Chapter Five

Execution-Driven Simulation of Error Recovery Techniques

DERT (Distributed Error Recovery Testbed) is a testbed for simulating and evaluating the performance of several different classes of application-transparent distributed error recovery schemes. DERT is built on top of an event-driven, message-passing, object-oriented, multithreaded simulation kernel. Above the kernel is a simulated hypercube multicomputer. Actual compiled distributed applications are executed on the simulated multicomputer. Instrumentation of the application assembly code is used to account for application execution time and to allow accurate simulation of periodic application-transparent checkpointing. Checkpointing is implemented in full detail, including associated overhead per message, additional messages, and changes to the memory system. The testbed is fully functional — it can use one of several recovery schemes to restore one or more processes when a simulated node failure(s) occurs.

The overhead of distributed error recovery schemes is caused by a complex combination of additional message traffic, additional processing of sends and receives, as well as the checkpointing of process states. Realistic performance comparisons can only be done with actual applications and the ability to measure the impact of all of these factors. DERT provides for execution-driven simulation and permits 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 chapter describes the design, functionality, and use of DERT, focusing on problems encountered and on examples of its flexibility and realism.

5.1. Introduction

The reliability requirements of distributed systems and of scalable multicomputers can only be met using fault tolerance techniques. The mechanism for recovering a valid system state following the detection of an error 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: *message logging* [Powe83, Borg83, Stro85, John87, Stro88, Sist89, Jalo89, Wang92, Elno92a] (a kind of ‘‘independent’’, or uncoordinated, checkpointing [Rand78, Russ80, Wood81, Bhar88]) and *coordinated checkpointing* [Bari83, Tami84, Koo87, Tami89]. Our research has involved developing new recovery algorithms that fall into both classes (see Chapter 3 for our message logging algorithm, and Chapter 4 for our coordinated checkpointing algorithms). An essential facet of this work has been the performance analysis of these and other distributed recovery schemes.

Realistic performance analysis of distributed checkpointing and rollback techniques is difficult due to the numerous factors that must be taken into account (e.g. parameterization 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 [John90b, Bhar90, Wang92, Elno92b, Kim93], but with almost no analysis in the context of multicomputer systems — where normal communication latency is small (microseconds) and many applications are communication-intensive.

Our main goal in developing DERT was to measure the impact of error recovery techniques on the execution time of distributed applications on realistic multicomputers. We wanted to be able to see how the overhead changes as various system parameters change and as applications are scaled to additional nodes. Thus, accuracy (realism) and flexibility were the primary factors influencing the choice of an evaluation methodology. To facilitate research on error recovery schemes it was particularly important to be able to evaluate and optimize a variety of low-level architectural and operating system features.

With DERT, we were successful in satisfying the above goals for a distributed error recovery testbed. DERT is built on top of a multi-threaded simulation kernel, called *Simon* [Swop86]. 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, saving only the modified application state. The impact of several different recovery schemes and optimizations, on the execution time of both coarse- and fine-grained applications during error-free operation has been measured. It is the intention of this chapter to relay our experiences in developing and using this simulator. We have found it to be a powerful and flexible tool for evaluating the overhead and scalability of recovery schemes in the absence of a large multicomputer system. There are disadvantages with the choice, however, and these will be discussed.

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.

5.2. Testbed Design Considerations

The testbed is designed to evaluate the performance of application-transparent error recovery techniques running on a scalable multicomputer. We focus on the overhead incurred by these schemes during error-free operation, since failures are assumed to be relatively infrequent in the target system. Our goal is to explore recovery scheme overhead while varying a number of system characteristics: stable storage and network parameters, cost of operating system functions, etc. We are also interested in exploring a range of application characteristics such as communication frequency and the degree to which an application is scaled. Finally, we want to be able to compare message logging to coordinated checkpointing — schemes with unrelated overhead factors. This pointed to execution-driven simulation where system parameters can be controlled and the time to *execute* applications under various schemes can be measured, with detailed measurement of the overhead incurred by each error recovery technique.

