckp_start and ckp_end signals [2]. This global interrupt sets or clears the local flag ckp_session at each processor as if a ckp_start or ckp_end is broadcast. During checkpoint sessions, the checkpoint operation is synchronized with communication such as in Wu's scheme [3]. The checkpoint session can be short since the convergence time is only 3 times the maximum access time to a block present in another processor's cache. At the end of the session, the checkpoint is committed, and the cache is switched from checkpointing operation to normal operation.

A shadow paged memory is needed because the state changes between checkpoint sessions can not overwrite the committed checkpoint [16]. A copy of the memory space is

[2] Since the global interrupts can usually be assumed to be delivered reliably and no error detection for mismatching checkpoint session numbers is necessary, the extra checkpointing information appended to each message required by the evolutionary algorithms (Figures 6.2, 6.3 and 6.4) can be eliminated.

[3] In a remote memory access, a source processor that provides data and a destination processor that initiates the access request can be distinguished. The checkpoint operation at the destination processor can be eliminated since the source processor backs up the data requested in its local checkpoint and the destination processor can retry the access and acquire the data from the checkpoint.

used for the committed checkpoint and another for the temporary working spacing. The unchangeable cache blocks are written back to the checkpoint pages when they are replaced from the caches. A copy-on-write mapping of the working pages to the checkpoint pages may save memory and avoid unnecessary memory copying. A rollback simply invalidates all cache blocks except unchangeable blocks and restarts all processors from the committed checkpoint.

Five parallel program traces from an 8-processor Encore Multimax 510 were used to evaluate this evolutionary scheme [18]. Program **tgen** is a test generator; **fsim** is a fault simulator; **pace** is a circuit extractor; **phigure** is a global router, and **gravsim** is an N-body collision simulator. Each benchmark program runs for about 10 seconds. At least 80 million references are traced in each applications [18]. The caches used are 64 K two-way set associate caches with 32-byte blocks. To apply our evolutionary scheme, we need to estimate the maximum access time for the Encore Multimax 510. The longest access is the cache miss that acquires the bus last when all processors have a miss. Since a 32-byte block takes 320 nsec (nanosecond) to fetch, the longest access is 8 × 320 or 1.28 μsec (microsecond) [72]. Thus, $\Delta = 1.28$ μsec. The processor is rated at 8.5 MIPS; the maximum number of instructions executed during Δ is about 8.5 × 8 × 1.28 or 87.04. Therefore, the convergence time for the Multimax 510 is about 3.84 μsec or 262 instructions. We used the number of references to determine the session length. We simulated five different session lengths of around 262 instructions: 10, 50, 100, 500, and 1000 instructions. The interval between checkpointing sessions for the evolutionary scheme can be set at any value. For our evaluation we set it at one million references. As a comparison, we evaluated the cache-based schemes of Wu et al. and Ahmed et al.

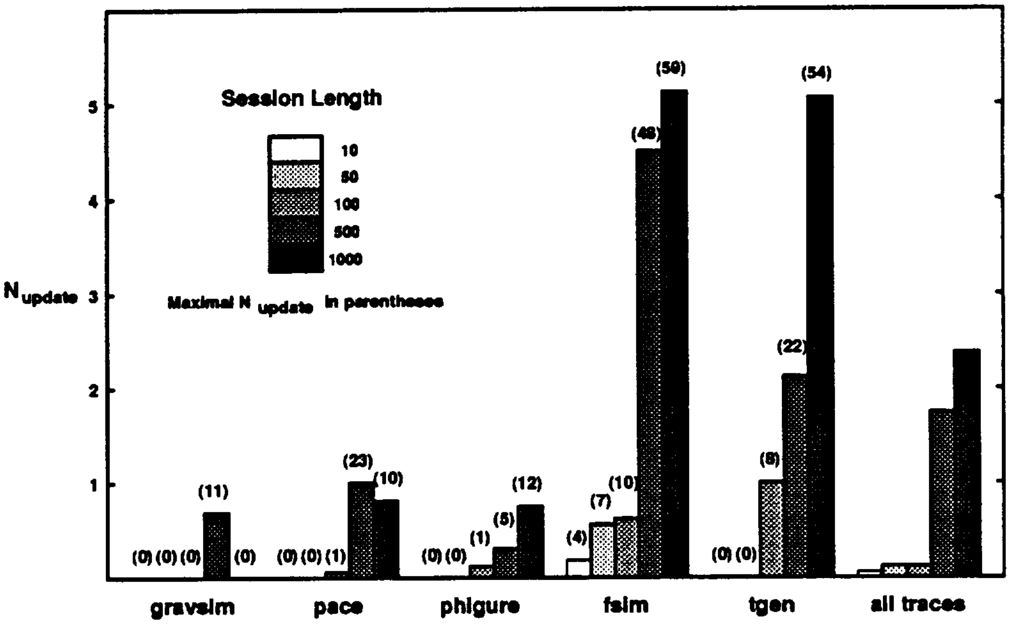

Figure 6.7. Average Number of Checkpoint Updates per Session.

The number of checkpoint updates that need to be performed during a checkpoint session depends on the amount of communication in the program. Figure 6.7 presents the average and maximum number of updates observed during a checkpoint session for each of the programs. For all but the largest session lengths in **fsim** and **tgen**, the average number of updates is at most one. For the longer sessions in **fsim** and **tgen**, the average is driven up by a few sessions with many updates. For all traces combined, however, even with a session length of 1000, the average number of updates is only around 2.5. We also found that the checkpoint size for an checkpoint update is either one or two. This indicates that local checkpoint updating only produces a limited run-time overhead.

A comparison can be made between the average checkpoint frequency for the evolutionary scheme and the other cache-based schemes. It should be noted that the checkpoint frequency for the evolutionary scheme can be controlled by adjusting the interval between sessions, while the checkpointing frequency for the other schemes is predetermined by the communication patterns of the applications we traced. For the evolutionary schemes we consider both the initial checkpoints in the session and the further updates to calculate the average frequency. The checkpoint frequencies are plotted in Figure 6.8. For a session interval of one million and a session length of 500, the checkpoint frequency varies between 1 and 2.5 per million accesses. On the other hand, the frequency for the globally synchronized scheme varies between 1.7 and 1500 per million accesses, and the frequency for the communication-synchronized scheme varies between 200 and 1000 per million references. The overhead of cache-based checkpointing depends on the number of cache blocks that are marked unchangeable (the checkpoint size) since extra cycles are needed to write these blocks back before they can be used. Figure 6.9 presents the sum of the sizes for all checkpoints during the execution of the program. This total checkpoint size is about an order of magnitude smaller for the evolutionary scheme than for the other schemes [4]. All the data show that the evolutionary scheme can provide checkpointing with a more controllable frequency and at a lower cost than previous schemes.

6.5.2. Shared virtual memory system

A shared virtual memory system supports a shared-memory programming model in a distributed computer environment [62]. An interprocessor memory access may be implemented as an *RPC* (synchronized message) over a network. A communication synchronized checkpointing scheme similar to those for multiprocessor systems was proposed by Wu and

[4] It may be worth noticing that the total checkpoint size is basically determined by the number of checkpoint sessions and the size of the initial checkpoints in each checkpoint session, since the number and size of checkpoint updates in the evolutionary scheme are limited. The number of checkpoint sessions can be controlled with proper placements of checkpoint sessions.

