Proceedings of the
30th Annual Simulation Symposium,
Atlanta, GA, pp. 4-13, April 1997.

EXECUTION-DRIVEN SIMULATION OF ERROR RECOVERY TECHNIQUES FOR MULTICOMPUTERS

Tiffany M. Frazier † *and Yuval Tamir*
Computer Science Department
UCLA
Los Angeles, California 90095
U.S.A.

Abstract

DERT (Distributed Error Recovery Testbed) is a testbed for simulation and performance evaluation of several classes of application-transparent distributed error recovery schemes. DERT is built on top of an event-driven, message-passing, object-oriented, multithreaded simulation kernel. Actual compiled distributed applications are instrumented for data collection and executed on the simulated multicomputer. Checkpointing is implemented in full detail, including associated overhead per message, additional messages, and changes to the memory system. DERT allows easy modification of a wide variety of system parameters, thus offering a level of flexibility not easily achieved by experimentation on a particular real machine. This paper describes the design, functionality, and performance of DERT. The main problems encountered in DERT's development are discussed, as well as examples of its use in evaluating recovery schemes.

I. Introduction

For many important applications of distributed systems and of multicomputers the reliability requirements can only be met using fault tolerance techniques. The mechanism for recovering a valid system state following error detection is key to implementing fault tolerance. The main desirable features of the error recovery mechanism are: 1) low overhead during normal operation, even for communication-intensive applications, and 2) application transparency so as not to increase the complexity of the application software. There are two main classes of distributed error recovery schemes that are application transparent and prevent unconstrained rollback propagation: *coordinated checkpointing* [1, 25, 15, 26] and *message logging* [21, 11, 22, 7]. Our research [8] has involved developing new recovery algorithms that fall into both classes. An essential facet of this work has been the performance analysis of these and other distributed recovery schemes.

Realistic performance evaluation of distributed checkpointing and rollback techniques is difficult due to the numerous factors that must be taken into account, involving detailed characterization of the target system and of the applications. Thus, most performance analyses of distributed recovery techniques have been limited to greatly simplified analytical models. Realistic experimental performance evaluations have appeared only recently [12, 3, 28, 6, 14], but with almost no analysis in the context of multicomputer systems — where normal communication latency is small (microseconds) and many of the applications are communication-intensive.

DERT was developed to measure the performance impact of several different error recovery techniques for multicomputers. We wanted to evaluate changes in the overhead as a function of various system parameters and application characteristics. To facilitate research on error recovery schemes it was particularly important to be able to evaluate a variety of related optimizations, including low-level features, of the target architecture and operating system. Thus, simulation accuracy and flexibility were the primary design goals of DERT.

DERT is implemented on top of a multi-threaded simulation kernel, called *Simon* [23]. It runs actual distributed applications whose assembly code has been instrumented to 1) count the number of assembly instructions executed and 2) take checkpoints at precise intervals. We have successfully used DERT to evaluate a variety of recovery schemes with coarse-grained and fine-grained applications. DERT has proven to be a powerful and flexible tool for evaluating the overhead and scalability of recovery schemes. This paper relays our experiences in developing and using this simulator.

The target system is described in Section II. Section III describes the testbed implementation, Section IV the recovery schemes simulated on the testbed, and Section V the benchmarks. Section VI relays experiences simulating the recovery schemes and Section VII discusses simulator performance. Previous work is the topic of Section VIII.

II. The Target System

The target (simulated) system is a scalable multicomputer consisting of nodes which communicate via messages over point-to-point links. Each node includes a processor, local memory, and a communication coprocessor. Nodes operate

† Currently with The MITRE Corporation.

asynchronously and messages may have to pass through intermediate nodes on the way to their destinations. The system is used for general-purpose applications with no hard real-time constraints. Errors can occur at any time due to *hardware* faults in the nodes or links.

Error recovery schemes require ‘‘safe’’ storage of some of the system state so that it can be accessed during recovery despite component failures. The target system is assumed to include such *stable storage* [17], implemented on ‘‘reliable disks’’ (e.g. mirrored disks), where checkpoints can be safely maintained. Some of the nodes in the system, which we call *disk nodes*, are connected to such reliable disks. We assume that the failure of a disk or a disk node causes a *crash* (i.e., an unrecoverable error).

The *state* of a process is the contents of the memory and registers used by the process. This includes some system tables (e.g. the list of virtual circuits currently established to and from the process).

Since each node can be time-shared between multiple processes, it may have to participate in multiple simultaneous checkpointing and recovery sessions. Hence, checkpointing and recovery are not implemented as part of the kernel; instead, a *handler* process is spawned to perform these functions. The handler can suspend an application process, manipulate its state, and allow it to resume normal operation.

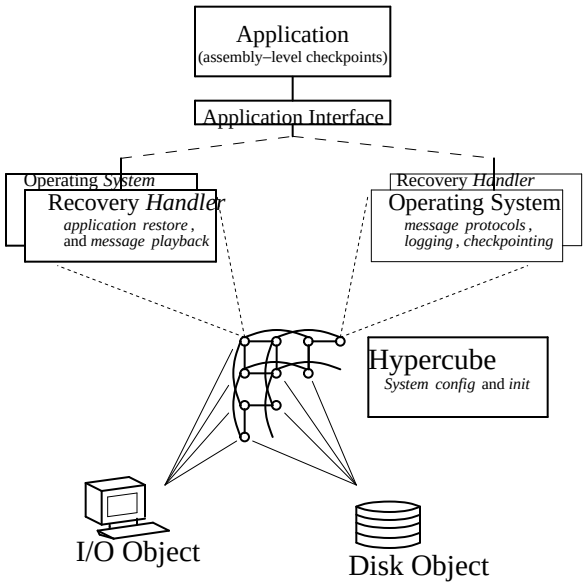

Figure 1: The simulation testbed.

