Chapter One
Introduction

Multicomputer systems, consisting of hundreds of processors interconnected by high-speed links, are now technologically and economically feasible[Dall88, Seit85, Whit85]. Such systems can achieve high performance for many applications at a low cost by exploiting parallelism. A large multicomputer consists of thousands of large VLSI chips and hundreds of printed circuit boards. If the entire system is likely to fail as a result of the failure of any of its components, the expected mean time between system failures is only a few hours[Peat81]. Such a high system failure rate is unacceptable for many applications that require continuous correct operation for tens or hundreds of hours. This problem will be exacerbated by the continued scaling of VLSI ICs, which will cause reliability to become a more dominant problem in the future, resulting in shorter chip lifetimes[Wood86]. Hence, the system must be able to operate correctly despite the failure of some of its components, i.e., it must be *fault tolerant*. In order to ‘‘tolerate’’ hardware faults, the system must be able to *detect* the error caused by the fault, *locate* the faulty component, *restore* a valid system state (since the state may have been corrupted by the fault), and (possibly) *reconfigure* the system so that it does not continue to use the faulty components[Rand78]. The topic of this thesis is distributed *error recovery*: the restoration of a valid system state following a hardware failure.

The scope of this thesis is limited to the support of ‘‘general-purpose’’ applications which have no hard real-time constraints. In such an environment

fault tolerance through hardware redundancy and error masking, e.g. TMR (triple modular redundancy), is not appropriate. Such techniques involve a very high level of overhead relative to a system with no fault tolerance (over 200 percent for TMR) which is not reasonable for non-real-time systems.

Software error recovery techniques based on dynamic redundancy can be broken into two classes: forward and backward error recovery. *Forward error recovery* techniques are inappropriate for general-purpose multicomputers because they require detailed knowledge of the behavior of the applications. Such techniques examine the failed state of an application and then correct, or ‘‘adjust’’, the failed state so that it meets specified acceptance criteria — hence no work is lost.

Backward error recovery techniques involve *checkpointing* and *recovery/rollback*. The states of all application processes that can fail are periodically saved (checkpointed) to stable storage, e.g. reliable disks. In the event of a hardware failure all processes which were affected or could have been affected by the error are *rolled back* to their checkpoints. Only backward error recovery techniques are suitable for a environment supporting a variety of applications. However, existing fault tolerance schemes using backward error recovery tend to have high levels of overhead and/or restrict the behavior of application software and message delivery protocols. This thesis presents new backward error recovery techniques that have low levels of overhead (which can be ‘‘tuned’’ to the needs of different applications), ensure efficient use of the communication network, and do not restrict the behavior of applications.

Given the difficulty of writing large distributed applications it is highly

desirable that the *virtual machine* visible to the programmer not be made more complex by the fault tolerance scheme. Hence, we restrict our attention to *application-transparent* techniques, where no restrictions are made on the behavior and structure of application software. Hence, non-application-transparent techniques such as recovery blocks and recovery schemes used in transaction-based systems are only briefly mentioned. We divide existing application-transparent error recovery techniques into two main classes: Message Logging and Coordinated Checkpointing. The overhead of these schemes includes the time and storage overhead incurred during normal operation, restrictions made on application processes and the message delivery system used, and time lost during recovery.

In the proposed techniques, checkpointing sessions are initiated by ‘‘timers’’ associated with each process[Bari83]. The frequency of checkpointing can thus be tuned to the specific needs of a task: a higher frequency of checkpointing results in higher overhead but in less work being lost when recovery is necessary. Thus the fault tolerance scheme can be ‘‘configured’’ to meet the requirements of each application instead of penalizing all applications equally in order to satisfy the requirements of the most demanding application.

Since the goal of multicomputers is to achieve high performance, the fault tolerance scheme must not slow down the system significantly, even with applications that are ‘‘communication-intensive’’ (exploit fine-grain parallelism). Thus performance and storage overhead during normal operation should be minimized at the cost of a potentially expensive recovery.