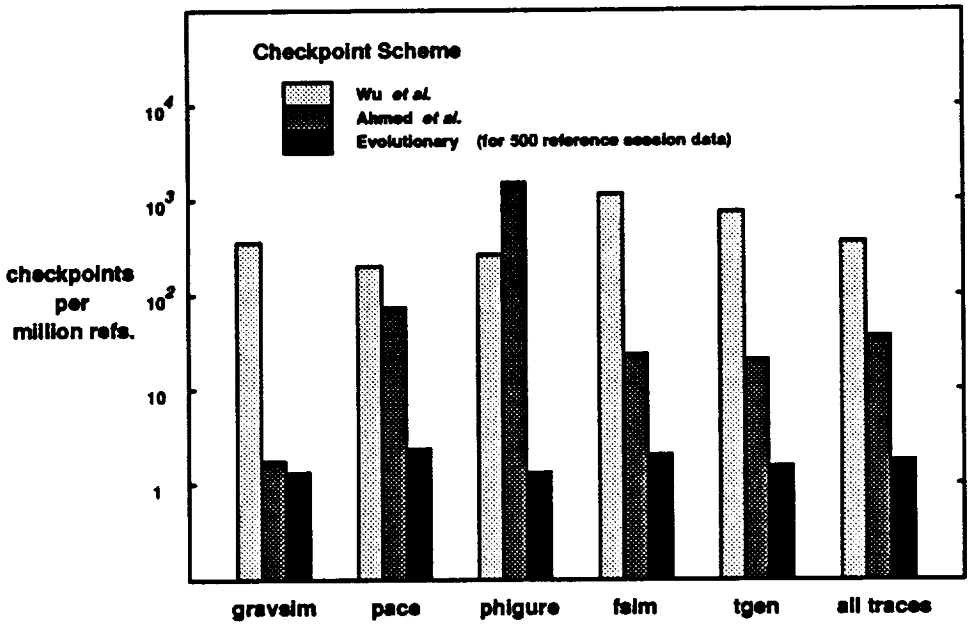

Figure 6.8. Scheme Comparison: Checkpoint Frequency.

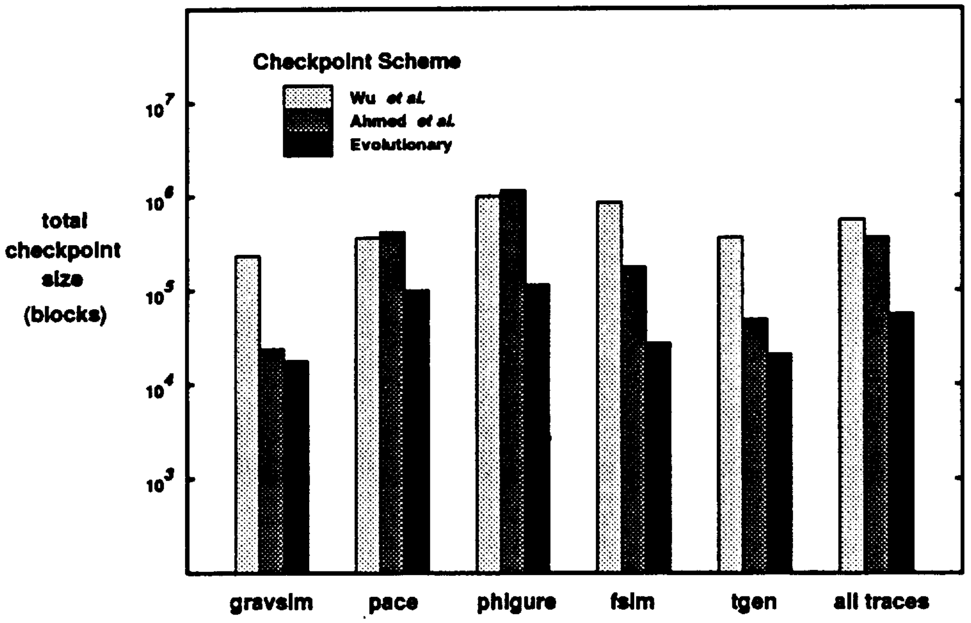

Figure 6.9. Scheme Comparison: Total Checkpoint Size.

Fuchs [16]. In such a system, the virtual memory is shared, cached in the main memory of individual processing nodes, and backed up on a stable storage. A checkpoint operation consists of flushing all dirty pages and saving processor registers to the stable storage. Whenever there is a remote access to a dirty page, the source processor takes a checkpoint. The checkpoint operation at the destination processor is eliminated since the source processor logs the requested page as a part of its checkpoint. If the destination processor rolls back, it can access the logged page from the checkpoint. Since the system is expected to be recoverable after node crashes, a shadow page system is used to accommodate the last committed virtual space (checkpoint) and the working space between checkpoints.

Similar to the multiprocessor case, our evolutionary scheme can be mapped to the distributed virtual memory case to provide controllable checkpointing. In this case, the message server is the pager process, and communication is a remote access to a dirty page. A checkpoint operation is performed at the source node during checkpoint sessions. A remote memory access is synchronized as the result of the *RPC* mechanism. Communication delay is likely to be bounded since page size is limited and the network is usually dedicated to the system. The timeout mechanism in *RPC* will further ensure the communication bound.

A global interrupt for the ckp_start and ckp_end broadcasting is not possible in a distributed-memory system, thus we need to use message broadcasting for ckp_start and ckp_end. To reduce the cost of flushing dirty pages, checkpoint operations may mark the local dirty pages unchangeable in memory. The marked pages are committed after the checkpoint session ends. A recovery simply restarts all processes from the recovery line. Unlike the multiprocessor case, the shadow pages needed for our evolutionary scheme are already used in the Wu and Fuchs scheme. Thus, our scheme incurs no additional memory overhead.

6.6. Summary

In this chapter we have presented an evolutionary checkpointing strategy for concurrent processes. This checkpointing scheme starts from a potentially inconsistent recovery line by checkpointing individual processes independently. Local checkpoints are updated whenever there is communication during the checkpoint session. This local checkpoint updating makes the recovery line evolve into a globally consistent recovery line. We showed that the convergence time from an inconsistent recovery line to a consistent one is three times the maximal communication latency upper bound.

We verified the low overhead of our evolutionary scheme by measurements on different computer systems. Unlike globally synchronized checkpointing schemes, our evolutionary scheme requires no global synchronization protocols. The evolutionary approach provides controllable checkpointing in contrast to the communication synchronized schemes. The trace-based evaluation has shown that our scheme can achieve low-cost checkpointing at a controllable interval for error recovery in multiprocessors and distributed virtual memory systems. However, our scheme is limited by the requirement of synchronous communication with a bounded latency. In the systems with low overhead synchronization mechanisms, our scheme may not be necessary, since a global synchronization scheme may be simpler to implement than our approach.

CHAPTER 7.