III. The Implementation of DERT

The key motivation for DERT was to allow accurate evaluation of the performance impact of various recovery schemes. Failures are assumed to be relatively rare, hence the performance impact of the recovery schemes is almost entirely due to the overhead during normal operation. Thus, a key requirement from DERT was the ability to accurately measure the impact of any extra bookkeeping, logging, and state saving required for recovery. These factors are dependent on system characteristics, such as the relative costs of computation versus communication, and application characteristics, such as communication frequency and the rate at which the application state changes. Our studies required the ability to vary these system and application characteristics in order to evaluate the suitability of different types of recovery schemes for different types of systems and applications. The only way to meet these requirements was to use *execution-driven* simulation.

DERT is implemented on top of an object-oriented multi-threaded, event-driven simulator called *Simon* [23]. *Simon* manages each object as a separate thread, implemented as a Modula-2 coroutine. The simulation kernel, which manages the event queue, is also a Modula-2 coroutine. Simon provides routines for creating objects, sending messages between objects, and debugging. Messages are delivered to and from objects through input and output ports. These ports are interconnected via unidirectional ‘‘links’’ at the beginning of the simulation.

DERT has been run only on Sun SPARC systems under SunOS. It has executed distributed applications written in C and Modula-2. DERT includes fully functional implementations of two major application-transparent error recovery schemes: asynchronous message logging and asynchronous coordinated checkpointing. It provides several optimizations of these schemes. DERT also includes an independent checkpointing scheme, used to evaluate the cost of periodic checkpointing of process states without the additional overhead for maintaining the information needed to recover consistent process states following node failure. DERT consists of 20,000 lines of code written in Modula-2, C, and assembly.

A. The Multicomputer

As previously discussed, the target system being simulated is a scalable multicomputer. We chose to simulate a system with up to 256 nodes, interconnected in a hypercube topology. Each of the nodes in the hypercube is simulated as two Simon objects: the application process and the operating system. The operating system object (see subsection C) handles checkpointing and recovery activities. The nodes being simulated are not multitasking and there is no direct context switching between the application and the operating system. The testbed was greatly simplified by simulating events in the application process and the OS as executing ‘‘in parallel’’ on the two Simon objects.

Interprocessor communication in the simulated multicomputer is performed via message exchanges. Message *send* operations are nonblocking. Messages become part of the receiving node state once they are delivered by the network. The receiving application

process can access an arriving message only after executing a blocking *receive* operation. It is assumed that all application messages are *anticipated*, i.e., a message is available to the receiving process only after it executes a *receive*. The restriction to anticipated application messages simplifies the simulation. As described in the next subsection regarding *checkpoint* messages, it is possible to relax this restriction.

The network operates with a store-and-forward datagram protocol using dimension-order routing. When link conflicts are encountered, messages are stored at that node until the link is available. A link bandwidth of $20\,Mbytes/sec$ is assumed.

B. Instrumentation of the Applications

Simulating application execution with fine-grain accuracy requires extensive instrumentation of the application code at the assembly code level. There are two purposes for this instrumentation: A) account for application execution time and B) provide a mechanism for initiating checkpointing based on elapsed simulated time. The application is compiled down to assembly code and then passed through a parser that inserts at the end of each basic block assembly instructions that count the number of executed machine instructions. This straightforward dynamic instruction count accumulation is used in many other tools, such as PIXIE [24]. In the simulations reported in this paper, most instructions are assumed to execute in $100\,nsec$, but floating point instructions execute in $300\,nsec$ or $900\,nsec$ for add/subtract or multiply/divide, respectively.

In many checkpointing schemes, checkpointing is initiated when the time since the last checkpoint exceeds a preset limit. To support this mechanism in DERT, the application code is instrumented so that at the beginning of each basic block it is determined whether it is time to take a checkpoint. This is done by comparing the accumulated running time since the previous checkpoint of this process to the checkpoint interval, which is a parameter to the simulator. If the running time is greater than or equal to the checkpointing interval, the application process 1) sends a checkpoint message to the operating system object and 2) executes a blocking receive to wait for a message from the operating system indicating that the application can resume execution.

The simulated time of a process is synchronized with the global simulated time before a message is sent or received and before a checkpoint is taken. If the simulation only had to deal with *anticipated* messages, this would be sufficient to ensure accurate simulations. However, while the simulator supports only anticipated *application* messages, the restriction to anticipated messages is violated by system messages that are supposed to initiate checkpointing or recovery and cannot be anticipated by the application. This can lead to a problem with processes that may execute for an extended period of time without sending or receiving a message. Specifically, in the simulation such a process may execute to a point far ahead of when in the real system an unanticipated ‘‘initiate checkpoint’’ message would have been received. This simulation inaccuracy may affect the number of checkpoints taken, the size of the checkpoints, and the application execution time. We solve this problem by not allowing any process to get ahead of the global simulation time by more than a fixed *TIME_SLICE*. At the beginning of every basic block, in addition to checking whether the checkpoint timer has expired, the augmented code also checks that no more than *TIME_SLICE* time units have elapsed since the process was last suspended. If the test fails, the process is forced to resynchronize with the global simulation clock (control is passed to the simulator kernel).