We need to simulate application execution and operating system functions at a very low level to build confidence in the results. The primary metric we want is the application's execution time with a particular checkpointing scheme relative to the application's execution time without any checkpointing. For message logging techniques this means incorporating into the application's execution time the cost of sending and updating dependency vectors with each application message, logging application messages to stable storage, maintaining some additional bookkeeping information, and periodically saving the state of each process. Coordinated checkpointing techniques require that the periodic saving of the states of interacting processes be properly coordinated and that a list of process interactions since the previous checkpoint be kept, and that the application's execution time include the time to perform these functions. To further increase the confidence in our results we made the testbed fully functional; even though our testbed, at this time, is not being used to measure recovery performance, the recovery algorithms can be used to successfully recover from simulated node failures. The next section describes the testbed in detail.

5.3. The Testbed

The testbed is a variable-sized hypercube built on top of the multi-threaded simulator *Simon* [Swop86]. *Simon* is an event-driven simulation tool which provides routines for creating and managing objects, time-multiplexing the execution of object procedures, sending messages between objects, and debugging. Messages are delivered between the objects (implemented as Modula-2 coroutines) through output ports, input ports, and the ''links'' that connect them. Using these facilities we have constructed a simulator of a multicomputer. This testbed runs distributed applications whose code has been instrumented at the assembly level to 1) count the number of assembly instructions executed by the application and 2) take checkpoints at precise intervals. Thus we can evaluate the performance overhead of error recovery schemes when used by a variety of applications employing coarse-, medium- or fine-grained parallelism.

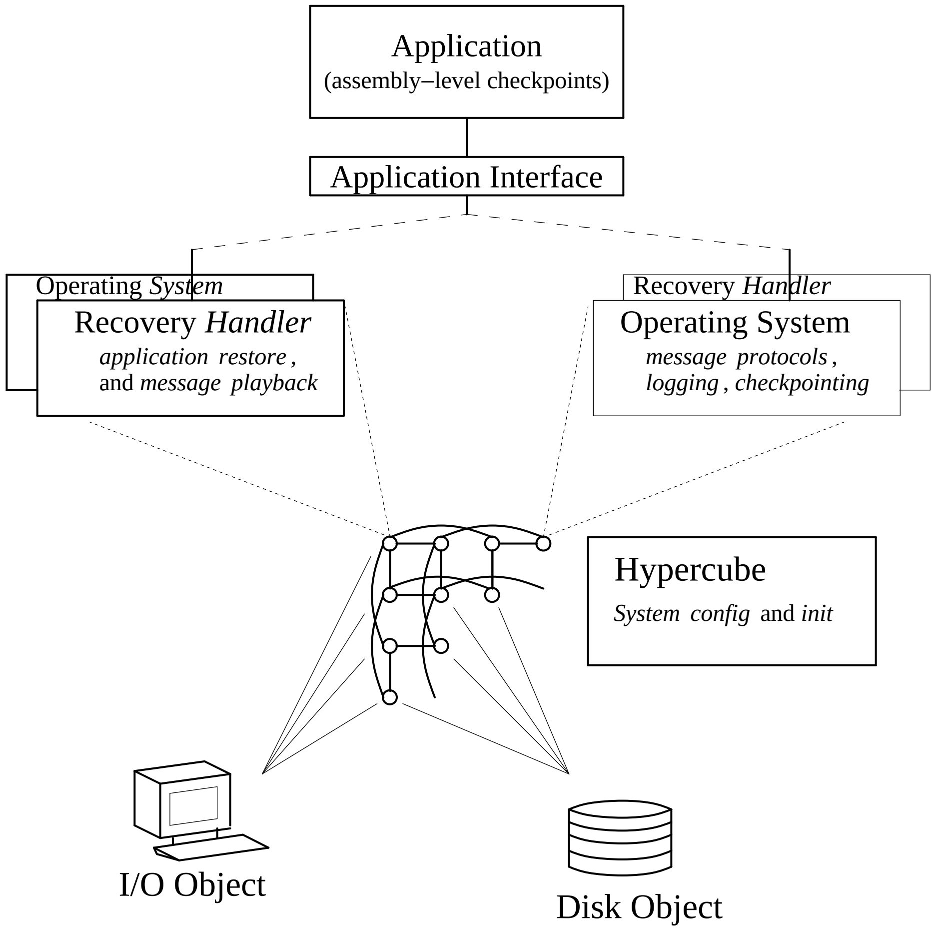

Figure 5.1: Structure of the simulation testbed.