CONCLUSIONS

7.1. Summary

This thesis has studied a checkpoint-based forward recovery strategy for parallel and distributed systems. The replication of a task in this strategy makes forward recovery independent of the computation, since the correct next new state is obtainable from some error-free replica even when some other replicas go astray. Optimistic execution is used for fast recovery without reprocessing due to rollback. To identify the correct replica, a rollback (validation) task is scheduled to generate a diagnostic checkpoint.

In Chapter 2, a general description of this forward recovery strategy was presented. We also discussed the design parameters of a forward recovery scheme based on this strategy. Base size indicates the minimal processor redundancy used by a scheme. Scheduling depth describes the limit on the retries of the rollback validation. It implicitly determines how many of the past uncommitted checkpoints will be kept. The recursive schemes keep all uncommitted checkpoints and can utilize all of the forward recovery potentials in a scheme. The nonrecursive schemes approximate their recursive counterparts with limited retries. Rollback size describes the processor redundancy used by rollback validation. It is related to the success rate of rollback validation. The schemes derived from our forward recovery strategy can handle the performance degradation both naturally and gracefully.

Chapter 3 presented an analytical evaluation of our forward recovery strategy, while Chapter 4 gave an experimental evaluation based on a distributed implementation. It has been shown that the recovery schemes based on our forward recovery strategy can achieve a near error-free execution under faults and use an average redundancy less than TMR. These schemes can also be designed to handle graceful degradation. The checkpointing overhead is inherent in any checkpoint-based schemes and has to be minimized by placing checkpoints optimally. The comparison test time is more significant than the restart time in a scheme using comparison tests. The centralized file server affects the performance of a particular scheme through serialized file accesses.

A compiler-assisted technique was studied in Chapter 5. This technique can insert static checkpoints in user programs in a user transparent manner. The heart of the static checkpoint insertions is the instruction-based time measure for checkpoint maintenance. This measure can describe the progress of computation and have a relationship with the execution time of this computation. The interrupt-driven mechanism can obtain a consistent checkpoint interval with low overhead but does require hardware support. The polling mechanism can also generate a stable and scalable checkpoint interval. The scheme based on loop iteration count is more favorable than other basic block-based schemes.

Chapter 6 described an evolutionary approach to concurrent checkpointing with low overhead. This approach avoids rollback propagation by establishing a consistent recovery line (checkpoint). It starts with an inconsistent recovery line consisting of independently made local checkpoints. The local checkpoints are updated during the checkpoint session such that the recovery line evolves to a consistent one. This approach requires no checkpointing coordination or logging. We have proven that the convergence time to a consistent recovery line can be very short. This approach can be applied to cache-based multiprocessor and distributed-memory systems to eliminate the excessive checkpointing induced by the common communication-induced checkpoint schemes.

7.2. Discussion and Future Research

7.2.1. Forward recovery in parallel and distributed systems

There are several issues that we have not studied in detail and that are worth further investigation. First, how can our schemes be implemented in parallel systems such as shared-memory systems? The implementation in a message-passing parallel system can be very similar to our implementation in the distributed environment. The major difficulty with shared-memory systems is the common failure mode of the main memory.

Second, the incorporation of N-version programming or recovery blocks needs to be studied further for software fault tolerance. The variable execution in a program version or recovery block may affect the analytical predictions based on the constant execution time (Section 3.2). Another problem is the usually limited number of program versions or recovery blocks. A recursive scheme may run out of alternatives before a valid checkpoint is obtained.

Third, other alternatives to construct checkpoints are also important, since the checkpoint size is the key factor that determines the overheads such as checkpoint, restart and comparison times. The run-time image of a process as a checkpoint is very conservative, since many variables in the process space are usually not state variables for this process. Any error in these nonstate variables should not be detected as a bad checkpoint since they have no bearing on computation. However, any checkpoints based on state variable extraction may require specific knowledge about the computation the process performs.

7.2.2. Compiler-assisted static checkpoint insertion

The checkpoint insertions at the intermediate code level (e.g., the register transfer language) may not be the best place in which to achieve an architecture-independent insertion. For the loop count based static scheme, implementing it at the source level has two advantages. First, the insertion boundary in source code is clearly defined and thus this scheme can be easily implemented at the source language level. Second, the inserted checkpoint operations are visible to a source level symbolic debugger, while the intermediate inserted code may confuse the debugger.

The interrupt-driven mechanism for checkpoint interval maintenance needs further study to achieve a static checkpoint insertion. For example, a hardware instruction count register can be added as part of the process context. It is decremented when an instruction is executed. Once it reaches zero, an interrupt of checkpointing can obtain a static checkpoint at an instruction boundary with no overhead for polling checkpoints.

There is a more important question regarding the static checkpoints in parallel and distributed computations. If the answer is positive, the undeterministic nature of these computations makes it very difficult to generate a reproducible checkpoint or recovery line if it is possible. We need to investigate the mechanisms that can accomplish this task.

7.2.3. Evolutionary concurrent checkpointing

During a checkpointing session, a message exchange between processes results in a local checkpoint update for both the sender and receiver in our current scheme. Other alternatives are possible if nondeterministic executions are allowed. For example, reading a dirty block from a nonlocal cache may be treated as a message exchange in a cache-based multiprocessor system [15]. In this case, only a local checkpoint update at the source processor is required. However, this also leads to nondeterministic execution. Another similar case is writing to a dirty block in a nonlocal cache.

In this thesis, our evolutionary checkpointing will benefit from further studies that evaluate this approach in terms of implementation and experiments, especially the performance evaluation against other approaches such as communication synchronized schemes. In addition, the detailed applications of our approach to cache-based multiprocessor and distributed-memory systems need to be studied further. Another possibility is to use a software approach to implement our approach in the cache-aided error recovery schemes in shared-memory multiprocessor systems. For example, Intel 860 has a cache flush instruction which can be used to take a checkpoint. The question is, "Is it possible to implement the evolutionary approach purely in software?" If not, what minimal hardware support is needed?

APPENDIX A.

ANALYTICAL DERIVATIONS

A.1. DMR-F-1

In Chapter 3, we gave an analysis of DMR-F-1 based on the probabilities of successful lookaheads and rollbacks, using a simple ratio analysis (i.e., $\frac{s}{(n+r)}$ and $\frac{r}{n+r}$). In this section, we demonstrate that this analysis is correct by obtaining the same result using the conventional method: the recursive equation. If an error occurs, DMR-F-1 behaves differently in the last checkpoint interval. Since no lookahead execution is possible for this interval, a rollback is always required during recovery. An n-session computation consists of n-1 lookaheadable sessions followed by a rollbackable session. Duda has an excellent analysis for this rollback situation [4]. Besides, the performance degradation contribution of the last rollbackable session is proportional to $\frac{1}{n}$, while that of the n-1 lookaheadable sessions is to $\frac{n-1}{n}$. The approximation of an n-session computation with an n lookaheadable sessions is adequate even for a moderate n. Therefore, our analysis focuses on the situation of an n lookaheadable sessions.

