Chapter Six
Performance Evaluation

In this chapter we characterize the overhead incurred when *any* checkpointing/rollback scheme is used to provide fault tolerance for a scalable multicomputer system similar to that which we describe in Chapter 2. We then characterize and compare the overhead incurred by several different message logging and coordinated checkpointing techniques. In Sections 6.2.1 and 6.2.2 we evaluate the performance of several specific techniques using DERT — our simulation testbed described in Chapter 5.

6.1. Detailed Itemization of Recovery Scheme Overhead

In a system which supports only non-interacting processes, checkpointing involves periodically suspending each process, copying the process' state to stable storage, and then allowing the process to resume normal computation. When a node fails, all information on the node is assumed to be lost. The checkpointed states of all the processes that were on the failed node are restored from stable storage to working nodes. The processes may then resume computation from these older states[Rand78]. In order to be able to tolerate node failures that occur during checkpointing, the old checkpoint of a process is erased only after the new checkpoint has been completely written to stable storage.

When processes interact with one another, additional costs are incurred by the checkpointing and rollback algorithms in order to ensure that a consistent system state can be recovered whenever a failure(s) occurs.

The overhead of a fault tolerance scheme is determined by the frequency of checkpointing, the ‘‘cost’’ of saving each checkpoint, extra operations required during normal computation for error detection and maintenance of bookkeeping information, and the ‘‘cost’’ of recovery. The rest of this section details the overhead of the checkpoint and recovery algorithms used by a ‘‘generic’’ algorithm which checkpoints processes independent (for non-interacting and interacting processes), the independent checkpointing algorithm presented in [Bhar88], optimistic message logging[Stro85], two other major message logging techniques [Elno92a, Wang92], and synchronous and asynchronous coordinated checkpointing of interacting sets of processes.

6.1.1. Overhead Of Checkpointing/Recovery of Independent Processes

Certain facets of overhead are shared by all checkpointing and rollback algorithms. What follows is an itemization of a ‘‘generic’’ recovery scheme that checkpoints and rolls back non-interacting processes without any coordination between checkpointing and recovery sessions. We also examine the impact that communication between processes can have on this overhead itemization.

a) *computation time lost during checkpointing and normal operation*: Without the use of redundant hardware, a fraction of the processing cycles on each node must be devoted to checkpointing. Specifically, each node that has processes that are participating in a checkpoint session, must suspend normal operation in order to run the handlers that perform the various functions necessary for checkpointing. If the communication coprocessor can operate independently of the application processor, normal execution of *other* processes on the node may resume before the

checkpoint session is complete. Depending upon whether the checkpointing scheme is synchronous or asynchronous the process being checkpointed may need to be suspended until its checkpoint session is complete and its state is stored on stable storage. With synchronous checkpointing a process does not resume normal execution until it has been confirmed that its checkpoint is committed to stable storage. With asynchronous checkpointing a volatile checkpoint is taken and then the process being checkpointed can resume normal operation. Taking a volatile checkpoint involves copying the entire process state, or just the modified state, into local volatile memory. The copy-on-write technique can be employed to reduce the amount of local copying required, as discussed in Section 4.6.

If processes interact with one another then checkpointing results in the indirect loss of additional computation time. This happens when an application process has to wait (longer) to receive a message because the sending process is currently suspended from execution due to its participation in a checkpointing session. The use of an underlying reliable communication protocol, required by the recovery algorithm, may also cause some loss of computation time as can the maintenance of local or remote bookkeeping information. Lost computation time directly impacts the time it takes to execute an application.

b) *load on the communication network*: In the ‘‘worst case’’ the entire process state has to be sent to disk. If virtual memory is employed, part of the state is already on disk and the size of the state in local memory, which has to be sent to disk, is related to the process’ *working set*. With appropriate hardware support for identifying modified pages (Section 4.6), only the process state which has been changed since the last checkpoint is sent to disk.

If processes interact then a checkpoint session may involve some coordination between checkpoint handlers thus increasing load on the communication network. Extra bookkeeping information and message logs or queues may need to be transmitted and saved to ensure that a consistent system state can be recovered in the event of a hardware failure. Again, if an underlying reliable communication protocol is required by the recovery algorithms then this will also cause some increased load on the communication network.

c) *disk bandwidth required*: A significant fraction of the total disk bandwidth available on the system may be needed for saving checkpoints, thus limiting the performance of I/O-intensive applications.