The simulator is written in Modula-2, with some C and assembly routines, and runs on SPARC workstations. It runs distributed applications written in C or Modula-2. Two major error recovery techniques, and several optimizations, have been implemented: asynchronous message logging and asynchronous coordinated checkpointing. Independent checkpointing has also been coded. Nearly 20K lines of code have been written, with the two main algorithms exhibiting similar complexity. For the message logging algorithm, most of the code implements message handling, logging, dependency tracking, and recovery actions, while with the coordinated checkpointing algorithm most of the complexity rests on determining which action to perform in which phase of the algorithm.

There are four different types of objects (threads) in the testbed, described in the sections below: the application object (one per simulated node, up to 256 nodes),the operating system object (one per simulated node, up to 256 nodes), the I/O object (or ‘‘host processor’’, one object), and the disk object (one object).

5.3.1. The Interconnection Network

The interconnection network is arranged in a hypercube topology. This topology was chosen mainly due to the availability of hypercube applications. All of the distributed error recovery schemes discussed in this thesis are independent of any particular multicomputer topology. The size of the hypercube must be specified by the application itself (up to 256 nodes). Each of the nodes in the hypercube is actually simulated as two objects: the application process and the operating system. Thus the nodes being simulated are not multitasking and there is no context switching between the application and the operating system. Instead, events in the operating system and the application process are simulated as executing in parallel; this greatly simplified the design of the simulator. (Mayfly[Davi92], designed at HP Laboratories actually implements this model).

The network operates with an inexpensive store-and-forward datagram protocol using *e*-cube routing[Sull77, Lang82]. When link conflicts are encountered, messages are stored at that node until the link is available. Up to thirty in-transit messages can be stored at each input port. The simulated cost of sending a message, forwarding a message (through an intermediate node), or receiving a message is ten assembly instructions. Link service time depends on the length of the message — a byte is transmitted each $50\,nsecs$ (corresponding to a link speed of $20\,Mbytes/sec$).

Flow control and deadlock handling are incorporated into the testbed. The costs of flow control and deadlock handling are not simulated and the availability of global information in the testbed is exploited to keep this code simple. When a message needs to be sent out of an output port, the sending operating system object checks the availability of buffer space, in the input port at the other end of the link. If there is space in the input buffer for the message then the message is sent, otherwise the sending operating system object will suspend itself until buffer space is available (unless doing so makes it possible for deadlock to occur). The intended receiving operating system object eventually ''wakes up'' the sending operating system object when space becomes available.

To prevent deadlocks from occurring an operating system object will check, before suspending its execution to wait for buffer space, if such a suspension will result in a cycle of suspended objects. For example, operating system object 6 needs to send a message to operating system object 2 which is waiting on operating system object 3 which is waiting on the disk object which is waiting on operating system object 6. If operating system object 6 suspends its execution to wait for operating system object 2 to free up buffer space then a cycle of suspended objects exists and it becomes possible for deadlock to occur. If such a cycle would occur then the operating system object will *not* suspend itself, but will instead go ahead and send the message even though the input port has no (simulated) room.

5.3.2. Simulating the Execution of Application Processes

Simulating application execution at the clock cycle level meant performing substantial instrumentation of assembly code. This instrumentation served two purposes: to accumulate the number of assembly instructions being executed and to determine if a checkpoint should be taken. The application is compiled down to assembly instructions and then passed through a parser that inserts assembly code, at the end of each basic block, to count the number of executed instructions. Each instruction is assumed to execute in $100\,nsec$, with most floating point instructions taking $300\,nsec$ and multiply and divide instructions taking $900\,nsec$. This dynamic instruction count accumulation is straightforward and similar to tools like PIXIE [Syst89]. The count is kept in a ‘‘stopwatch’’ variable which is used to periodically synchronize the application process with the global simulation clock.

At the *beginning* of each basic block it is determined if it is time to take a checkpoint, by checking the accumulated running time since the previous checkpoint (of this process). The checkpoint interval is a parameter to the simulator and is specified in clock ticks. 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 and 2) executes a blocking receive to wait for a message from the operating system indicating that the checkpoint is sufficiently complete for the application to resume execution. When implementing this part of the instrumentation we encountered several SPARC architecture- and SunOS-specific problems. For example, performing a procedure call as part of the instrumentation meant that we first had to save (and later restore) register O7, of the current register window. This is due to the fact that the SPARC architecture uses the O7 register of each register window to store the return address for called procedures, which see the return address register as I7. Between procedure calls, however, register O7 is used to store temporary values. If code to save and restore O7 is not placed before and after an inserted call instruction, a ‘‘live’’ value in O7 may be lost.