C. The Operating System

On each node there is an application process and an operating system process, which execute concurrently. The simulator does not support multiple application processes on a node. The *application interface* between the application process (object) and the operating system process (object) is through three routines: Send, Receive, and Checkpoint. The operating system handles the network interface, checkpointing and recovery tasks, and the maintenance of bookkeeping information. An execution time (cost), expressed in terms of an approximate machine instruction count, is associated with every operating system routine (e.g. send, receive, memory-to-memory copy, table lookup). This allows the execution time of operating system tasks to be taken into account in the simulation.

Each application process is checkpointed prior to execution. After this initial checkpoint, only the modified state of the process is checkpointed. The checkpoint-detection code in each basic block checks the checkpoint timer. If it has expired, a message is sent to the operating system. Upon receipt of this message the operating system enters checkpoint mode and initiates a checkpoint session on behalf of the application process. The operating system 1) takes a volatile checkpoint of the process in local memory 2) wakes up the process as soon as possible considering the size of the checkpoint and the specific checkpointing algorithm and 3) sends the volatile checkpoint to stable storage. With message logging, the application resumes execution immediately after the volatile checkpoint has completed. The checkpoint state consists of: the stack, statically allocated memory containing the process’ global variables, and heap from which memory is allocated on demand. Some bookkeeping information is also checkpointed.

The size of the checkpoint state significantly impacts the overhead of the checkpointing algorithms. Thus, for all techniques, only the pages modified since the previous checkpoint are copied during a volatile checkpoint. In a real system, hardware support may be used to keep track of which pages are ‘‘dirty-since-

checkpoint.'' In the simulation, the operating system object computes the checksum of every page when a checkpoint is taken. At the beginning of checkpointing, the operating system object identifies modified pages by comparing the current checksum of every page to its checksum from the previous checkpoint.

Recovery sessions are initiated by a simulation message whose receipt signifies that a hardware (node) failure has occurred. The application state and all volatile memory is assumed to be lost and the operating system enters recovery mode to initiate a recovery session and restore the application's state. At some point during the recovery session the operating system wakes up the application process to complete the recovery or to begin re-execution from its consistent restored state.

D. The Host Processor (I/O Object) and Stable

Applications must be structured as an I/O object and some number of application processes. The I/O object is responsible for initializing the hypercube and calling the testbed procedures that create all the required objects. The I/O object then acts as the ''outside world'' (host processor) by generating all inputs and collecting all outputs. There is no restriction as to how this is done.

Our simulations assume that disks and disk nodes are reliable and can thus be used directly to implement stable storage [17]. By default there is one disk per application process. Each disk has a bandwidth of three megabytes per second. A single disk object is used to simulate *m* disks. The disk object's key function is to log messages and checkpoints. Traffic to the disks does not use the network used for internode communication.

IV. The Recovery Schemes Simulated

When one or more processes are rolled back care must be taken to ensure that each pair of processes are *consistent* and hence that the system state is *valid*; every pair of processes must agree which messages have been sent and which have not [5], such that no messages are lost or duplicated. A set of checkpoints, one per process being rolled back, which are consistent with each other is called a *recovery line* [19]. If processes checkpoint independently, ignoring interactions with one another, the recovery algorithm must find a recovery line amongst the available checkpoints [29]. The problem with ''independent checkpointing'' is that there is no guarantee a recovery line exists and *domino effect* [19] can occur, causing the system to roll back to its initial state.

Two major application-transparent distributed checkpointing and rollback schemes avoid domino effect: message logging [21, 11, 22, 13, 20, 10, 7] and coordinated checkpointing [1, 25, 15, 26].

A. Message Logging

Message logging techniques checkpoint process states and (log) interprocess messages. When a process fails and is rolled back, its *message log* is played back to it, so that when the log has been depleted, the process is in a state consistent with the non-failed processes in the system. Application processes are required to be *deterministic*: given a process state and an ordered message log, the process will generate the same outputs. Processes are checkpointed independently and a single failed process can be recovered, potentially without interfering with the operation of other processes.

With *optimistic* (or *asynchronous*) message logging techniques [21] logging proceeds asynchronous of process execution. Thus, when a process fails some messages might not yet be logged. To achieve a consistent system state other processes (*orphans*) may have to roll back. This approach is ''optimistic'' because it does not incur synchronization overhead during normal operation in order to avoid orphans.

Ensuring correctness despite the loss of interprocess messages requires *dependency tracking*. Each process maintains an *RSN* (receive sequence number), incremented each time a message is received, which indicates the start of the next state interval in that process. Each process *x* also maintains an O(n) dependency vector indicating the most recent state interval of every process *y* upon which *x depends* [16, 21]. The dependency vector is appended to outgoing messages which receivers use to update their dependency vectors. A process, or message, is an orphan if it depends on a state intervals that is known to be lost (rolled back). Each process maintains an incarnation start table (*IST*) which contains information identifying which state intervals are thought to be *live* (not lost).

When a process *q* fails its earliest checkpoint is restored and the message log is replayed until an orphan message or the end of the log is reached. *q* then broadcasts its current state interval number to all processes in the system — informing them that any larger (later) state intervals have been lost. Upon receiving this broadcast a process updates its *IST* and determines if it is an orphan. Orphan processes are rolled back and orphan messages are discarded. For details see [21].

In addition to asynchronous message logging, an ''optimal'' independent checkpointing algorithm has also been implemented. In this algorithm processes are checkpointed independently but no additional overhead (e.g. dependency tracking) is simulated.

B. Coordinated Checkpointing

Coordinated checkpointing techniques checkpoint a set of processes together (the entire system or a subset) such that each pair of process states *on stable storage* are consistent with each other [5]. The recovery algorithm is therefore guaranteed to be able to find a recovery line. There is no need to store message logs and little bookkeeping information is required.