A recovery scheme, intended for use on a large multicomputer, should have

the following attributes:

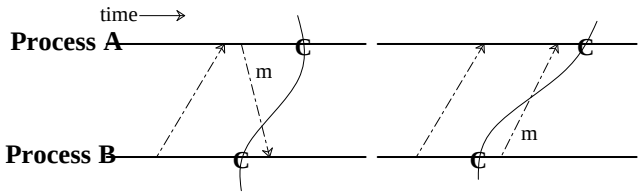
- *application-transparent*
- *minimal restrictions on application behavior*
- *low-overhead*: determined by the frequency of checkpointing, the ‘‘cost’’ of saving each checkpoint, extra operations required during normal computation for error detection and maintenance of bookkeeping information, and, secondarily, the ‘‘cost’’ of recovery.
- *n-fault-tolerant*: since nodes may be multitasking, it is desirable for the recovery scheme to be n-fault-tolerant, i.e., be able to handle the case where *n* processes in the system fail simultaneously. Such a recovery technique can consequently also handle multiple simultaneous node failures.
- *distributed*: the recovery scheme must not have a single point of failure, i.e., it cannot depend on any single resource.
- *suitable for a variety of applications*: computation-intensive and communication-intensive applications.

1.1. Basic Concepts

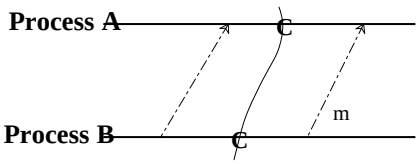
In a system which supports only *non-interacting* processes, checkpointing involves periodically suspending each process, copying the process’ state to stable storage, and then allowing the process to resume normal computation. When a node fails, all information on the node is assumed to be lost. The checkpointed states of all the processes that were on the failed node are restored from stable storage to working nodes. The processes may then resume computation from these

older states. In order to be able to tolerate node failures that occur during checkpointing, the old checkpoint of a process is erased only after the new checkpoint has been completely written to stable storage.

Checkpointing and recovery are more complex when processes are allowed to interact via messages. When one or more processes are rolled back care must be taken to ensure that each pair of processes are *consistent* with each other and hence that the entire system state is *valid*. Specifically, every pair of processes must agree which messages have been sent and which have not[Chan85], such that no messages are lost or duplicated (see Figure 1.1).



a) *Inconsistent checkpoints*



b) *A consistent checkpoint*

Figure 1.1: In a), if A and B roll back to their latest checkpoints their states will be inconsistent. In the first case B expects a message from A which A's rolled back state has already sent (lost message). In the second case B will resend a message which A has already received (duplicate message). In b), the latest checkpoints of A and B are consistent with each other.

A set of checkpoints, one per process being rolled back, which are consistent with each other is called a *recovery line*[Rand78]. Graphically a recovery line can

be represented as a line which connects a set of checkpoints and intersects no communication lines (see Figure 1.1b). If processes are checkpointed independently, ignoring interactions with other processes, the recovery algorithm must find a recovery line amongst the available checkpoints[Wood81]. The major problem with this method (referred to as ‘‘independent checkpointing’’ in this thesis) is the fact that there is no guarantee that a recovery line exists and *domino effect*[Rand78] can occur, where, in the worst case, the entire system must be rolled back to its initial state (see Figure 1.2). Furthermore, many generations of checkpoints for each process have to be maintained on disk. This is necessary since, in the best case, it is desirable to roll back to a recent checkpoint (to minimize lost computations) while in the worst case it may be necessary to roll back to the initial process state.

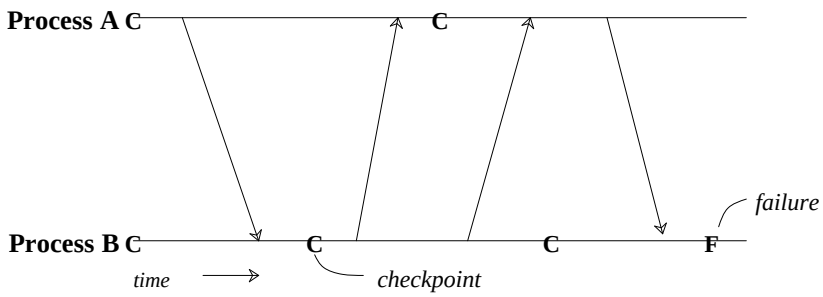


Figure 1.2: Process B fails and is rolled back to its latest checkpoint. Since B will later expect a message from A, A must also be rolled back, requiring B to roll back to a previous checkpoint. In this case, the entire system must roll back to its initial state.