Let T_n be the expected execution time for an n-session (lookaheadable) computation. Let p_l and p_r be the probabilities of successful lookahead and rollback in DMR-F-1, respectively. Thus,

$$T_n = \begin{cases} \Delta + t_k + T_{n-1}, & 1 - p_l - p_r \\ \Delta + t_k + t_r + 2.5t_t + T_{n-1}, & p_l \\ 2\Delta + 2t_k + 2t_r + 3t_t + T_n, & p_r \end{cases}$$

Thus,

$$\begin{aligned} T_n &= (1 - p_l - p_r)(\Delta + t_k + T_{n-1}) + p_l(\Delta + t_k + t_r + 2.5t_t + T_{n-1}) \\ &\quad + p_r(2\Delta + 2t_k + 2t_r + 3t_t + T_n). \end{aligned}$$

In other words,

$$T_n = (\Delta + t_k) + \frac{p_l}{1 - p_r}(t_r + 2.5t_t) + \frac{p_r}{1 - p_r}(2\Delta + 2t_k + 2t_r + 3t_t) + T_{n-1}.$$

Solving this equation with the initial condition, $T_0 = 0$, we have

$$T_n = n(\Delta + t_k) + \frac{np_l}{1 - p_r}(t_r + 2.5t_t) + \frac{np_r}{1 - p_r}(2\Delta + 2t_k + 2t_r + 3t_t).$$

This is exactly the result in Section 3.3.2 with $T_e = T_n$.

A.2. DMR-F-2

In DMR-F-2, tree situations can happen with respect to the computation progress if there is at least an erroneous checkpoint produced during the normal paired execution. First, the computation is recovered through successful lookaheads. This is the case in which one of the uncommitted checkpoints is correct and at least one of the two diagnosis checkpoints is correct. Second, the computation rolls back one session and finishes in the next session. In this case, both of the original tasks fail and the validation pair succeeds.

The checkpoints generated by the validation pair are correct and can be committed. Third, the computation rolls back two sessions and makes no progress at all. This is when the rollback validation also fails, and no valid checkpoint can be identified. Let l, s, r be the average number of successful lookaheads, one session rollbacks and two session rollbacks, respectively. Their corresponding probabilities are

$$\begin{aligned} p_l &= 2(1-p_f)p_f(1-p_f^2), \\ p_s &= p_f^2(1-p_f)^2, \\ p_r &= 2(1-p_f)p_f p_f^2 + p_f^2\left(2(1-p_f)p_f + p_f^2\right). \end{aligned}$$

Since the expected number of computations is $n + r$, we can expect the following:

$$\begin{aligned} p_l &= \frac{l}{n+r}, \\ p_s &= \frac{s}{n+r}, \\ p_r &= \frac{r}{n+r}. \end{aligned}$$

In other words,

$$\begin{aligned} l &= \frac{np_l}{1-pr}, \\ s &= \frac{np_s}{1-pr}, \\ r &= \frac{np_r}{1-pr}. \end{aligned}$$

For each successful lookahead execution, one restart and 3.5 checkpoint comparisons are exercised, while two restarts and five comparisons are made for each of rollbacks. Thus,

$$\begin{aligned} T_e &= n(\Delta + t_k) + l(t_r + 3.5t_t) + s(\Delta + t_k + 2t_r + 5t_t) + r(2\Delta + 2t_k + 2t_r + 5t_t) \\ &= n(\Delta + t_k)\left(1 + \frac{p_s + 2p_r}{1 - p_r}\right) + nt_r\frac{p_l + 2p_s + 2p_r}{1 - p_r} + nt_t\frac{3.5p_l + 5p_s + 5p_r}{1 - p_r}, \end{aligned}$$

and thus the relative execution time

$$R_e = \frac{T_e}{T_0} = 1 + \frac{p_s + 2p_r}{1 - p_r} + \frac{p_l + 2p_s + 2p_r}{1 - p_r}\frac{t_r}{\Delta + t_k} + \frac{3.5p_l + 5p_s + 5p_r}{1 - p_r}\frac{t_t}{\Delta + t_k}.$$

The only difference between DMR-F-2 and DMR-F1 is that DMR-F-2 uses one more processor (thus producing one more checkpoint) than DMR-F-1 does during recovery. Following a similar line in the DMR-F-1 analysis we have

$$\begin{aligned} \int_0^{T_e} N_c(t)dt &= (n - l)(\Delta + t_k) + 3l(\Delta + t_k + t_r + t_t) + 9l(2.5t_t) \\ &\quad + st_r + 3s(\Delta + t_k + t_r + t_t) + 9s(5t_t) \\ &\quad + r(\Delta + t_k + t_r) + 3r(\Delta + t_k + t_r + t_t) + 9r(5t_t) \\ &= T_e + 2T_0\frac{p_l + p_s + p_r}{1 - p_r} + 2nt_r\frac{p_l + p_s + p_r}{1 - p_r} + 2nt_t\frac{11p_l + 21.5p_s + 21.5p_r}{1 - p_r}. \end{aligned}$$

Therefore,

$$N_c = 1 + 2\frac{p_l + p_s + p_r}{(1 - p_r)R_e} + 2\frac{p_l + p_s + p_r}{(1 - p_r)R_e}\frac{t_r}{\Delta + t_k} + 2\frac{11p_l + 21.5p_s + 21.5p_r}{(1 - p_r)R_e}\frac{t_t}{\Delta + t_k}.$$

$$\begin{aligned} \int_0^{T_e} N_p(t)dt &= 2(n - l)(\Delta + t_k) + 2lt_t + 6l(\Delta + t_k + t_r + 2.5t_t) + 2s(t_r + t_t) \\ &\quad + 6s(\Delta + t_k + t_r + 5t_t) + 2r(\Delta + t_k + t_r + t_t) + 6r(\Delta + t_k + t_r + 5t_t) \\ &= 2T_e + 4T_0\frac{p_l + p_s + p_r}{1 - p_r} + 4nt_r\frac{p_l + p_s + p_r}{1 - p_r} + 4nt_t\frac{2.5p_l + 5.5p_s + 5.5p_r}{1 - p_r}, \end{aligned}$$

and

$$N_p = 2 + 4\frac{p_l + p_s + p_r}{(1-p_r)R_e} + 4\frac{p_l + p_s + p_r}{(1-p_r)R_e}\frac{t_r}{\Delta + t_k} + 4\frac{2.5p_l + 5.5p_s + 5.5p_r}{(1-p_r)R_e}\frac{t_t}{\Delta + t_k}.$$