In addition to this problem we also had to write an assembly routine to save (and later restore) the floating point registers in the floating point coprocessor. This is because TRANSFER(), the system call used by Modula-2 to context switch between coroutines, does not save or restore these registers. Erroneous behavior can occur if, when an application process is in the midst of a series of floating point calculations, a decision to checkpoint is made, and the subsequent transfer to another object happens without saving the floating point registers.

When engineering a major software project, it is desirable to avoid assembly-language programming and implementing architecture-specific routines. Tracking down low-level bugs which this approach generates within a complex simulator is not easy. Further, the portability of the testbed is severely limited. However, to achieve the highest degree of accuracy in our simulations, we had to run applications whose code had been instrumented at this level. This code is completely separate from the rest of the testbed, and once debugged has run without problem for many months.

Another problem we encountered could have significantly affected the accuracy of our simulations. Processes normally synchronize with the global simulation clock before a message is sent or received and before a checkpoint is taken. In some applications a process(es) may execute for an extended period of time without sending or receiving a message. Thus processes can get ‘‘out of sync’’ with one another and the global simulation clock. The problem this can cause can be illustrated as follows: a process *x* executes for a long time and then starts a lengthy delay to account for the time just executed. This process now receives an ‘‘unexpected’’ message from process *y* (e.g. ‘‘take a checkpoint’’). The problem is that *x* has executed far ahead of the point where it would have received the message if *x* and *y* were actually running in parallel (and not simulated as running in parallel). Such a simulation is correct — it corresponds to a *possible* execution, but probably not a likely execution. This problem exists whenever unanticipated messages or events can occur (e.g. interrupt handling) and it affects the number of checkpoints taken, the size of the checkpoints, and the application execution time. Our solution was simple: when we check if the checkpoint timer has expired (at the start of each basic block) we also check how much time has passed since the process was last suspended. If it is greater than some *TIME_SLICE* amount the process is forced to give up the processor and synchronize with the global simulation clock (see simulation measurements in Section 5.5).

5.3.3. The Operating System

Each application process (object) is bound to a separate operating system object, which can be called only through the three routines that make up the *application interface*: Send, Receive, and Checkpoint. The operating system handles the network interface, checkpointing and recovery duties, and the maintenance of bookkeeping information. All operating system routines — send, receive, memory-to-memory copies, table lookup, etc — have costs associated with them in terms of the number of assembly instructions required to perform the function; these costs are incorporated into the simulator. Thus all functions in our testbed are simulated at the clock cycle level for greatest accuracy.

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, for example, 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 order to determine which pages have been modified, the operating system keeps a checksum for each page from the previous checkpoint of the process. A new checksum is calculated at the beginning of the current checkpoint and the two are compared; a difference indicates that the page has been modified and thus needs to be checkpointed †. The new checksum is then stored. The taking of a volatile checkpoint can be optimized further (see Sections 5.6.3 and 4.6).

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.

5.3.4. The Host Processor (I/O Object) and Stable Storage (Disk Object)

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

† With 32-bit checksums, on average one of every 2^{32} modified pages is not detected using this method.

and collecting all outputs. There is no restriction as to how this is done.

A single fully-connected disk object is used to simulate m disks; the disk nodes are assumed to have unlimited buffering capability so conflicts do not exist. By default there is one disk per application process. The disk object's main function is to log messages and checkpoints in 100 byte blocks at a rate of three megabytes per second per disk; m disks are simulated by multiplying the speed of the disk by m.

5.3.5. Summary

Like the target multicomputer, the testbed can be scaled to hundreds of processes (nodes) which communicate solely via messages over point-to-point links. Hardware faults are simulated in the nodes, but not currently in the links. Reliable disk nodes are simulated by a single disk object with one three Mbyte/sec disk per application process; disk traffic does not traverse the network. All application and system operations are simulated at the assembly level. To reduce complexity, multitasking and context switching are not simulated; the operating system and application execute concurrently on a node.

5.4. The Recovery Schemes Simulated

Several different error recovery schemes have been incorporated into the testbed: asynchronous message logging (and optimizations), synchronous (fully and partially) and asynchronous coordinated checkpointing, as well as an ''optimal'' independent checkpointing scheme. Checkpointing and recovery are implemented in full detail for the message logging and coordinated checkpointing algorithms. The independent checkpointing algorithm simulates the checkpointing of application process states, but is ‘‘optimal’’ because it does not simulate *any* additional overhead, such as message ordering, maintenance of bookkeeping information, etc.