Global checkpointing [25] checkpoints and recovers the entire system. With coordinated checkpointing of interacting sets [1, 15, 26] a consistent set of one or more processes are checkpointed or rolled back together. An interacting set of processes is defined to be the set of processes which have communicated directly or indirectly since their last checkpoint; if p is a member of an interacting set, then any process q that has communicated with p since q's last checkpoint is also a member. Each process maintains a *Buddies* list of the processes with whom direct communication has taken place since its last checkpoint. Checkpointing overhead is reduced compared to global checkpointing and overhead during normal operation is reduced compared to message logging. During recovery all processes in the interacting set(s) containing the failed process(es) must be rolled back. These algorithms are synchronous — process execution is not resumed until the checkpointing session has completed.

Asynchronous coordinated checkpointing [27, 8], achieves the benefits of checkpointing processes independently, but avoids the domino effect. Checkpointing a process begins by taking a volatile checkpoint (copying the changed state of a process to local volatile storage), after which the process resumes execution. The rest of the checkpointing session involves identifying the interacting set and transfering the volatile checkpoints to stable storage. As with independent checkpointing, the length of time a process is suspended is independent of network delays and the number of processes that constitute the application. The cost 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.

C. Overhead Breakdown

The performance overhead of the error recovery schemes can be broken into three parts: 1) overhead incurred during normal operation; 2) the cost of checkpointing process states; and 3) the cost of recovery when failures occur. In the testbed we focus on (1) and (2) since this is the overhead that impacts the system the majority of the time. Only the functionality of recovery, not the performance, has been tested.

(Asynchronous) Message Logging —

During normal operation the following is required:

- Extra processing by the send and receive primitives (to track dependencies).
- Periodic broadcast of logging status.
- Periodic reclamation of checkpoints and message logs.
- Local memory for bookkeeping data structures and volatile message logs.
- Stable storage for the message log including the dependency vectors.
- Network bandwidth for dependency vectors and for copies of messages (and their dependency vectors) being sent to remote stable storage.
- Disk bandwidth for messages and their dependency vectors.
- Delay in committing outputs to the external world.

This overhead is likely to have a more impact on applications with fine-grained parallelism than on applications with coarse-grained parallelism.

During checkpointing, network and disk bandwidth are needed to transmit local bookkeeping data structures and the process' current state, and stable storage is required for the checkpoint state and data structures. This overhead is dependent on the checkpoint interval and the size of the process state.

(Asynchronous) Coordinated Checkpointing —

During normal operation the overhead incurred is:

- Local memory for the *Buddies* list and processing to update it.
- Delay in committing outputs to the external world.

Also, compared to a system that has no process recovery but uses check bits to send reliable messages — *less* OS processing and network bandwidth are required to transmit messages, if message transmission error checking is incorporated into coordinated checkpointing as described in [27].

During checkpointing the following is required:

- The process is suspended the length of time it takes to copy the current state into local volatile memory and to queue outgoing *Checkpoint* messages.
- Significant (''background'') processing by the checkpoint handlers.
- Network and disk bandwidth to transmit checkpoints and message queues; additional network bandwidth for recovery scheme messages.
- Stable storage for checkpoints: up to two per process.
- Coordinated checkpointing results in ''bursty'' network and disk traffic.

Message logging techniques incur more overhead during normal operation than do coordinated checkpointing techniques and less during checkpointing sessions. Thus, detailed characterization of distributed applications is required to accurately compare the two techniques. All overhead itemized above is simulated by the testbed at the level of assembly instrucions.

V. The Benchmarks

Six applications have been incorporated into the simulator: fast fourier transform (fft), router, placement, matrix QR factorization, olfactory and extract [9, 4]. Some of the characteristics of these applications, determined by examining simulation output, are listed in

#Nodes	4	8	16	32	64	128	256
	Total #assembly instructions (in millions) executed by the application						
Extract	2219	3001	3957	-	-	-	-
OLF	-	359	486	737	1245	2259	4303
FFT	-	-	2017	2081	2160	2260	2422
QR	819	819	820	821	-	-	-
Place	349	468	692	-	-	-	-
Router	254	299	351	-	-	-	-
	Total #msgs sent & total message volume (MBytes)						
Ext: #msgs	1299	2969	7941	-	-	-	-
volume	12.7	25.3	63.5	-	-	-	-
OLF: #msgs	-	2807	7483	18.7K	44.9K	105K	239K
volume	-	11.8	25.3	52.2	106.1	214.3	432.1
FFT: #msgs	-	-	32.0K	80.1K	192K	448K	1.02M
volume	-	-	65.6	82.0	98.4	114.8	131.1
QR: #msgs	55.3K	55.3K	55.4K	55.4K	-	-	-
volume	58.6	58.6	58.6	58.6	-	-	-
Place: #msgs	735	16.5K	131K	-	-	-	-
volume	1.8	3.4	7.9	-	-	-	-
Rtr: #msgs	127K	326K	715K	-	-	-	-
volume	7.2	16.3	15.5	-	-	-	-
	Avg. #pages modified after 146 million instrs (for Extract) and 46 million instrs (for all others).						
Extract	1.85K	1.62K	1.42K	-	-	-	-
OLF	-	16.34	19.70	22.14	24.41	25.68	26.54
FFT	-	-	2.31	2.56	2.97	3.55	4.23
QR	1.15	1.30	1.58	2.07	-	-	-
Place	58.63	59.18	57.40	-	-	-	-
Router	30.00	37.45	48.99	-	-	-	-

Table 1: Application statistics: communication frequency (avg. no. of assembly instructions executed between sends), no. of messages sent, message volume, and typical checkpoint state size (the number of 1Kbyte *modified* pages checkpointed by an independent checkpointing algorithm using a particular checkpoint interval).