It can be shown that the following results for a single file server system:

$$\begin{aligned} R_e(fs) &= 1 + \frac{p_s + 2p_r}{1-p_r} + \frac{p_l + 5/3p_s + 5/3p_r}{1-p_r}\frac{3t_r}{\Delta + 2t_k} \\ &\quad + \frac{3.5p_l + 5p_s + 5p_r}{1-p_r}\frac{t_t}{\Delta + 2t_k} + \frac{p_l + p_s + p_r}{1-p_r}\frac{3t_k}{\Delta + 2t_k}, \end{aligned}$$

$$\begin{aligned} N_c(fs) &= 1 + 2\frac{p_l + p_s + p_r}{(1-p_r)R_e(fs)} + 2\frac{p_l + p_s + p_r}{(1-p_r)R_e(fs)}\frac{3t_r}{\Delta + 2t_k} \\ &\quad + 2\frac{11p_l + 21.5p_s + 21.5p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta + 2t_k} + 2\frac{p_l + p_s + p_r}{(1-p_r)R_e(fs)}\frac{3t_k}{\Delta + 2t_k}, \end{aligned}$$

and

$$\begin{aligned} N_p(fs) &= 2 + 4\frac{p_l + p_s + p_r}{(1-p_r)R_e} + 4\frac{p_l + p_s + p_r}{(1-p_r)R_e}\frac{3t_r}{\Delta + 2t_k} \\ &\quad + 4\frac{2.5p_l + 5.5p_s + 5.5p_r}{(1-p_r)R_e}\frac{t_t}{\Delta + 2t_k} + 4\frac{p_l + p_s + p_r}{(1-p_r)R_e}\frac{3t_k}{\Delta + 2t_k}. \end{aligned}$$

A.3. TMR-F

In TMR-F, the trio of replicated tasks continues when there is no erroneous checkpoint at the end of Δ. If there is a match in checkpoints, TMR-F performs a forward recovery via masking off the erroneous checkpoint. If all checkpoints are different, a rollback is scheduled. The rollback probability is given as

$$p_r = 3p_f^2 - 2p_f^3.$$

The number of rollbacks, r, is expected to satisfy

$$\begin{aligned} p_r &= \frac{r}{n+r}, \\ r &= \frac{np_r}{1-p_r}. \end{aligned}$$

The expected execution time is

$$\begin{aligned} T_e &= n(\Delta + t_k) + r(\Delta + t_k + t_r + 3t_t) \\ &= n(\Delta + t_k)\left(1 + \frac{p_r}{1-p_r}\right) + nt_r\frac{p_r}{1-p_r} + nt_t\frac{3p_r}{1-p_r}. \end{aligned}$$

Therefore, the relative execution time is

$$R_e = \frac{T_e}{T_0} = 1 + \frac{p_r}{1-p_r} + \frac{p_r}{1-p_r}\frac{t_r}{\Delta + t_k} + \frac{3p_r}{1-p_r}\frac{t_t}{\Delta + t_k}.$$

For a single file server system, the restart time and checkpoint time are tripled for the TMR-F scheme due to the file access by three processes at the same time. Thus,

$$R_e(fs) = 1 + \frac{p_r}{1-p_r} + \frac{p_r}{1-p_r}\frac{3t_r}{\Delta + 3t_k} + \frac{3p_r}{1-p_r}\frac{t_t}{\Delta + 3t_k}.$$

The number of checkpoints is one (the most recently committed checkpoint) for the normal pair and four for checkpoint validation. Therefore,

$$\begin{aligned} \int_0^{T_e} N_c(t)dt &= n(\Delta + t_k) + 4r(3t_t) + r(\Delta + t_k + t_r) \\ &= T_e + 6nt_t\frac{p_r}{1-p_r}, \end{aligned}$$

and

$$N_c = 1 + \frac{6p_r}{(1-p_r)R_e}\frac{t_t}{\Delta + t_k}.$$

Since t_r do not appear in nc, the impact of the single file server is reflected indirectly through R_e:

$$N_c(fs) = 1 + \frac{6p_r}{(1-p_r)R_e(fs)}\frac{t_t}{\Delta + 3t_k}.$$

The number of processors is always three. Thus,

$$N_p = max(N_p) = 3.$$

A.4. DMR-B-1

In this scheme, there is no forward recovery. If there is an error in any of the two original replicated tasks, the task will be rolled back repeatedly until there is a match in the uncommitted checkpoints produced by both the original task pair and the rollback tasks. Two situations can cause a rollback: (1) there is one error-free checkpoint produced by the original task pair and the rollback iterations need only to generate another correct checkpoint; and (2) both checkpoints are erroneous for the original task pair run and the rollback iterations need to produce two valid checkpoints. Their probabilities are $p_1 = 2p_f(1-p_f)$ and $p_2 = p_f^2$, respectively. Let the i-th iteration of the rollback retries lead to a match in checkpoints, and let s be the situation number. The conditional probability given a rollback situation for i is then

$$p(i,s) = \begin{cases} p_f^{i-1}(1-p_f), & i \geq 1, s = 1, \\ (i-1)p_f^{i-2}(1-p_f)^2, & i \geq 2, s = 2. \end{cases}$$

Since all of the checkpoint comparisons during the previous $i-1$ iterations have failed for a given i, the number of checkpoint comparisons can be calculated as

$$\sum_{j=1}^{i-1}(i+1) = \frac{(i+2)(i-1)}{2}.$$

The number of checkpoint comparisons for the i-th iteration is expected as 1.5 for situation 1 since there is a valid checkpoint among the first two uncommitted checkpoints (produced by the original task pair) and $\frac{i+1}{2}$ for situation 2. Thus, the expected execution time is

$$T_e = n(\Delta + t_k) + np_1 \left\{ t_t + \sum_{i=1}^{\infty} \left[i(\Delta + t_k + t_r) + t_t(\frac{(i+2)(i-1)}{2} + 1.5) \right] p(i|1) \right\}$$

$$\begin{aligned}
&+np_2\left\{t_t+\sum_{i=2}^{\infty}\left[i(\Delta+t_k+t_r)+t_t(\frac{(i+2)(i-1)}{2}+\frac{i+1}{2})\right]p(i|2)\right\}\\
=\;& n(\Delta+t_k)\left(1+\frac{p_1+2p_2}{1-p_f}\right)+nt_r\frac{p_1+2p_2}{1-p_f}\\
&+nt_t\frac{p_f(10-15p_f+18p_f^2-7p_f^3)}{2(1-p_f)^2}\frac{t_t}{\Delta+t_k},
\end{aligned}$$

and

$$R_e = 1+\frac{p_1+2p_2}{1-p_f}+\frac{p_1+2p_2}{1-p_f}\frac{t_r}{\Delta+t_k}+\frac{p_f(10-15p_f+18p_f^2-7p_f^3)}{2(1-p_f)^2}\frac{t_t}{\Delta+t_k}.$$

The number of checkpoints accumulates as the rollback iteration increases. For the j-th iteration, there are $j+2$ checkpoints. Thus,

$$\begin{aligned}
\int N_c(t,i,s)dt &= \sum_{j=1}^{i}(j+2)(\Delta+t_k+t_r+t_t)+\begin{cases} 1.5(i+3)t_t, & s=1\\ \frac{(i+3)(i+1)}{2}t_t, & s=2\end{cases}\\
&= \left[\frac{(i+1)i}{2}+2i\right](\Delta+t_k+t_r)+\left[\frac{i(i+1)(i+2)}{3}+\frac{i(i+1)}{2}\right]t_t\\
&\quad+\begin{cases} 1.5(i+3)t_t, & s=1\\ \frac{(i+3)(i+1)}{2}t_t, & s=2\end{cases}\;.
\end{aligned}$$