Additional bandwidth may be needed for bookkeeping information and message logs or queues.

d) *disk storage required*: The minimum storage requirement is one complete checkpoint state per process plus additional space for storing a second copy of the checkpoint of processes during checkpointing sessions.

If processes interact then multiple checkpoints may have to be stored (due to domino effect) as well as bookkeeping information and message logs or queues.

e) *storage required in local memory*: Part of local memory is needed for storing program code for handlers and for volatile checkpoints.

If processes interact then bookkeeping information (e.g. a list of processes that have communicated with local processes) and message logs or queues may also need to be stored. Such use of local memory can increase paging activity, load on communication links and the average memory access time.

f) *special processing during recovery*: Each node participating in a recovery session must suspend normal operation so that it can run the recovery handlers. If processes interact then some coordination between recovery handlers will also be required.

g) *computation time lost due to rollback*: When a process is rolled back, the computations it performed since its last checkpoint are lost and have to be repeated. This overhead is inversely proportional to the frequency of checkpointing.

If processes interact then the computation time lost can be more significant. It is still possible that only the failed process(es) will have to roll back (e.g. if synchronous message logging with independent checkpointing is employed). However, some techniques that checkpoint processes independently may need to roll back a failed process to a checkpoint that is not the most recent (due to domino effect), and some additional non-failed (orphaned) processes may also have to roll back. If processes checkpoint in a coordinated fashion then a group of processes will need to be rolled back together to yield a consistent system state. An increase in network load may also result from the replaying of computations.

h) *delay in committing outputs to the outside world*: Outputs can not be committed to the outside world until it can be guaranteed that they will not be rolled back in the event of a failure(s). When processes do not interact outputs produced by a particular process can be committed to the outside world after the next checkpoint of that process has completed.

When processes interact output commitment delays vary widely from algorithm to algorithm. For many algorithms outputs can not be committed until

the next recovery line is established on stable storage (via coordinated checkpointing or independent checkpointing). Output commitment is identical to the issue of checkpoint reclamation. If an output can be committed then all previous checkpoints (and message logs) can be reclaimed.

Bhargava and Lian have proposed an algorithm based on the independent checkpointing of process states[Bhar88]. With this algorithm, the use of a reliable end-to-end communication protocol is assumed and processes can be nondeterministic in their execution. The overhead incurred by this algorithm can be broken down as follows:

a) *computation time lost during checkpointing and normal operation*: The overhead lost during checkpointing is ‘‘minimized’’ — checkpointing is asynchronous of process execution (uses volatile checkpoints) and involves only one process at a time. Each process maintains an input information table (IIT) which is updated on stable storage every checkpointing session. This two-dimensional table is used to keep track of message exchanges between the various checkpoint intervals of all of the processes. During normal operation, computation time is lost due to the processing needed to append a checkpoint interval number and sessions number onto each application message and also to check the sequencing of those numbers upon receipt of the message. The checkpoint interval number must be inserted into the current (local) entry of the IIT after a message has been received.

b) *load on the communication network*: Extra load on the communication network is incurred when a process state and the corresponding IIT are sent to disk. Addition load is required for the reliable communication protocol and to send the

checkpoint interval number and sessions number with each message.

c) *disk bandwidth required*: For the checkpointing of process states and the updating of the IITs.

d) *disk storage required*: For each process one (possibly large) IIT is maintained and one or more checkpoint states. In the worst case, all checkpoint states may have to be retained until the application completes.

e) *storage required in local memory*: For local volatile checkpoints (if used), the current entry of the IIT, and local bookkeeping information (small).

f) *special processing during recovery*: All communicating processes participate in the recovery session. Significant processing is required to use the IITs to calculate the most recent recovery line on stable storage.

g) *computation time lost due to rollback*: A group of communicating processes is rolled back to the recovery line which has been calculated by the recovery algorithm. This recovery line can include the current states of non-failed processes. The amount of computation time actually lost is highly dependent on the communication behavior and checkpoint frequency of the application. Nonetheless, this algorithm does not guarantee any progress of the recovery line, thus processes may have to roll back to their initial states. Non-failed processes continue with normal operation until it has been determined that they must be rolled back. Thus the recovery handlers run, at least in part, asynchronous of process execution.

h) *delay in committing outputs to the outside world*: Outputs can be committed to the outside world after the next system-wide recovery line has been established. In the worst case this may not occur until the application has completed. Reclamation

of checkpoint states is the same.

This algorithm is tolerant of *n* process failures and the worst case scenarios can be avoided if it is possible to periodically invoke a coordinated checkpointing algorithm to establish a recovery line on stable storage.