1.2. Application-Specific Techniques

Many error recovery techniques have been proposed which avoid domino effect, but which restrict the actions of the application or require the application to contain special features. The two main areas of such application-specific techniques are Recovery Blocks[Rand75, Russ80, Wood81, Lee84, Kim84, Kim86] and Transaction Processing[Gray81, Nett86, Chin83, Skee81]. Both methods are similar in that they require applications to be structured as atomic actions, or transactions.

Recovery blocks, a language construct, provide facilities to detect failures (acceptance tests), and to reconfigure (one or more alternate algorithms which can be executed in place of the primary algorithm if an error does occur). The application programmer specifies where a recovery block starts (at which point a checkpoint is taken), primary and alternate algorithms which (assuming no software design errors) are functionally interchangeable, and an acceptance test at the end of the block. If the acceptance test fails the previously taken checkpoint is restored and an alternate code block is executed.

Using *conversations*[Rand75] recovery blocks can be applied to interacting processes. Processes that wish to exchange messages must enter a conversation (checkpointing their states) and then may interact freely, but only amongst themselves. Exiting the conversation requires that all participating processes pass their acceptance tests — and no process may proceed until all have done so.

Alone, however, neither recovery blocks nor conversations avoid the worst-case domino effect. One way to avoid the domino effect is for the application programmer to specify sequences of sets of recovery points that form recovery

lines. If an error is detected, the system rolls back to the last programmer-specified recovery line. This approach is not an option if the fault tolerant characteristics of the systems are to be kept hidden from the programmer. Further, in a large general-purpose system, keeping many generations of checkpoints for each process is not a viable option due to required disk space. Hence, recovery schemes that may require indefinite storage of all checkpoints (until the task terminates[Wood81]) are not a practical alternative for our target systems.

In transaction-processing techniques application processes are broken into transactions which always take the system from one consistent state to another. Each transaction terminates by committing or aborting its updates to stable storage. Such a system can ‘‘readily’’ handle recovery by treating hardware failures as aborts. While the system processes performing recovery can be transparent to the user application processes must be structured out of transactions. While some transactions can be executed in parallel *serializability* is required and can significantly degrade the level of concurrency achievable in a system. Transaction-processing recovery techniques are widely used in database systems.

A third technique, N-version (or ‘‘diverse’’) programming[Aviz85], worthy of mention, is an implementation of software masking redundancy. With n-version programming multiple versions of software are independently developed versions and then executed concurrently. A software driver supplies inputs to the different versions and then compares and votes on the outputs. This technique carries a large software design and development cost and has been used in real-time control environments.

As suggested by the name, application-specific techniques are designed for

use with certain types/classes of applications. Since we want to support a wide variety of general-purpose applications on a multicomputer the scope of this thesis is limited to application-transparent techniques. The next section reiterates the classes of application-transparent recovery techniques, briefly describes different schemes that fall in each class, and compares these schemes to the error recovery techniques presented in this thesis.

1.3. Application-Transparent Techniques

There are two major application-transparent approaches to avoiding domino effect in distributed checkpointing and rollback schemes:

I) Message Logging, where messages as well as process states are saved in order to allow the state of a restored process to be ‘‘adjusted’’ so that it is consistent with other processes in the system[Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a].

II) Coordinated Checkpointing, where processes are not checkpointed independently but are, instead, checkpointed in a coordinated way with some or all of the other processes in the system such that if recovery is necessary the restored states are guaranteed to be consistent[Bari83, Tami84, Koo87, Tami89].

1.3.1. Message Logging

Message logging techniques checkpoint process states and (log) interprocess messages onto stable storage. When a process fails and is rolled back, its *message log* is played back to it, so that when the message log has been depleted, the process is in a state consistent with the non-failed processes in the system.

Message logging techniques require application processes to be *deterministic*: given a process state and a sequence of inputs (message log), the process will generate the same outputs. Processes can be checkpointed *independently*, thus minimizing disruptions to the entire system, and a single failed process can be recovered, potentially without interfering with the operation of other processes.

Many papers have been published which use the *message logging* technique as the basis for application-transparent recovery algorithms [Borg83, Powe83, Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a]. These algorithms have proved to be complex and difficult to understand. In this thesis a new message logging algorithm based on [Stro85] is presented. A correctness model derived from the ideas presented in [Lamp78, Chan85] is used as the foundation for a complete set of proofs. This model is also used to thoroughly explain the technical details of asynchronous (optimistic) message logging and to make corrections to previously published algorithms. The new algorithm uses both stable *and* volatile message logs to recover more of the failed process(es)' lost computation. The use of volatile message logs also reduces the number of non-failed processes (orphans) that are forced to roll back in order to recover a consistent system state. This work makes two major contributions: 1) a set of proofs that verify the correctness of this optimized algorithm and 2) a performance analysis of the algorithm using execution-driven simulation.