Table 1. fft is run over 2^{12} points. Matrix QR is run for four successive matrices (64x64, 96x96, 128x128, 160x160). Router (with input of 72 nets) and placement (with 183 cells) are both VLSI CAD tools that use simulated annealing algorithms. Extract is a VLSI CAD tool that performs circuit extraction (on a CIF file with 66K rectangles). Olfactory is a hypercube implementation of an anatomically realistic model of a real neural network (the piriform cortex).

All of the applications, with the exception of QR, become more communication-intensive as they scale (with constant problem size). Extract has the largest state (e.g. 1.4 to 2.9 MBytes of modified state checkpointed per process) and is the least communication-intensive (sending a message every .5 to 1.7 million instructions). This is in contrast to router where, with 16 processes, each process sends a message, on average, every 492 instructions. We did not know, prior to executing the applications on the testbed what sort of execution profile they would exhibit. Thus, the simulator is useful for obtaining such information (e.g. for use in an analytical model).

In developing the testbed a significant problem was encountered with the applications in the form of intraprocess and interprocess nondeterminism. Message logging algorithms require applications to be deterministic in their actions — therefore, all *intra*process nondeterminism had to be removed. For example, random number seeds had to be saved and restored with the checkpoint.

More difficult was a specific case of *inter*process nondeterminism caused by the interaction of 1) the use (by *Placement*) of a ring broadcast to update data on the processors and 2) the use of periodic checkpointing. The simulated annealing algorithm used by placement chooses master-slave pairs in every annealing iteration. Without checkpointing, an application, using the ring broadcast, can be run multiple times, with the same inputs, and it will always give the same outputs; the same master-slave pairs will be chosen in the same order. With checkpointing, however, it is possible for a *different* process to exit a particular ring broadcast first and then decide to become a master. Thus the chosen master-slave pairs are different than in the simulation without checkpointing — altering the application's execution and resulting in significant changes to application execution time and application results. This made it impossible to compare the execution times of the application with and without checkpointing.

There were two possible solutions. We could run multiple simulations with checkpointing, measure confidence intervals, and determine an average checkpointing overhead. Instead, we chose to replace the ring broadcast with a broadcast that guarantees the order in which processes will exit the broadcast. It should be noted that this problem is not peculiar to the simulator; the same problem would exist if the applications were running directly on a multicomputer.

VI. Experiences with the Recovery Schemes

We are primarily interested in the overhead incurred by error recovery schemes during normal operation and checkpointing. Therefore, for the simulations which do not measure the overhead of failure recovery, much of the code that performs the actual checkpointing and rollback functions is ''turned-off'' while still incorporating the overhead incurred by these functions into the simulations. This conserves disk, memory, and swap space, increasing the degree to which the applications can be scaled. With these optimizations Olfactory and fft could be scaled to 256 nodes when executing on a Sun-4 with 32 *MBytes* of physical memory and 100 *MBytes* of swap space. The other applications had been hard-coded to scale to 16 or 32 nodes.

A. Message Logging

The flexibility of the testbed allowed us to examine recovery scheme overhead at a very low level. By measuring the processing overhead of individual recovery components, we can 1) determine the

	Original	Orph_roll	DV_diffs	Optimal	No_Chkpt
Sender:					
DV diff	-	-	$N*33$	-	-
Tag copy	$N*1.5$	$N*1.5$	$X*1.5$	-	-
SEND	10	10	10	10	10
Save DV	-	-	$N*1.5$	-	-
Setup to -	129	129	129	-	-
Save msg	$(size+N)*1.5$	$(size+N)*1.5$	$(size+X)*1.5$	-	-
Link Xfer:	$(size+N+4)/2$	$(size+N+4)/2$	$(size+X+4)/2$	$(size+N+4)/2$	$size/2$
Interm. Node:	10	10	10	10	10
Receiver:					
RECV	10	10	10	10	10
SSN check	43	43	43	-	-
Orphan check	$6+N*33$	-	-	-	-
Update DV	$6+N*48$	-	-	-	-
DELIVER	0	0	0	0	0
Send Ack	145	145	145	-	-
Send Log	129	129	129	-	-
Update DV	-	$6+N*48$	$6+X*48$	-	-
Ack_Msg	304	304	304	-	-
Disk_Ack	274	274	274	-	-

Table 2: Breakdown of OS processing overhead for message logging incurred per application message in terms of assembly instructions executed. N is the number of processes and $X \leq N$. Reliable message delivery is assumed to be free. Each dependency vector entry fits in a single byte. There is no cost to send the *SSN*, just to check the sequencing.

# Nodes	4	8	16	32	64	128	256	dynamic
	OS overhead per appl. msg (in assembly instr. executed)							
Original	1392	1728	2400	3744	6432	11808	22560	$size*1.5$
Orph_roll	1254	1458	1866	2682	4314	7578	14106	$size*1.5$
DV_diffs	1188	1326	1602	2154	3258	5466	9882	$size*1.5$ $+X*51$

Table 3: Assembly instruction counts, as the hypercube is scaled. The dynamic count must be *added* to the static instruction count.

performance impact of each component, 2) devise how to improve the performance of the recovery scheme, and 3) measure the effectiveness of the resulting optimizations. This capability is illustrated below, using the asynchronous message logging technique. We broke the overhead of asynchronous message logging (*Original*) into its component parts — *Msg_Ovrhd*, *Chkpt_Ovrhd* (also called *Optimal*), and *No_Chkpt*. After measuring the performance impact of each of these, we devised and simulated optimizations to the kernel code (*Orph_roll* and *DV_diffs*) and hardware support (*Optimal*) to reduce the total overhead. The components of message logging overhead and the optimizations are briefly described below. The assembly-level operations (overhead), required by the message handling routines of the message logging algorithm (*Original*) and its optimizations (*Orph_roll*, *DV_diffs*, and *Optimal*), are listed in in Table 2. Table 3 shows the impact of scaling the hypercube on these routines in terms of total assembly instructions executed.