Therefore,

$$\int_0^{T_e} N_c(t)dt = n(\Delta+t_k)+np_1\sum_{i=1}^{\infty}\int N_c(t|i,1)dtp(i|1)+np_2\sum_{i=1}^{\infty}\int N_c(t|i,2)dtp(i|2)$$

and,

$$\begin{aligned}
N_c =\;& \left[1+\frac{p_1+3p_2}{(1-p_f)^2}+\frac{2p_1+4p_2}{1-p_f}\right]\frac{1}{R_e}++\left[\frac{p_1+3p_2}{(1-p_f)^2}+\frac{2p_1+4p_2}{1-p_f}\right]\frac{t_r}{(\Delta+t_k)R_e}\\
&+\left[\frac{2p_1+8p_2}{(1-p_f)^3}+\frac{p_1+6p_2}{(1-p_f)^2}+\frac{3/2p_1+3p_2}{1-p_f}+\frac{9p_1+3p_2}{2}\right]\frac{t_t}{(\Delta+t_k)R_e}.
\end{aligned}$$

Similarly

$$N_p = 1 + \frac{1}{R_e} + \frac{p_1 + p_2}{R_e}\frac{t_k}{\Delta + t_k} + \left[\frac{p_1 + 3p_2}{(1 - p_f)^2} + \frac{p_2}{1 - p_f} + \frac{7p_1 + p_2}{2}\right]\frac{t_t}{\Delta + t_k}.$$

If a single file server is used, the spontaneous file accesses occur only during executions of the task pair. During recovery, DMR-B-1 uses only one processor and has no bearings on t_k and t_r. Thus,

$$R_e(fs) = \frac{\Delta + t_k}{\Delta + 2t_k}R_e + \frac{t_k}{\Delta + 2t_k}$$

$$N_c(fs) = \frac{\Delta + t_k}{\Delta + 2t_k)}N_c + + \frac{t_k}{(\Delta + 2t_k)R_e(fs)},$$

$$N_p(fs) = \frac{\Delta + t_k}{\Delta + 2t_k}N_p + \frac{2t_k}{(\Delta + 2t_k)R_e(fs)}.$$

A.5. DMR-B-2

Like DMR-B-1, this scheme employs the recursive rollback to find a pair of matched checkpoints. However, they differ in that DMR-B-2 uses two rollback tasks whereas DMR-B-1 uses one. If the task execution succeeds at the i-th iteration, three situations can happen. (1) There are no error-free checkpoints produced in the previous $i - 1$ iterations and two error-free checkpoints at the i-th iteration; (2) there is a correct checkpoint during the previous $i-1$ iteration and one for the i-th iteration; and (3) there is a correct checkpoint for the previous $i - 1$ iterations and two for the i-th iteration. It can be shown that the probability of the task succeeding in the i-th iteration ($i \geq 1$) in situation s is

$$p(i,s) = \begin{cases} p_f^{2(i-1)}(1 - p_f)^2, & s = 1, \\ 4(i-1)p_f^{2(i-1)}(1 - p_f)^2, & s = 2, \\ 2(i-1)p_f^{2i-3}(1 - p_f)^3, & s = 3. \end{cases}$$

To calculate the number of checkpoint comparisons needed for a given i, we need to know the order in which we compare a checkpoint against a set of checkpoints. We assume that the checkpoints produced are stored in a list according to their creation time. When a new checkpoint is compared against the list, the order of comparison is from the first to the last one in the list. It can be shown that the number of comparisons (n_c) for a successful execution of i iterations is

$$\begin{aligned} n_c(i,s) &= \sum_{j=1}^{i-1}[4(j-1)+1] + \begin{cases} 4(i-1)+1-\delta(i-1), & s=1 \\ 2(i-1)+1/2, & s=2 \\ (i-1)+1/2, & s=3 \end{cases} \\ &= 2(i+1)i - 7i + 3 + \begin{cases} 4i-3, & s=1, \\ 2i-3/2, & s=2, \\ i-1/2, & s=3. \end{cases} \end{aligned}$$

Thus, the expected execution time is

$$\begin{aligned} T_e &= n\sum_{i=1}^{\infty}\sum_{s=1}^{3}[i(\Delta+t_k)+(i-1)t_r+n_c(i,s)t_t]\,p(i,s) \\ &= n(\Delta+t_k)\left[1+\frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}\right] + nt_r\frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2} \\ &\quad +nt_t\left[4p_f^2\frac{4p_f+3}{(1+p_f)^3(1-p_f)^2}+p_f\frac{6+2p_f-p_f^2}{(1+p_f)^2}\right]. \end{aligned}$$

$$\begin{aligned} R_e &= 1+\frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2} + +\frac{p_f(p_f^2+p_f+2)}{(1-p_f)(1+p_f)^2}\frac{t_r}{\Delta+t_k} \\ &\quad +\left[4p_f^2\frac{4p_f+3}{(1+p_f)^3(1-p_f)^2}+p_f\frac{6+2p_f-p_f^2}{(1+p_f)^2}\right]\frac{t_t}{\Delta+t_k}. \end{aligned}$$

Similarly,

$$\begin{aligned} N_c &= \left[1 + \frac{p_f(6 + 5p_f + 3p_f^2 - 3p_f^3 + p_f^4)}{(1 + p_f^3)(1 - p_f)^2}\right] \frac{1}{R_e} + \frac{p_f(6 + 5p_f + 3p_f^2 - 3p_f^3 + p_f^4)}{(1 + p_f^3)(1 - p_f)^2 R_e} \frac{t_r}{\Delta + t_k} \\ &+ \frac{p_f(137 + 10p_f - 16p_f^2 - 22p_f^3 + 19p_f^4 + 4p_f^5 - 4p_f^6}{(1 - p_f)^3(1 + p_f)^4} \frac{t_t}{\Delta + t_k}, \end{aligned}$$

$$N_p = max(N_p) = 2.$$

Given the file server impact,

$$\begin{aligned} R_e(fs) &= 1 + \frac{p_f(p_f^2 + p_f + 2)}{(1 - p_f)(1 + p_f)^2} + + \frac{p_f(p_f^2 + p_f + 2)}{(1 - p_f)(1 + p_f)^2} \frac{2t_r}{\Delta + 2t_k} \\ &+ \left[4p_f^2 \frac{4p_f + 3}{(1 + p_f)^3(1 - p_f)^2} + p_f \frac{6 + 2p_f - p_f^2}{(1 + p_f)^2}\right] \frac{t_t}{\Delta + 2t_k}, \end{aligned}$$

$$\begin{aligned} N_c(fs) &= \left[1 + \frac{p_f(6 + 5p_f + 3p_f^2 - 3p_f^3 + p_f^4)}{(1 + p_f^3)(1 - p_f)^2}\right] \frac{1}{R_e(fs)} \\ &+ \frac{p_f(6 + 5p_f + 3p_f^2 - 3p_f^3 + p_f^4)}{(1 + p_f^3)(1 - p_f)^2 R_e(fs)} \frac{2t_r}{\Delta + 2t_k} \\ &+ \frac{p_f(137 + 10p_f - 16p_f^2 - 22p_f^3 + 19p_f^4 + 4p_f^5 - 4p_f^6}{(1 - p_f)^3(1 + p_f)^4} \frac{t_t}{\Delta + 2t_k}, \\ N_p(fs) &= 2. \end{aligned}$$