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. What follows is a brief itemization of the basic functions required by the message logging and coordinated checkpointing algorithms during normal operation and checkpointing that had to be incorporated into the testbed. A more detailed overhead itemization for these and other error recovery techniques can be found in Section 6.1.

(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 old 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 significant performance 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* operating system processing and network bandwidth are required to transmit messages, if message transmission error checking is incorporated into coordinated checkpointing as described in Section 4.3.1 and [Tami91].

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 the checkpoint: up to two copies 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 in this section is simulated by the testbed at the level of assembly instructions.

5.5. Experiences with the Applications

Six applications have been incorporated into the simulator: *fast fourier transform*, *Router*, *Placement*, *Matrix QR factorization*, *Olfactory* and *Extract* [Hsu90, Bowe88]. Some of the characteristics of these applications, determined by examining simulation output, are listed in Table 5.1. Fast fourier transform (*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 (Olfactory) 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 within the applications themselves 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 however, 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

No. Nodes	4	8	16	32	64	128	256
	Total number of assembly instrs executed by the application						
Extract:	2219M	3001M	3957M	-	-	-	-
OLF:	-	359M	486M	737M	1245M	2259M	4303M
FFT:	-	-	2017M	2081M	2160M	2260M	2422M
QR:	819M	819M	820M	821M	-	-	-
Place:	349M	468M	692M	-	-	-	-
Router:	254M	299M	351M	-	-	-	-
	Total number of messages sent and total message volume (in MBytes)						
Extract: # 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	-	-	-	-
Router: # msgs	127K	326K	715K	-	-	-	-
volume	7.2	16.3	15.5	-	-	-	-
	Avg. No. of pages modified after 146 million instructions (for Extract) and 46 million instructions (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 5.1: Application statistics: communication frequency (avg. number of assembly instructions executed between sends), number 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).

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 rest of the application's execution and resulting in significant changes to application execution time and application results. This made it nearly impossible to compare the execution times of the application with and without checkpointing and say anything meaningful about the overhead incurred by checkpointing.

There were two possible solutions. We could run multiple simulations with checkpointing, measure confidence intervals, and determine an average checkpointing overhead. Instead of this we chose the simpler solution of replacing 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.

Out of the large number of simulations that we have run, a very small number had unexpected results in terms the application's execution time, number of checkpoints taken, etc. This turned out to be due to the ‘‘out of sync’’ problem described in Section 5.3, which was solved by time-slicing the execution of the application processes. After this problem was solved, we ran additional simulations to determine a reasonable value for the TIME_SLICE variable and to see if our results are sensitive to the value of this variable (see Table 5.2). In general, varying the TIME_SLICE (specified in assembly instructions) does not appreciably alter application execution time, but was successful in eliminating our rare, aberrant results.

TIME_SLICE	# processes 2	4	8	16	32	64	128	256
	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	1.0034	1.0078
4.5K	-	-	1.0030	1.0033	1.0031	1.0036	1.0034	1.0078
450K	-	-	1.0031	1.0033	1.0031	1.0036	1.0036	1.0078
45M	-	-	1.0031	1.0033	1.0031	1.0036	1.0036	1.0078

Table 5.2: Execution time with asynchronous coordinated checkpointing relative to execution time without any checkpointing for several values of TIME_SLICE (the number of assembly instructions executed during a single timeslice).

5.6. Experiences with the Recovery Schemes

The design of the simulator was driven by the following goals: to examine the impact of multiple error recovery schemes on application execution time (without failures), to use a range of communication- and computation-intensive applications, and to examine the overhead of the schemes as the applications are scaled (without failures). Our simulator is fully functional; using one of the recovery algorithms, when a simulated failure occurs, one or more processes can be restored from checkpointed states and the application will correctly (re)execute from that point. However, we are predominately interested in the overhead incurred 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 space, memory, and swap space, increasing the degree to which the applications can be scaled. The main things we did were:

With all schemes:

— Account for the time the simulated disk(s) take to checkpoint process states, but do not checkpoint the states onto the workstation’s disk.

— Do not send checkpoints to the (simulated) disk, but simulate the sending of a message of the correct size and include the size information in the message.

With message logging:

— Do not log messages (or dependency vectors) onto disk or in the sender’s memory; simulate the size of the message but do not actually send/log it.