The algorithms investigated were:

- *Original* is the basic asynchronous message logging technique [21].
- *No_Chkpt* is the application without any recovery scheme and with an inexpensive send and receive protocol (see Table 2). In Figure 2 *No_Chkpt* is always 1.0. All schemes are graphed relative to *No_Chkpt*.
- *Msg_Ovrhd* is the application without any recovery scheme but with expensive send and receive primitives equivalent to the computation time performed by the OS in *Original*. This isolates the OS processing overhead (by the sender and receiver) in the *Original* scheme.
- *Chkpt_Ovrhd* assumes that the operations listed in Table 2 are performed in parallel with normal processing by special-purpose hardware (hence *Optimal*). This isolates the checkpointing overhead in the *Original* scheme.
- *Orph_roll* is the *Original* scheme with two optimizations we devised: 1) Messages are *not* examined to determine if they are orphans — orphan detection is delayed until a *Recovery* message arrives. The cost is the potential rollback of additional processes that could have been avoided by detecting the orphan message earlier. 2) The message is delivered to the application as soon as it passes the SSN protocol. The dependency vector is updated *after* the message is delivered; the OS must complete this

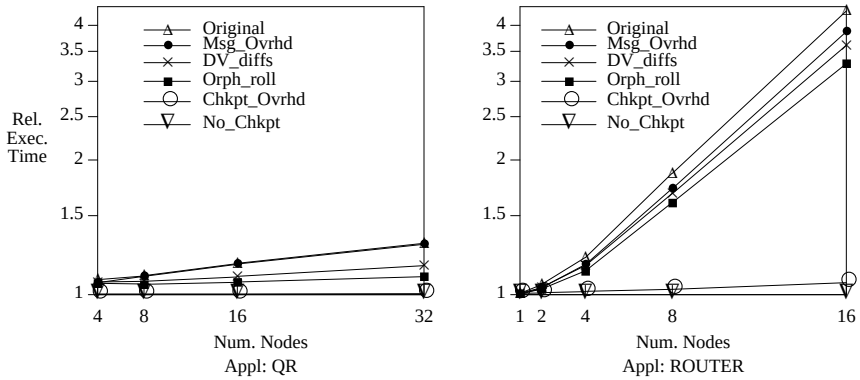

Figure 2: Application execution time with message logging relative to execution time with no recovery scheme. The checkpoint interval is 4600000 instructions.

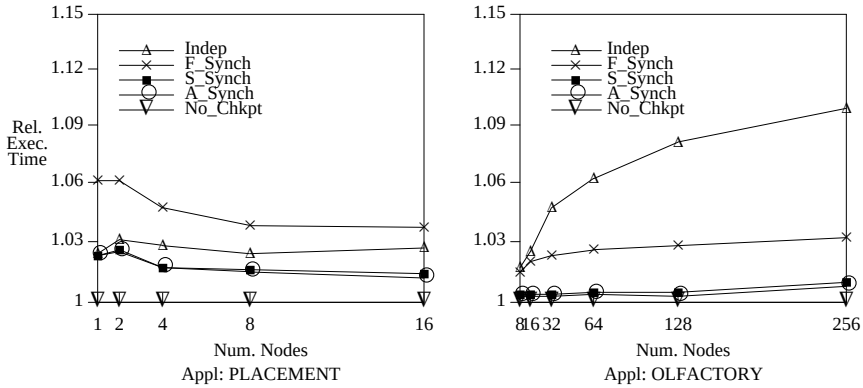

Figure 3: Application execution time with coordinated and independent checkpointing relative to execution time without checkpointing. The checkpoint interval is 4600000 instructions.

update before sending any new messages.

- *DV_diffs* is an optimization, previously proposed by others, that sends only the changes between the dependency vector previously sent by S to R and S's current dependency vector (X entries instead of N). These simulations *include Orph_roll* optimizations.

Figure 2 shows typical results from the simulator running these recovery algorithms for two of the applications (fft performed much worse, Placement a little better than QR, Olfactory a little better than Router, and Extract's overhead came only from checkpointing).

Using the asynchronous message logging technique we have shown that our testbed can be used to examine the impact of recovery techniques on actual distributed applications in a high-performance multicomputer environment and that the maximum flexibility afforded by such a simulator allows low-level investigation of architectural tradeoffs and optimizations.

B. Coordinated Checkpointing

Running simulations using coordinated checkpointing and independent checkpointing illustrates the importance of measuring recovery algorithm overhead on applications as they are executing. It is generally expected that independent checkpointing is less disruptive to system operation (e.g. [3, 18]) because only a single process is involved in a checkpoint session. This intuition is compounded by the fact that, with coordinated checkpointing, processes will checkpoint more often than is dictated by the checkpoint interval. Indeed, in the simulations most interacting sets contained *all* application processes and processes checkpointed measurably more often than with independent checkpointing.