A complete message logging algorithm is presented. This recovery algorithm is intended for use on a general-purpose distributed system where communication between processes is via asynchronous message passing (e.g. a multicomputer). It is required that the algorithm be distributed, application-transparent, provide *n*-

fault tolerance, and place minimal restrictions on application behavior. The techniques presented in [John87, Sist89] assume that no additional failures will occur during recovery while the optimistic technique presented in [John90a] is a centralized algorithm and hence not appropriate for a scalable multicomputer. In order to support the most general communication model no synchronization (pessimism) [Borg83, Powe83, John87, Jalo89] should be imposed on the message passing protocol, for the purposes of recovery.

A complete set of proofs verify the correctness of the presented asynchronous message logging algorithm. The proofs given in previously presented papers have 1) only shown that their algorithms hold certain properties[Stro85, Stro88, Jalo89], 2) made stringent assumptions[Sist89], or 3) were the basis for a centralized algorithm[John88].

1.3.2. Coordinated Checkpointing

Coordinated checkpointing techniques checkpoint a set of processes together (the entire system or a subset thereof) in such a way that each pair of process states *on stable storage* are consistent with each other [Chan85]. The recovery algorithm is guaranteed to be able to find a recovery line and hence be able to recover a valid system state. Recovery and checkpointing are likely to disrupt more of the system with coordinated checkpointing than with message logging (or, more generally, independent checkpointing), since more than one process is involved. However, there is no need to store message logs and very little bookkeeping information is required.

In error-recovery schemes based on *global* checkpointing[Tami84, Tami87,

Chan85] a consistent ‘‘snapshot’’ of the entire system state is periodically checkpointed and when an error is detected the last checkpointed state is restored. This scheme has very low performance overhead for long batch-oriented applications but is not appropriate for more interactive environments.

With coordinated checkpointing[Bari83, Koo87, Tami89] a consistent set of processes (an *interacting set*) are checkpointed or rolled back together. Disruptions to system operation are reduced compared to global checkpointing schemes and the overhead during normal operation is reduced compared to message logging techniques.

The new low-overhead synchronous and asynchronous coordinated checkpointing error recovery schemes presented in this thesis allow the system to recover from multiple simultaneous failures. Checkpointing involves saving a consistent snapshot of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained in stable storage[Chan85, Tami84]. This allows the system to be *n-fault-tolerant*, keep only one generation of checkpoints on disk, and avoid the domino effect. In the proposed schemes checkpointing and recovery are done at the level of processes with no system-wide central coordination. Many checkpointing and recovery sessions may be active simultaneously. Unrelated sessions do not interfere with each other, while the actions of related ones are properly coordinated. Processes which are not part of the interacting set need not participate in checkpointing/recovery and may continue to do useful work.

The major disadvantage of synchronous coordinated checkpointing is the requirement to block computation of the processes being checkpointed during the

entire checkpoint session. This periodic *synchronous* checkpointing of multiple process states disrupts system operation and can result in significant performance overhead.

The asynchronous coordinated checkpointing scheme achieves the benefits of checkpointing processes independently, but without suffering from the domino effect. The key feature of this scheme is the use of volatile checkpoints *in combination with* the coordinated checkpointing technique to minimize disruption to normal operation due to checkpointing. A volatile checkpoint is simply a copy of the process state in local volatile memory. Checkpointing begins by copying the changed state of a process to local volatile storage, after which the process may resume execution. As discussed in Section 4.6, volatile checkpoints allow much of this local copying to be avoided through the use of a slightly modified virtual memory system. The rest of a checkpointing session involves identification of the set of processes to be checkpointed ‘‘simultaneously’’ and transfer of the volatile checkpoints to stable storage. Thus, a process is suspended only a small fraction of the time it takes to complete a checkpointing session. As in many independent checkpointing techniques, this fraction of time is *independent of network delays and the number of processes that constitute the application*. The cost, compared to synchronous coordinated checkpointing, is an increase in the complexity of the checkpointing algorithm which must ensure that the volatile checkpoints are saved as part of a consistent global state even though the processes being checkpointed have resumed execution.