A.6. Self-Testable Scheme

In this scheme, a test on a checkpoint is enough to validate this checkpoint, and thus no rollback validation is needed. Let l and r be the expected number of successful lookaheads and rollbacks, respectively. Thus,

$$\begin{aligned} p_l &= 2p_f(1 - p_f), \\ p_r &= p_f^2, \\ l &= \frac{np_l}{1 - p_r}, \end{aligned}$$

$$r = \frac{np_r}{1-p_r}.$$

Unlike DMR-F-1 and DMR-F-2, the self-testable scheme rolls back only one computation session when both checkpoints of the original task pair are tested erroneous. Thus,

$$\begin{aligned}
T_e &= n(\Delta + t_k) + l(t_r + t_t) + r(\Delta + t_k + t_r + 2t_t) \\
&= n(\Delta + t_k)\left(1 + \frac{p_r}{1-p_r}\right) + nt_r\frac{p_l + p_r}{1-p_r} + nt_t\frac{p_l + 2p_r}{1-p_r}, \\
R_e &= 1 + \frac{p_r}{1-p_r} + \frac{p_l + p_r}{1-p_r}\frac{t_r}{\Delta + t_k} + \frac{p_l + 2p_r}{1-p_r}\frac{t_t}{\Delta + t_k}.
\end{aligned}$$

The number of checkpoints is one for both normal and lookahead executions and three for the checkpoint testing. Therefore,

$$\begin{aligned}
\int_0^{T_e} N_c(t)dt &= n(\Delta + t_k) + 3lt_t + lt_r + r(\Delta + t_k + t_r) + 3r(2t_t) \\
&= T_e + 2(l + 2r)t_t, \\
N_c &= 1 + 2\frac{p_l + 2p_r}{1-p_r}\frac{t_t}{\Delta + t_k}, \\
max(N_c) &= 3.
\end{aligned}$$

Since no lookahead from the erroneous checkpoint is scheduled, the number of processors used is still two during the lookahead execution. Then we have

$$N_p = max(N_p) = 2.$$

A.7. Graceful Performance Degradation Scheme

The analysis for the DMR-F-1 degradation scheme is almost identical with that of DMR-F-1. We are not going to repeat these formulas here but list the differences between this degradation scheme and DMR-F-1:

- The probability of successful lookaheads is half of that for DMR-F-1, since half of the lookaheads fail when the misscheduled lookahead process happens to be the correct lookahead.
- The probability of rollback increases to include the additional failed lookaheads.
- The number of checkpoints during the checkpoint testing is six instead of eight in DMR-F-1.
- The number of processors used is three during lookahead execution instead of five for DMR-F-1.

REFERENCES

[1] P. L'Ecuyer and J. Mallenfant, "Computing optimal checkpointing strategies for rollback and recovery systems," *IEEE Trans. Comput.*, Vol. 37, No. 4, pp. 491–496, April 1988.

[2] S. Toueg and O. Babaoglu, "On the optimum checkpoint selection problem," *SIAM J. Comput.*, Vol. 13, pp. 630–649, Aug. 1984.

[3] C. M. Krishna, K. G. Shin, and Y.-H. Lee, "Optimization criteria for checkpoint placement," *CACM*, Vol. 27, No. 6, No. 6, pp. 1008–1012, Oct. 1984.

[4] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, Vol. 16, pp. 221–229, 1983.

[5] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *CACM*, Vol. 21, No. 6, No. 6, pp. 493–499, 1978.

[6] J. W. Young, "A first order approximation to the optimal checkpoint interval," *CACM*, Vol. 17, No. 9, pp. 530–531, Sept. 1974.

[7] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice.* Springer-Verlag/Wien, 1990.

[8] D. J. Taylor and J. P. Black, "A locally correctable b-tree implementation," *Comput. J.*, Vol. 29, No. 3, pp. 269–276, June 1986.

[9] C.-C. J. Li, P. P. Chen, and W. K. Fuchs, "Local concurrent error detection and correction in data structure using virtual backpointers," *IEEE Trans. Comput.*, Vol. 38, No. 11, No. 11, pp. 1481–1492, 1989.

[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Comput. Conf.*, pp. 483–485, April 1967.

[11] C. C. Li and W. K. Fuchs, "CATCH: Compiler-assisted techniques for checkpointing," *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 74–81, 1990.

[12] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, Vol. 1, No. 2, pp. 220–232, June 1975.

[13] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic recovery schemes for distributed processes," *Proc. IEEE 2nd Symp. on Reliability in Distributed Software and Database Syst.*, pp. 124–130, 1981.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, Vol. 21, No. 7, pp. 558–566, July 1978.

[15] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. Parallel and Distributed Syst.*, Vol. 1, No. 2, No. 2, pp. 231–240, 1990.

[16] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. Comput.*, Vol. 39, No. 4, pp. 460–469, April 1990.

[17] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (carer) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 82–88, 1990.

[18] B. Janssens and W. K. Fuchs, "Experimental evaluation of multiprocessor cache-based error recovery," *Proc. Int. Conf. Parallel Processing*, Vol. I, pp. 505–508, Aug. 1991.

[19] J. F. Bartlett, "A nonstop kernel," *Proc. ACM 8th Symp. Oper. Syst. Principles*, pp. 22–29, Dec. 1981.

[20] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under unix," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, No. 1, pp. 63–75, Feb., 1985.

[21] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," *Proc. 9th Symp. Oper. Syst. Principles*, pp. 100–109, Oct., 1983.

[22] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing.," *J. Algorithms*, Vol. 11, No. 3, pp. 462–491, Sept. 1990.

[23] T. T.-Y. Juang and S. Venkatesan, "Efficient algorithms for crash recovery in distributed systems," *Proc. 10th Conf. Foundations of Software Technology and Theoretical Comput. Sci.*, pp. 349–361, 1990.

[24] T. T.-Y. Juang and S. Venkatesan, "Crash recovery with little overhead," *Proc. 11th Int. Conf. Distributed Comput. Syst.*, pp. 454–461, May 1991.

[25] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," *Proc. 8th Symp. Principles of Distributed Comput.*, Aug. 1989.

[26] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[27] P. Agrawal, "RAFT: A recursive algorithm for fault-tolerance," *Proc. Int. Conf. Parallel Processing*, pp. 814–821, 1985.

[28] P. Agrawal and R. Agrawal, "Software implementation of a recursive fault-tolerance algorithm on a network of computers," *Proc. 13th Annual Symp. Comput. Arch.*, pp. 65–72, 1986.

[29] A. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Trans. Comput. Syst.*, Vol. 2, No. 2, pp. 123–144, May 1984.

[30] S. Thanwastien, R. S. Pamula, and Y. L. Varol, "Evaluation of global rollback strategies for error recovery in concurrent processing systems," *Proc. 16th Int. Symp. Fault-Tolerant Comput.*, pp. 246–251, 1986.