This greatly reduced the memory usage of the simulator such that it is dominated by the memory needs of the application, O(n) arrays used to store recovery information and collect statistics, and simulator information (object information, etc). 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 only 16 or 32 nodes.

5.6.1. 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 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 briefly illustrated below, using the asynchronous message logging technique.

We broke the overhead of asynchronous message logging (*AML*) into its major component parts — *Msg_Ovrhd* (overhead caused by dependency tracking, maintenance of bookkeeping information, etc) and *Chkpt_Ovrhd* (overhead caused by the checkpointing of process states, also called *Optimal*)). The functions of these major components are all simulated at the assembly level. After measuring the performance impact of each component, we then devised and simulated optimizations to the kernel code (*Orph_roll* and *DV_diffs*) and hardware support (*Optimal*) to reduce the total overhead, focusing on the message handling component (*Msg_Ovrhd*).

The assembly-level operations (overhead), required by the message handling routines of the message logging algorithm (*AML*) and its optimizations (*Orph_roll*, *DV_diffs*, and *Optimal*), are listed in Table 6.1. The algorithms investigated were:

- *AML* is the basic asynchronous message logging technique (Chapter 3 and [Stro85]).
- *No_Chkpt* is the application without any recovery scheme and with an inexpensive send and receive protocol. In Figure 5.2 *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 operating system in *AML*. This isolates the operating system processing overhead (by the sender and receiver) in the *AML* scheme.
- *Chkpt_Ovrhd* assumes that the message handling operations, simulated in the *Msg_Ovrhd* simulations, are performed in parallel with normal processing by special-purpose hardware (hence *Optimal*). This isolates the checkpointing overhead in the *AML* scheme.
- *Orph_roll* is the *AML* 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 operating system must complete this update before sending any new messages.
- *DV_diffs* is an optimization, previously proposed by others, that sends only the difference between the dependency vector previously sent by S to R and S's current dependency vector (i.e. X entries instead of N). These simulations *include* the *Orph_roll* optimizations.

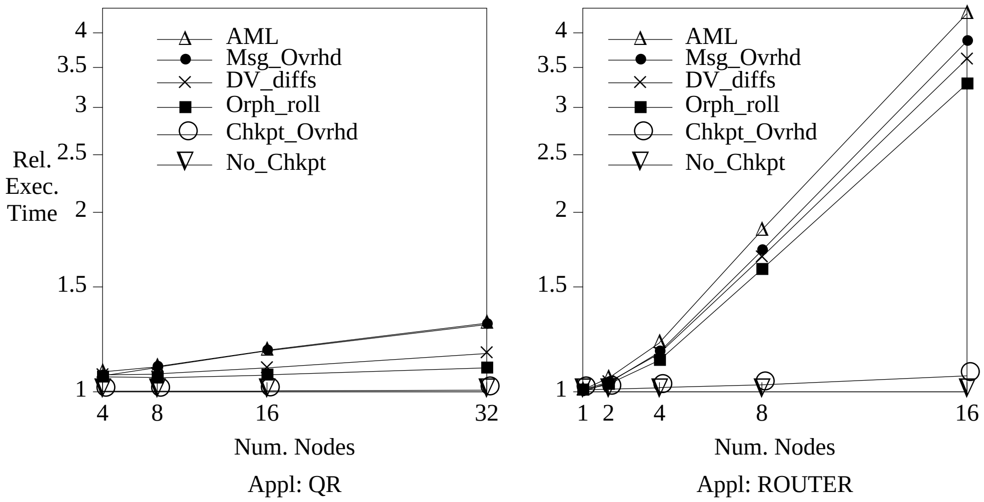

Figure 5.2: Application execution time with message logging relative to execution time with no recovery scheme. The checkpoint interval is 4600000 instructions.

Figure 5.2 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). A detailed performance analysis is contained in Chapter 6. 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.

5.6.2. 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. [Bhar90, Li90]) because only a single process is involved in a checkpoint session; coordinated checkpointing requires the ‘‘simultaneous’’ checkpointing of multiple processes. 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* of the application processes and processes checkpointed measurably more often than with independent checkpointing.

We coded several coordinated checkpointing algorithms: *F_Synch*, (fully-)synchronous coordinated checkpointing where processes do not resume execution until the checkpoint has completed, *P_Synch*, (partially-)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 5.3 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.