No message logging and minimal bookkeeping information is required. Also, no hardware support (e.g. clock synchronization) is assumed. Since processes are

suspended from execution only a small, locally determined, amount of time, our scheme should scale well and support both multi-tasking nodes and communication-intensive applications with less overhead than previously published coordinated checkpointing techniques.

Two recently published algorithms [Bhar88, Wang92] have stated ‘‘independent checkpointing’’ as a primary goal and advantage. These papers presented algorithms that, in order to avoid any synchronization penalty, do not guarantee forward progress of the recovery line maintained on stable storage. We argue that coordinated checkpointing is more general and can perform better than independent checkpointing — during recovery *and*, as shown in simulation results, during normal operation in a system without failures. Our claim is not necessarily true if the ‘‘independent’’ checkpointing algorithm actually relies on some form of clock synchronization (low-level coordination) to limit its overhead as in the simulation results given in[Wang92].

1.4. Scope of the Thesis

This thesis examines techniques to provide transparent recovery from hardware failures in general purpose multicomputers. Both of the major classes of transparent recovery, message logging and coordinated checkpointing, are considered in detail. A system model based on[Lamp78] is described. This model provides the framework for proving the correctness of both message logging and coordinated checkpointing algorithms.

A new asynchronous message logging algorithm is presented. This algorithm

is based upon a previously published algorithm, but includes corrections, an optimization and several important details. The major contributions of this algorithm are 1) the accompanying proof of correctness and 2) the performance analysis. Previously published proofs were only partial proofs or made simplifying assumptions that rendered the algorithm impractical.

Two new coordinated checkpointing algorithms are also presented. The first is a synchronous algorithm that checkpoints and recovers interacting sets of processes. The correctness of this algorithm is shown. The basic idea behind this synchronous coordinated checkpointing algorithm is not new but many important details of the presented algorithm have not previously been addressed. This algorithm is then extended to be asynchronous — unlike in synchronous coordinated checkpointing, processes do not have to be suspended for the duration of their checkpoint session. Instead, normal execution is resumed as soon as possible after taking a volatile checkpoint, such that the process is suspended a fraction of time that is independent of network delays and the number of processes in the application. The cost is more complex send and receive primitives during the first phase of the checkpoint session. No asynchronous coordinated checkpointing algorithm has been previously presented.

Performance analysis is an important part of the work in the thesis. No attempt has previously been published to compare, through simulation or implementation, the overhead of the schemes belonging to the two major classes of transparent error recovery. Also, no such comparison has been made between schemes within the same class. Further, all previously published performance analyses and implementations of the techniques in these two classes (known to this

author) have been restricted to ethernet-based multiprocessors and have not considered general-purpose, scalable, multicomputers. This thesis contributes a first-order performance analysis of the schemes within and between the two classes. Most of the performance analysis, however, is performed on a simulation testbed.

The execution-driven simulator was first used to analyze the performance of the message logging techniques on a failure-free system. Real parallel applications were used along with low-overhead send and receive primitives similar to those used on current multicomputers. In this environment the message handling overhead required by asynchronous message logging algorithms has a substantial impact on application execution time. Two simple optimizations we proposed (elimination of the orphan detection filter and postponement of the dependency vector update at the receiver) offered significant improvement, while sending dependency vector differences often increased overhead. However, the message logging algorithm still scales poorly for three of the six applications and may require extensive hardware support to eliminate this trend.

We also showed that our simple optimization to reduce the likelihood of orphan processes occurring (by recording *RSN*s at the senders) can be beneficial to recovery performance due to the probability that, if one orphan process occurs, continued message transmission by this orphan is likely to result in many orphans. Future work involves validating this observation through performance analysis of the recovery procedure, comparing the message logging technique to other application-transparent error recovery schemes, examining possible hardware support, and working on methods to reduce the $O(n)$ dependency vector

requirement without losing the benefits of the message logging algorithm.

Secondly, simulations we ran using the same applications show that asynchronous coordinated checkpointing can be an appropriate error recovery technique for high-performance multicomputers in which fine-grain parallelism is exploited. Additionally, it is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory. Hence, the disruption to normal processing will last, at most, tens of microseconds. The amount of data that has to be sent to stable storage is minimized and consists only of pages that have been modified since the last checkpoint and are resident in local memory when the checkpointing session is initiated.