[31] Y.-H. Lee and K. G. Shin, "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, Vol. 33, No. 2, No. 2, pp. 113–124, 1984.

[32] N. H. Vaidya and D. K. Pradhan, "Fault-tolerant design strategies for high reliability and safety," Tech. Rep. Manuscript, Department of Electrical and computer Engineering, University of Massachusetts at Amherst, 1992.

[33] L. M. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Proc. 8th Int. Symp. Fault-Tolerant Comput.*, pp. 3–9, 1978.

[34] J.-Y. Jou and J. A. Abraham, "Fault-tolerant matrix operation on multiple processor systems using weighted checksums," *SPIE Proc.*, Vol. 495, Aug. 1984.

[35] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, Vol. 33, No. 6, pp. 518–528, June 1984.

[36] M. A. Vouk, A. M. Paradkar, and D. F. McAllister, "Modeling execution time of multistage n-version fault-tolerant software," *Proc. COMPSAC 90*, pp. 505–511, 1990.

[37] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Comput.*, Vol. 38, No. 5, No. 5, pp. 626–636, 1989.

[38] J. M. Smith, "Implementing remote fork() with checkpoint/restart," *Tech. Committee on Oper. Syst. Newsletter*, Vol. 3, No. 1, No. 1, pp. 15–19, 1989.

[39] M. Litzkow, M. Livny, and M. Mutka, "CONDOR – A hunter of idle workstations," *Proc. 8th int. Conf. Distributed Comput. Syst.*, 1988.

[40] D. J. Taylor and M. L. Wright, "Backward error recovery in a unix environment," *Proc. 16th Int. Symp. Fault-Tolerant Comput.*, pp. 118–123, 1986.

[41] SPEC, *SPEC Newsletter*. Fremont, CA: SPEC, Feb. 1989.

[42] S. Feldman and C. Brown, "A system for program debugging via reversible execution," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, Vol. 24, No. 1, pp. 112–123, Jan. 1989.

[43] D. Pan and M. Linton, "Supporting reverse execution for parallel programs," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, Vol. 24, No. 1, pp. 124–129, Jan. 1989.

[44] L. D. Wittie, "Debugging distributed c programs by real time replay," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, Vol. 24, No. 1, pp. 57–67, Jan. 1989.

[45] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpoint for parallel programs," *Proc. 2nd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 79–88, March 1990.

[46] J. Long, W. K. Fuchs, and J. A. Abraham, "A forward recovery strategy using checkpointing in parallel systems," *Proc. Int. Conf. Parallel Processing*, Vol. 1, pp. 272–275, 1990.

[47] J. Long, W. K. Fuchs, and J. A. Abraham, "Implementing forward recovery using checkpointing in distributed systems," *Proc. 2nd IFIP Working Conf. Dependable Comput. for Critical Applications*, pp. 20–27, Feb. 1991.

[48] 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.

[49] J. S. Upadhyaya and K. K. Saluja, "A watchdog processor based general rollback technique with multiple retries," *IEEE Trans. Software Eng.*, Vol. 12, No. 1, pp. 87–95, Jan. 1986.

[50] J. S. Upadhyaya and K. K. Saluja, "An experimental study to determine task size for rollback recovery systems," *IEEE Trans. Comput.*, Vol. 37, No. 7, pp. 872–877, July 1988.

[51] R. M. Stallman, "Using and porting gnu cc," *Proc. 2nd Int. Conf. Comput. and Applications*, 1990.

[52] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, Vol. 15, No. 7, pp. 875–889, July 1989.

[53] ROSS Technology, *SPARC RISC User's Guide*. ROSS Technology, Inc., 1990.

[54] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 3–12, 1988.

[55] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," *Proc. ACM 9th Symp. Oper. Syst. Principles*, pp. 90–99, Oct. 1983.

[56] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

[57] M. Spezialetti and P. Kearns, "Efficient distributed snapshots," *Proc. 6th Int'l. Conf. Distributed Comput. Syst.*, pp. 382–388, 1986.

[58] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, Vol. 13, No. 1, pp. 23–31, Jan. 1987.

[59] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," *Proc. 10th Symp. Reliable Distributed Syst.*, pp. 2–11, 1991.

[60] Z. Tong, R. Y. Kain, and W. T. Tsai, "Rollback recovery in distributed systems using loosely synchronized clocks," *IEEE Trans. Parallel and Distributed Syst.*, Vol. 3, No. 2, pp. 246–251, March 1992.

[61] F. Cristian, "A timestamp-based checkpoint protocol for long-lived distributed computations," *Proc. 10th Symp. Reliable Distributed Syst.*, pp. 12–20, 1991.

[62] K. Li, "IVY: A shared virtual memory systems for parallel computing," *Proc. Int. Conf. Parallel Processing*, pp. 94–101, 1988.

[63] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, "Proiority inversions in real-time communication," *Proc. 10th IEEE Real-Time Syst. Symp.*, Dec. 1989.

[64] H. Tokuda and C. W. Mercer, "ARTS: Adistributed real-time kernel," *ACM Oper. Syst. Rev.*, Vol. 23, No. 3, July 1989.

[65] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *Proc. 5th ACM Symp. Principles Distributed Comput.*, pp. 229–239, 1986.

[66] J.-M. Hsu and P. Banerjee, "Hareware support for message routing in a distributed memory multicomputer," *Proc. Int. Conf. Parallel Processing*, pp. 508–515, Aug. 1990.

[67] J. Long, W. K. Fuchs, and J. A. Abraham, "Compiler-assisted static checkpoint insertion," *Proc. 22th Int. Symp. Fault-Tolerant Comput.*, 1992.

[68] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *IEEE Comput.*, Vol. 21, pp. 37–45, Feb. 1988.

[69] N. S. Bowen and D. K. Pradhan, "Vitual checkpoints: Architecture and Performance," *IEEE Trans. Comput.*, Vol. 41, No. 5, May 1992.

[70] T. P. Ng, "Checkpointing in a virtual shared memory system," Tech. Rep. UIUCDCS-R-91-1700, Department of Computer Science, University of Illinois, Dec. 1991.

[71] D. B. Hunt and P. N. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," *Proc. 17th Symp. Fault-Tolerant Comput.*, pp. 170–175, 1987.

[72] Encore Computer Corporation, *Multimax Technical Summary.* Encore Computer Corporation, Jan. 1989.

VITA

Junsheng Long received a B.S. degree in Geography in 1986 from Beijing University, Beijing, China. He also received an M.S. degree in Watershed Management in 1986 and an M.S. in Electrical Engineering in 1987 from the University of Arizona, Tucson, Arizona. While pursuing his Ph.D. degree at the University of Illinois, he held a research assistantship in the Center for Reliable and High-Performance Computing at the Coordinated Science Laboratory from 1987 to 1992. He is a member of Phi Kappa Phi and a student member of the IEEE Computer Society. Upon completing his Ph.D. degree, he will join the Department of Computer Science, the University of North Carolina at Charlotte, as an assistant professor. His research interests include parallel and distributed processing, object-oriented programming, software engineering, and fault-tolerant computing.