Chapter Seven
Conclusions

We have described several application-transparent error-recovery techniques which can be used to provide fault tolerance for large general-purpose multicomputers. Message logging techniques have the advantage of fast checkpoint and recovery sessions but may be inappropriate for high-performance multicomputer systems running communication-intensive applications due to the overhead of 1) operating system processing required to maintain and transmit bookkeeping information with each application message and 2) logging messages and bookkeeping information to disk (or local memory). Furthermore, message logging can only be used to provide fault tolerance for processes which are deterministic in their actions. Global checkpointing techniques are likely to incur the lowest level of overhead during normal operation but are only appropriate for use with applications that can tolerate the possible loss of many minutes of computation during recovery (e.g. batch applications). Coordinated checkpointing of interacting sets is a ‘‘compromise’’ between message logging and global checkpointing and therefore shows the most promise in providing error recovery for high-performance multicomputers that run both communication- and computation-intensive applications. Coordinated checkpointing techniques minimize the overhead incurred during normal operation at the cost of a potentially expensive recovery. In the worst case, the entire application may need to be rolled back as a result of the failure of one node. However, the overhead incurred during checkpointing is dependent almost entirely on the checkpoint frequency which can

be tuned to the specific needs of each application. Further, any application process which must be rolled back during a recovery session is always rolled back to its most recent (committed) checkpoint.

A correctness model was presented and used as the foundation for a complete set of proofs for the new asynchronous message logging algorithm and the new coordinated checkpointing algorithms.

The presented message logging algorithm is distributed, application-transparent, and n-fault-tolerant. Process checkpoints and interprocess messages are logged to stable storage asynchronous of process execution. Messages are also temporarily logged in the local volatile memory of the sender. This allows the algorithm to use volatile message logs in addition to stable message logs in order to recover as much of the failed process(es)' pre-failure execution as possible. The use of these volatile message logs reduces the number of non-failed processes (orphans) which must roll back in order to achieve a consistent system state. We were motivated to develop this algorithm when a previously published message logging algorithm failed to perform correctly when implemented on our simulation testbed. One of the major contributions of this chapter was the presentation of a complete proof that this optimized algorithm is *correct* - in the presence of failures the algorithm can be used to recover one or more processes in a distributed application without violating the application's partial ordering of events and without duplicating or deleting any events.

We have also presented two new distributed error recovery schemes, based on coordinated checkpointing of interacting sets of processes, that minimize disruptions to normal processing due to checkpointing. In the synchronous

coordinated checkpointing scheme an interacting set of processes is suspended from normal computation until all of the process states have been committed to stable storage. The asynchronous coordinated checkpointing scheme is based on the (coordinated) volatile checkpointing of interacting sets of processes followed by copying of these checkpoints to stable storage ‘‘in the background’’ involving minimal interference with normal computation. The asynchronous algorithm is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying any pages in local memory. The cost of asynchronous coordinated checkpointing, compared to the synchronous scheme, 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. Proofs of correctness were provided for both coordinated checkpointing schemes.

Both coordinated checkpointing schemes are integrated with an efficient error detection mechanism that avoids the need for message acknowledgments, transmission of message sequence numbers, and transmission of check bits with each message. Both schemes can recover from multiple node failures, lost messages, and corrupted messages; the domino effect can not occur and only one generation of checkpoints needs to be maintained on stable storage. Multiple checkpointing and recovery sessions may be active in the system simultaneously, operating independently if possible and merging correctly when necessary.

Realistic performance analysis of distributed checkpointing and rollback

techniques is very difficult due to all of the factors that must be taken into account (e.g. parameterization of the target system and of distributed applications). Thus, most performance analyses of distributed recovery techniques have been limited to greatly simplified analytical models. Realistic performance evaluations have appeared only recently, with almost no analysis in the context of multicomputer systems — where low overhead and scalability are highly desirable features in an error recovery scheme. Our goal was to measure the impact of error recovery techniques on the execution time of actual distributed applications as the applications are scaled. We built a Distributed Error Recovery Testbed (DERT) on top of an event-driven, message-passing, object-oriented, multithreaded simulation kernel in order to achieve accurate results, reasonable efficiency, and maximum flexibility, without consuming large amounts of resources. This testbed simulates a scalable multicomputer (up to 256 nodes) and runs on SPARC workstations; 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. Several recovery schemes can be simulated on the testbed. These 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 our experiences simulating applications and recovery schemes.

DERT has been used to measure the impact of several recovery schemes and optimizations, on the execution time of both coarse- and fine-grained applications during error-free operation. Using this execution-driven simulator we have shown

that the message handling overhead required by asynchronous message logging algorithms can have a substantial impact on application execution time. This overhead increases dramatically for more communication-intensive applications as they are scaled (up to 16, 32, or 256 processes, with constant problem size). Two optimizations we proposed significantly reduced recovery overhead, while a third optimization, previously suggested by others, often increased overhead. However, asynchronous message logging, even with the optimizations, still scaled very poorly for three of six applications. Additional simulations showed that hardware support may be required to eliminate this trend.

Three coordinated checkpointing algorithms were also coded into the simulator: fully-synchronous, partially-synchronous, and asynchronous. These algorithms were compared to one another and to independent checkpointing, without message logging, by measuring their impact on application execution time in a system without failures. Both the partially-synchronous and asynchronous coordinated checkpointing algorithms add less than six percent overhead to the application's execution time for five of the six applications (for a fixed checkpoint interval); independent checkpointing added less than fourteen percent. The sixth application's higher level of overhead (less than 20 percent for partially-synchronous and asynchronous coordinated checkpointing, less than 40 percent for independent checkpointing) is due to that application's larger working set. This overhead could be significantly reduced, for the asynchronous coordinated or independent checkpointing algorithms, if the copy-on-write optimization is available. In all cases, the percentage of overhead incurred by asynchronous coordinated checkpointing is nearly independent of the size of the multicomputer

(with constant problem size). Asynchronous coordinated checkpointing also exhibits less increase in overhead, compared with the other algorithms, as checkpoint frequency is increased.

In many simulations, the coordinated checkpointing of multiple processes resulted in significantly *less* overhead than independent checkpointing of processes. Independent checkpointing is usually assumed to have lower overhead because no coordination of checkpointing sessions is required and because coordinated checkpointing results in processes being checkpointed more often than is dictated by the checkpoint interval. This assumed lower overhead makes independent checkpointing attractive even though it may suffer from the domino effect during recovery. However, we found that coordinated checkpointing actually performed better than independent checkpointing in many cases because processes are more likely to be checkpointing at the same time, overlapping much of their checkpoint overhead, and doing useful work at the same time — actually resulting in *less* total overhead.

In conclusion, using the proposed techniques it is possible to use, in particular, asynchronous coordinated checkpointing to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics involve minimal performance overhead and are completely transparent to the user.