Using general assumptions our simulations showed that, even *without* this optimization, the amount of overhead incurred by asynchronous coordinated checkpointing is nearly independent of multicomputer scale (with constant problem size). Asynchronous coordinated checkpointing performs at least as well as synchronous coordinated checkpointing and better when the checkpoint interval is decreased. This is because the overhead of the asynchronous algorithm is independent of network delays and the size of the interacted set (or the number of processes that constitute the application), unlike the synchronous algorithm. Furthermore, for communication-intensive applications (or those with large volatile checkpoints) the coordination of checkpointing processes is *desirable* in order to overlap checkpointing overhead; in several of our simulations independent checkpointing (without message logging) performed worse than even *fully*

synchronous coordinated checkpointing. Lastly, with coordinated checkpointing recovery is guaranteed to be free of the domino effect, while independent checkpointing techniques (without message logging) do not not guarantee progress of the recovery line.

1.5. Thesis Organization

Many distributed application-transparent error recovery techniques have been proposed [Rand78, Russ80, Wood81, Bari83, Hoss83, Powe83, Borg83, Tami84, Stro85, Koo87, John87, Stro88, Bhar88, Sist89, Jalo89, Tong92, Wang92, Elno92a] — Table 1.1 characterizes representative examples of the major approaches to application-transparent error-recovery. This table shows, for each listed recovery scheme:

1) Assumptions about the system model (e.g. does it require reliable communication and/or fail-stop nodes?).
2) The overhead incurred during normal operation (e.g. are messages logged?, is bookkeeping information transmitted with each message?).
3) The overhead incurred during checkpointing sessions (e.g. do checkpointing sessions involve multiple processes or are they independent?).
4) Recovery features (e.g. is the algorithm n-fault-tolerant?, how many processes have to be rolled back?).
5) Does the algorithm suffer from domino effect? (i.e. can it guarantee progress of a recovery line?).

Contrary to what is stated in the table (in the third row, where the second column is

Recovery Scheme	Assumes	Overhead During Normal Operation	Overhead During Checkpointing	Recovery Features
				Domino Effect?
Synchronous Message Logging (J&Z)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged in volatile memory. ●Process is suspended if a 2nd message is sent before the 1st is logged.	Suspend process for asynch chkpt. (independent)	●Single process failures only.
				No.
Asynchronous Message Logging (S&Y,F&T††)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged on stable storage.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore failed processes and any orphans.
				May roll processes back > 1 times.
"Chase" Protocols (Wood)	Reliable comm, fail-stop nodes†	Maintain prop-list and PRI-list ●Each message: new PRI-list entries.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore recovery line.
				●Yes — no guarantee recovery line will progress.
Independent Checkpointing (W&F)	Reliable comm, fail-stop nodes	●Each message: add'l ovrhd incl. O/S processing; logged in volatile memory.	Suspend process for asynch chkpt. (independent)	N-fault-tolerant; Restore local recovery line.
				●Yes — no guarantee recovery line will progress.
Global Checkpointing (C&L,T&S)	Fail-stop nodes	No.	●Suspend system for synch chkpt.	N-fault-tolerant; Restore entire system.
				No.
Synchronous Coordinated Checkpointing (K&T,T&F††)	Fail-stop nodes (K&T: Rel. Comm.)	Keep track of Buddies list.	●Suspend interacting set for synch chkpt.	N-fault-tolerant; Restore failed processes and their interacting sets.
				No.
Asynchronous Coordinated Checkpointing (T&F††)	Fail-stop nodes	Keep track of Buddies list.	Suspend process for asynch chkpt. (interacting set)	N-fault-tolerant Restore failed processes and their interacting sets.
				No.

Table 1.1: Main characteristics of application-transparent error recovery schemes. Major drawbacks are indicated with a (●).

marked with a †), Wood[Wood81] does not actually require that nodes be fail-stop but instead assumes that errors are detected by the applications themselves through the use of acceptance tests; the algorithm is, however, compatible with either assumption. Work presented in this thesis is highlighted in the table with (††).

In Chapter 2, the system model used in this thesis is described. A correctness model based on the concepts in[Lamp78] is presented. *Correct* execution is defined for both systems without failures and systems where failures can occur. A brief description as to how message logging and coordinated checkpointing techniques ensure correct execution of application processes, in the event of a hardware failure, is given. The details of the hardware model of the target multicomputer are then described in Section 2.2 and the error detection used is discussed in Section 2.4.