We coded several coordinated checkpointing algorithms: *F_Synch*, synchronous coordinated checkpointing where processes do not resume execution until the checkpoint has completed, *S_Synch*, synchronous coordinated checkpointing where processes resume execution after the interacting set has been found, and *A_Synch*, our asynchronous coordinated checkpointing algorithm. The coded independent checkpointing algorithm is, in a sense, ''optimal'' because only the overhead of checkpointing is simulated — any costs for dependency tracking and/or message ordering are not simulated. Figure 3 shows some results of simulations using the Placement and Olfactory applications. In Olfactory, independent checkpointing

performed significantly worse than all three coordinated checkpointing algorithms, and in Placement, outperformed only the fully synchronous coordinated checkpointing algorithm. It appears that the coordination of checkpointing sessions can be beneficial. With independent checkpointing there is a higher probability that process y is checkpointing when process x is waiting to receive a message from y. With coordinated checkpointing x and y are likely to be checkpointing simultaneously, reducing CPU idle time. This confirms results presented in [6].

	Time Slice	# processes 2	4	8	16	32	64
Extract	450	1.1584	1.1739	1.1384	1.1215	-	-
	4.5K	1.1582	1.1740	1.1383	1.1215	-	-
	450K	1.1548	1.1722	1.1391	1.1143	-	-
	45M	1.1557	1.1753	1.1417	1.1156	-	-
Olfactory	450	-	-	1.0030	1.0034	1.0031	1.0036
	4.5K	-	-	1.0030	1.0033	1.0031	1.0036
	450K	-	-	1.0031	1.0033	1.0031	1.0036
	45M	-	-	1.0031	1.0033	1.0031	1.0036

Table 4: Execution time with asynchronous coordinated checkpointing relative to execution time without any checkpointing for several values of TIME_SLICE (specified in assembly instructions).

In Section III we discussed the use of bounded time slices for each simulation thread in order to handle the ‘‘out of sync’’ problem due to messages which are not anticipated. Table 4 shows the impact of varying the size of the time slice on the results obtained from the simulator for asynchronous coordinated checkpointing. These results increase confidence in testbed results since varying the time slice does not appreciably alter simulated application execution time.

# Nodes	1	2	4	8	16	32	64	128	256
	Simulation time (nsec) per application assembly instr.								
Extract:	2281	2305	2235	2210	2231	-	-	-	-
w/ indep	2694	2734	2677	2648	2600	-	-	-	-
OLF:	-	-	-	349	459	558	646	700	745
w/ indep	-	-	-	440	539	649	730	794	836
FFT:	-	-	-	-	946	1053	1153	1384	1832
w/ indep	-	-	-	-	1084	1138	1261	1486	1979
QR:	-	-	564	575	573	578	-	-	-
w/ indep	-	-	599	621	628	631	-	-	-
Place:	672	721	776	802	1052	-	-	-	-
w/ indep	1018	1068	1122	1169	1439	-	-	-	-
Router:	2028	2344	2680	3462	4877	-	-	-	-
w/ indep	2711	2995	3281	4090	5341	-	-	-	-

Table 5: Simulation time per application assembly instruction — for the application without checkpointing and with independent checkpointing.

VII. Simulator Performance

Finally, we comment on the efficiency of the simulator as it runs the applications. Since processes run sequentially on a uniprocessor, the simulator will be at least a factor of n slower than an n-node multicomputer running the application. We examine the overall efficiency of the simulator by measuring simulation time per application assembly instruction (Table 5). As can be seen from the table, the simulator execution rate ranges from a low of .187 MIPS (Router) to 2.71 MIPS (Olfactory) on a host whose rate is 25-35 MIPS. It appears that, for Router, a communication-intensive application, the overhead of sending messages in the simulator shows up as the application is scaled. For extract and qr, the simulator becomes relatively more efficient as the application is scaled because the application is becoming less efficient (e.g spending more time waiting for messages). For the remaining applications the simulator becomes less efficient as it spends more time switching processes contexts.

VIII. Previous Work

In [12], sender-based message logging was implemented on an ethernet-based network of Sun workstations (V-system). Applications composed of 8 processes were run to obtain experimental results of checkpointing, logging, and recovery overhead. In [3], synchronous coordinated checkpointing and independent checkpointing were implemented on an ethernet-based network of Sun-3/50 workstations. Input parameters were varied using ‘‘dummy’’ processes which ‘‘communicated’’ through Sun Unix message queues. Many simulation results were reported. In [6], coordinated checkpointing and independent checkpointing [2] were implemented on an ethernet-based network of Sun workstations (V-system). Applications composed of 16 processes were run to show that the coordination overhead of coordinated checkpointing can be quite small. In [28], communication trace-driven simulation of 8 process applications from an Intel iPSC/2 hypercube were used to determine rollback distance, percentage of messages logged, and number of checkpoints reclaimed. In [14], analytical simulations, using fixed parameters (e.g. message sending intervals of 500 to 7500 secs), were used to obtain some specific, limited results.

IX. Summary and Conclusions

The different sources of overhead of various distributed error recovery schemes preclude the use of analytical models for accurately evaluating the performance impact of these schemes. Hence, we have developed a simulation testbed for analyzing the performance of distributed application-transparent error recovery techniques. The testbed is realistic and accurate — application and operating system execution is simulated at the assembly instruction level through low-level timing and assembly code instrumentation.

The testbed simulates a scalable multicomputer (up to 256 nodes) and is currently able to run six applications and a number of recovery schemes. The recovery schemes are fully functional; an application will execute correctly in the presence of simulated node failures. We have described the functionality of the testbed, the difficulties we encountered during development, and some of our experiences simulating applications and recovery schemes.