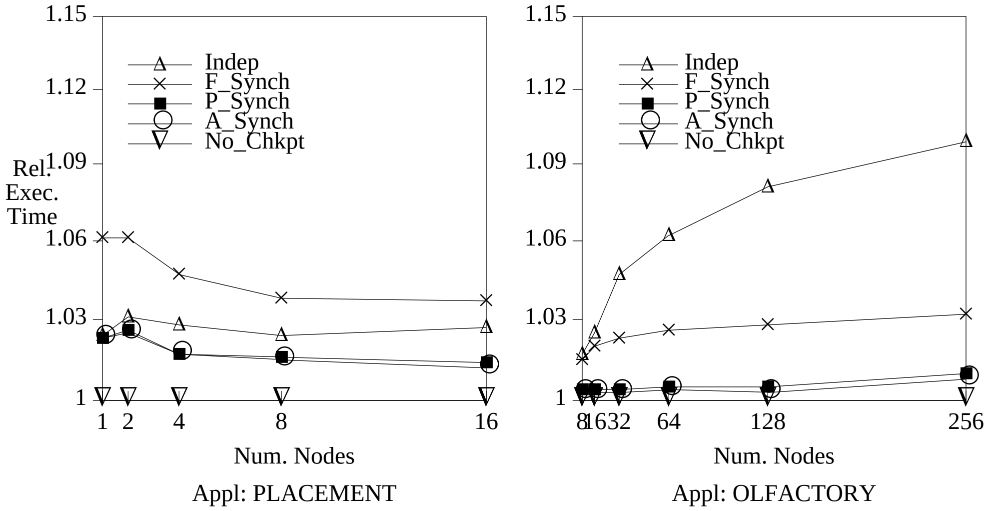

Figure 5.3: Application execution time with coordinated and independent checkpointing relative to execution time without checkpointing. The checkpoint interval is 4600000 instructions.

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 [Elno92b]. A detailed performance analysis of coordinated checkpointing can be found in Chapter 6. It is fairly easy to see the difficulty in comparing message logging and coordinated checkpointing techniques analytically; the results of our examination of coordinated and independent checkpointing, without message logging, further confirm the significance of using execution-driven performance analysis.

5.6.3. Optimizations

The flexibility of the testbed can be exploited to get an approximate measure of the effectiveness of the copy-on-write technique [Bobr72]. With the copy-on-write technique all modified pages are flagged for checkpointing (marked as copy-on-write) at the beginning of each checkpoint. The application process then resumes execution. The first time it attempts to write to a page marked copy-on-write then, if the page has not yet been written to stable storage, the process is suspended while the page is copied in volatile memory. After the copy-on-write has completed, the process continues execution. A reduction in overhead occurs when copy-on-write is performed for less than the total number of modified pages. Incorporating ‘‘true’’ copy-on-write into the simulator requires each memory access (during a checkpoint) to determine whether the page being written to is marked copy-on-write. Indirect and relative addressing modes made this low-level check all but impossible, so an approximation of copy-on-write was pursued. In [Elno92b], true copy-on-write was analyzed by altering the V-System kernel.

We had already implemented an efficient mechanism for tracking which pages have been written since the previous checkpoint (Section 5.3.3). We added a copy-on-write approximation to the simulator by computing the checksum for each (previously modified) page at fixed-length intervals after a checkpoint until all the pages have been written to stable storage. In our approximation we do *not* suspend the process when a new page has been written to and thus overestimate the number of pages that need to be copied in volatile memory. To correct for this we ‘‘speed up’’ writing pages to disk by an amount equivalent to the time that the process would have been delayed. (Thus we do not need to run separate simulations to get

Placement				Extract			
# Procs	CoW Pgs	Dirty Pgs	$\frac{\text{CoWPgs}}{\text{DirtyPgs}}$	# Procs	CoW Pgs	Dirty Pgs	$\frac{\text{CoWPgs}}{\text{DirtyPgs}}$
6.0 MB/s disks (one per process)							
1	1580	17456	0.09	1	55614	223903	0.25
2	3914	37837	0.10	2	41380	268921	0.15
4	2039	24918	0.08	4	36721	247437	0.15
8	3909	49031	0.08	8	33715	310941	0.11
16	4615	80685	0.06	16	46094	418667	0.11
3.0 MB/s disks (one per process)							
1	2781	17456	0.16	1	81521	223903	0.36
2	6478	37837	0.17	2	53902	268921	0.20
4	3460	24918	0.14	4	49413	247437	0.20
8	6492	49031	0.13	8	44444	320941	0.14
16	7887	80685	0.10	16	62255	418667	0.15
0.75 MB/s disks (one per process)							
1	7201	17456	0.41	1	126259	223903	0.56
2	14280	37837	0.38	2	81964	268921	0.30
4	7098	24918	0.28	4	81360	247437	0.33
8	12037	49031	0.25	8	64774	320941	0.20
16	13521	80685	0.17	16	100398	418667	0.24