Message logging as a method for providing application-transparent error recovery is examined in detail in Chapter 3. Previous work is described in Section 3.1. The data structures which must be maintained for each process when using asynchronous message logging are enumerated in Section 3.2. The basic approach of asynchronous message logging algorithm, as originally proposed in[Stro85] is described in Section 3.3. Section 3.4 discusses, in detail, how the correctness model presented in Section 2.1 applies to asynchronous message logging algorithms. Our new asynchronous message logging algorithm, which has an optimization to reduce the number of orphans that occur as a result of a failure, provides a correction over previous algorithms, and provides essential details that have been previously omitted, is presented in Section 3.5. Normal operation and checkpointing is detailed in Section 3.5.1 and the recovery algorithm in

Section 3.5.2. The proof of the algorithm's correctness is given in Section 3.6. Several important issues are further discussed in Section 3.7 and conclusions are drawn in Section 3.8.

Chapter 4 starts out describing previous work in coordinated checkpointing techniques. A simple algorithm to record a global state, which is a reworded version of the snapshot algorithm in[Chan85] is briefly discussed in Section 4.1.1. The global checkpointing algorithm presented in[Tami84], which is a complete error recovery technique that records a consistent global state on stable storage, is discussed in Section 4.1.2. In Section 4.1.4 the global state recording algorithm is extended to record the states of a subset of the system (an interacting set of processes) as a way of *maintaining* a consistent global state on stable storage. Sections 4.1.4 and 4.1.5 prove the correctness of checkpointing and recovering (respectively) an interacting set of process to and from stable storage.

Section 4.2 describes that basic structure of an error recovery scheme based on the previous, proven, interacting set algorithms. Section 4.2.1 provides pseudo-code and some important high-level details on the coordinating of the checkpoint and recovery algorithms. Sections 4.2.1.1 and 4.2.1.2 show that non-interfering checkpoint/recovery sessions and coordinator conflicts in interfering checkpoint/recovery sessions are correctly handled, respectively. In Section 4.3 the complete synchronous coordinated checkpointing algorithm is presented including discussion of support for both the dynamic virtual circuit and packet-switching communication models. Section 4.4 provides details on recovering from communication errors (Section 4.4.1) and node failures (Section 4.4.2). Handling failures during checkpoint sessions is shown to not be a problem in Section 4.4.3.

A new asynchronous coordinated checkpointing algorithm is presented in Section 4.5. For ease of explanation the correctness of this algorithm is first described for the case where there is one active coordinator (process) and one active checkpointing session in the system (Section 4.5.1). This case is then extended to *n* coordinators and 1 checkpointing session (Section 4.5.2), and finally, *n* coordinators and *n* checkpointing sessions (Section 4.5.3). Techniques for minimizing data movement during checkpointing by using an enhanced virtual memory system are outlined in Section 4.6.

In Chapter 5 we present DERT (Distributed Error Recovery Testbed) a testbed for execution-driven simulation of error recovery techniques. The motivation behind the testbed design is given in Section 5.2. Section 5.3 describes the testbed and Section 5.4 the recovery schemes simulated on the testbed. Experiences getting applications to run on the testbed are highlighted in Section 5.5 followed by experiences simulating the recovery schemes in Section 5.6. Emphasis is placed on problems encountered and examples of the simulator's flexibility and realism. Section 5.7 discusses simulator performance. Previous work is briefly described in Section 5.8 and Section 5.9 concludes the chapter.

Our performance evaluation of application-transparent distributed error recovery techniques is the subject of Chapter 6. In Section 6.1 we first characterize the overhead incurred when *any* checkpointing/rollback scheme is used to provide fault tolerance for a scalable multicomputer system (Section 6.1.1) and then characterize and compare the overhead incurred by several different message logging (Section 6.1.2) and coordinated checkpointing techniques (Section 6.1.3). In Section 6.2 we use DERT to evaluate the performance of asynchronous message

logging and some optimizations (Section 6.2.1), and then to evaluate synchronous and asynchronous coordinated checkpointing and an ‘‘optimistic’’ independent checkpointing algorithm (Section 6.2.2). The primary focus of this evaluation is on the impact of the checkpointing algorithms on application execution time relative to the execution time of the application without any checkpointing, under error-free operation. Chapter 7 concludes the dissertation.