The flexibility of the testbed facilitated directly comparing different error recovery schemes as well as devising and measuring various optimizations to the schemes. For example, we simulated coordinated and independent checkpointing (without message logging) and found that, for the target system and the applications used, the coordination of checkpointing sessions led to *less* overhead than with independent checkpointing. We were also able to evaluate optimizations to the message handling routines of asynchronous message logging schemes, measure the effectiveness of reducing the size of checkpoints using copy-on-write pages [8], simulate an ''optimal'' message logging scheme, and fine-tune a large number of parameters.

Acknowledgements

We are grateful to Prof. Prith Banerjee who provided us with the QR, Router, Extract, Placement, and FFT applications.

References

1. G. Barigazzi and L. Strigini, ''Application-Transparent Setting of Recovery Points,'' *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
2. B. Bhargava and S.-R. Lian, ''Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach,'' *7th Symposium on Reliable Distributed Systems*, Columbus, Ohio, pp. 3-12 (October 1988).
3. B. Bhargava et al., ''Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms,'' *Proceedings of the International Conference on Data Engineering*, Los Angeles, California, pp. 182-189 (March, 1990).
4. J. Bower et al., ''Piriform (Olfactory) Cortex Model on the Hypercube,'' *The Third Conference on Hypercubes, Concurrent Computers, and Applications*, Pasadena, CA, pp. 977-999 (January 1988).
5. K. M. Chandy and L. Lamport, ''Distributed Snapshots: Determining Global States of Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(1), pp. 63-75 (February 1985).
6. E. N. Elnozahy et al., ''The Performance of Consistent Checkpointing,'' *11th Symposium on Reliable Distributed Systems*, Houston, TX, pp. 39-47 (October 1992).
7. E. N. Elnozahy and W. Zwaenepoel, ''Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit,'' *IEEE Transactions on Computers* **41**(5), pp. 526-531 (May 1992).
8. T. M. Frazier, ''Application-Transparent Error Recovery Techniques for Multicomputers,'' Ph.D. Dissertation, Computer Science Department Technical Report CSD-950012, University of California, Los Angeles, CA (January 1995).
9. J.-M. Hsu and P. Banerjee, ''Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer,'' *17th Annual International Symposium on Computer Architecture*, Seattle, WA, pp. 260-269 (May 1990).
10. P. Jalote, ''Fault Tolerant Processes,'' *Distributed Computing* **3**(4), pp. 187-195 (September 1989).
11. D. B. Johnson and W. Zwaenepoel, ''Sender-Based Message Logging,'' *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).
12. D. B. Johnson and W. Zwaenepoel, ''Distributed System Fault Tolerance Using Sender-Based Message Logging,'' COMP TR90-119, Rice University, Houston, TX (May 1990).
13. D. B. Johnson and W. Zwaenepoel, ''Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,'' *Journal of Algorithms* **11**(3), pp. 462-491 (September 1990).
14. J. L. Kim and T. Park, ''An Efficient Protocol for Checkpointing Recovery in Distributed Systems,'' *IEEE Transactions on Parallel and Distributed Systems* **4**(8), pp. 955-960 (August 1993).
15. R. Koo and S. Toueg, ''Checkpointing and Rollback-Recovery for Distributed Systems,'' *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 23-31 (January 1987).
16. L. Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of the ACM* **21**(7), pp. 558-565 (July 1978).
17. B. W. Lampson and H. E. Sturgis, ''Crash Recovery in a Distributed Storage System,'' Technical Report, Xerox PARC, Palo Alto, CA (April 1979).
18. K. Li et al., ''Real-Time, Concurrent Checkpointing for Parallel Programs,'' *Symposium on Principles and Practice of Parallel Programming*, pp. 79-88 (March 1990).
19. B. Randell et al., ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2), pp. 123-165 (June 1978).
20. A. P. Sistla and J. L. Welch, ''Efficient Distributed Recovery Using Message Logging,'' *Eighth ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, Canada, pp. 223-238 (August 1989).
21. R. E. Strom and S. Yemini, ''Optimistic Recovery in Distributed Systems,'' *ACM Transactions on Computer Systems* **3**(3), pp. 204-226 (August 1985).
22. R. E. Strom et al., ''Volatile Logging in n-Fault-Tolerant Distributed Systems,'' *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 44-49 (June 1988).
23. S. M. Swope and R. M. Fujimoto, ''Simon II Kernel Reference Manual,'' Technical Report UUCS 86-001, University of Utah, Salt Lake City, UT (March 1986).
24. MIPS Computer Systems, *RISCompiler and C programmer's Guide*, 1989.
25. Y. Tamir and C. H. Séquin, ''Error Recovery in Multicomputers Using Global Checkpoints,'' *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).
26. Y. Tamir and T. M. Frazier, ''Application-Transparent Process-Level Error Recovery for Multicomputers,'' *Hawaii International Conference on System Sciences-22*, Kailua-Kona, Hawaii, pp. 296-305, Vol I (January 1989).
27. Y. Tamir and T. M. Frazier, ''Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing,'' Computer Science Department Technical Report CSD-910066, University of California, Los Angeles, CA (September 1991).
28. Y.-M. Wang and W. K. Fuchs, ''Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems,'' *11th Symposium on Reliable Distributed Systems*, Houston, TX, pp. 147-154 (October 1992).
29. W. G. Wood, ''A Decentralized Recovery Control Protocol,'' *11th Fault-Tolerant Computing Symposium*, Portland, Maine, pp. 159-164 (June 1981).