Table 5.3: This table shows, for the applications *Placement* and *Extract*, 1) the estimated aggregate number of pages copied by the copy-on-write protocol, 2) the actual aggregate number of dirty pages, and 3) the fraction of dirty pages which need to be copied. 1) and 3) are calculated assuming that the disk server is writing dirty pages to stable storage at 6.0, 3.0, and 0.75 Mb/s per process.

our results). The length of the interval between checksum calculations determines the accuracy of the approximation; an interval of 2.25 msec appears to be accurate since, for the majority of intervals (e.g. 92 - 95 percent), *no* new pages are written.

We ran simulations of the copy-on-write approximation using *Placement* and *Extract*. Checkpoints were taken every 46 million instructions (*Placement*) and 146 million instructions (*Extract*). Table 5.3 shows results for disk throughputs of

.75, 3.0, and 6.0 Mbytes/sec. Even for slower disks, copy-on-write can be effective at reducing the number of pages which need to be copied in local memory as part of a volatile checkpoint.

5.7. Simulator Performance

No. Nodes	1	2	4	8	16	32	64	128	256
	Simulation time (nsec) per application assembly instruction								
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 5.4: Simulation time per application assembly instruction; numbers are given for the application without checkpointing and with independent checkpointing.

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 5.4). As can be seen from the table, the simulator execution rate ranges from a low of .187 MIPS (*Router*) to 2.71 MIPS (*Olfactory*), when run 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.

5.8. Previous Work

In [John90b], 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 [Bhar90], 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 [Elno92b], coordinated checkpointing and independent checkpointing [Bhar88] 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 [Wang92], 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 the number of checkpoints reclaimed. Finally, in [Kim93], analytical simulations, using fixed parameters (e.g. message sending intervals of 500 to 7500 seconds), were used to obtain some specific, limited, results.

5.9. Summary and Conclusions

We have developed a testbed for analyzing the performance of distributed application-transparent error recovery techniques. It 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, in detail, the functionality of the testbed, the difficulties we encountered during development, and some of our experiences simulating applications and recovery schemes. The actual performance evaluations of the recovery techniques are described in Chapter 6.

The development of the testbed was motivated by the desire to be able to compare the performance of message logging and coordinated checkpointing techniques in a scalable multicomputer environment. The differing sources of overhead for message logging and coordinated checkpointing techniques precluded the use of analytical models in order to achieve accurate results. Our comparison of coordinated and independent checkpointing (without message logging) has further indicated the significance of using execution-driven simulations (or experiments) instead of those driven by communication traces or employing fixed parameters. We found that the coordination of checkpointing sessions led, in general, to *less* overhead than did independent checkpointing, given our target system assumptions and the applications used.

We have striven to develop a testbed that 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 flexibility of the testbed has enables us to: directly compare different error recovery schemes, devise and measure optimizations to the message handling routines of asynchronous message logging schemes, measure the effectiveness of the copy-on-write technique, simulate an ‘‘optimal’’ message logging scheme, fine-tune a large number of parameters, readily switch from one recovery scheme to another, etc. Few resources are required, the testbed runs on any one of our SPARC 2 or SPARC IPC workstation (25 or 40 MHz, with 24 to 64 Mbytes of physical memory).

This testbed does have some drawbacks. The most serious is the restriction on the size of the application; the degree to which the application can scale is limited by the virtual and physical memory of a single workstation. An obvious solution to this would be to distribute the simulator to run on multiple workstations. Significant changes to the simulator would be required: for example, the deadlock avoidance code would have to be rewritten, as it currently utilizes global variables. Another disadvantage is that the simulation of a multicomputer, running actual distributed applications and recovery schemes, results in a large, complex testbed. Nonetheless, we have found this testbed to be a powerful, flexible tool for analyzing application-transparent error recovery techniques.