6.1.2. Overhead of Asynchronous Message Logging

The overhead incurred in a system using optimistic message logging (Chapter 3[Stro85]), can be characterized as follows:

a) *computation time lost during checkpointing and normal operation*: The time lost during checkpointing is ‘‘minimized’’ — checkpointing is asynchronous of process execution (uses volatile checkpoints) and involves only one process at a time. Throughout normal operation processing is required by the sender to append the dependency vector and (updated) send sequence number onto each application message. The receiver must check the send sequencing, determine if the message is an orphan (compare the dependency vector to the IST), update the receiver’s dependency vector, and update the receive sequence number. Each message must also be both acknowledged and logged to stable storage. All communication must be through a reliable protocol. Logging status is periodically broadcast and checkpoints and message logs are periodically reclaimed (a coordinated effort).

b) *load on the communication network*: Includes checkpoints (plus current local bookkeeping information), the logging of all messages sent between processes, the overhead of the reliable communication protocol (including message acknowledgments), and the transmission and logging of bookkeeping information (e.g. dependency vectors). This overhead is further examined in the next

subsection.

c) *disk bandwidth required*: Similar to (b).

d) *disk storage required*: Generally, the storage requirement might be one checkpoint state per process plus the stable message logs. Older checkpoints are discarded as soon as it can be determined that they will not be needed for either *state backout* (undoing the effects of messages) or for resending messages lost by a failed process. Earlier message logs can also be committed/reclaimed at this time (as soon as all the state intervals upon which a message depends have been logged). This is dependent on the communication behavior of the applications, the rate at which messages are logged, and the frequency of checkpointing. If some processes are slow to log messages and/or checkpoint then all processes that (transitively) interact with these processes will not be able to reclaim storage as quickly as they otherwise would.

e) *storage required in local memory*: Local storage requirements include volatile checkpoints, messages being held until they have been logged, dependency vectors, SSNs, RSNs, INCs, log vectors, etc.

f) *Special processing during recovery*: Collect any volatile logs from the senders, and the stable log from the disk. The message log must then be played back to the recovering process(es) until it is brought up-to-date. The IST is updated and *Recovery* messages are broadcast.

g) *lost computation time due to rollback*: All processes *directly* affected by a hardware failure must be rolled back, along with orphans, if any. Processes may have to roll back multiple times (to earlier checkpoints) unless they are, by default, restored to the earliest non-reclaimed checkpoint. There is no need to wait for

messages to be sent ‘‘on-demand’’ by non-failed (non-rolled back) processes because these messages are already present in the message log.

h) *delay in committing outputs to the outside world*: A message can not be committed to the outside world until every state interval upon which it depends has been logged. This is determined by comparing the message’s dependency vector to a log vector (which records the logging progress of every process).

Optimistic recovery is a distributed, n-fault-tolerant, application-transparent scheme requiring an underlying reliable communication protocol. The only restriction made on application processes is that they be deterministic.

6.1.2.1. The Cost of Logging Messages to Disk

Some application programs for multicomputers may require each process to send a short message (say, eight bytes) every 100 instructions [Jage87, Dall87]. Such communication-intensive applications may be incompatible with message logging techniques since very large amounts of data may have to be logged. For example, consider a multicomputer with 512 20 MIPS processors running such an application. In this system a total of approximately **820 Mbytes** are sent as messages every second. Logging this much information is a major problem. Specifically, high-performance disk drives have a maximum bandwidth of 3 Mbytes per second. Thus, if all messages are to be logged to disk, as required by Strom and Yemini’s original scheme[Stro85], more than *200* disk drives will have to be dedicated to logging messages.

Message logging techniques also require bookkeeping information (dependency vectors, sequence numbers, etc) to be transmitted with each message.

As a result, the load on the communication network and the complexity of send and receive operations, which have to manipulate this bookkeeping information, is significantly increased. Some of this information (e.g. dependency vectors) must also be logged to disk, further increasing the number of disks required to support message logging techniques.

6.1.2.2. Other Message Logging Techniques

In sender-based message logging[John87] messages are logged in local, volatile store and correct operation can not be assured if more than one process can fail at a time. In n-fault-tolerant logging[Stro88] messages are logged in volatile store until it is determined that they can be discarded or spooled (logged to disk). Using the above example, volatilely logging messages in local memory involves storing an average of 1.6 Mbytes/sec (391 4K pages every sec) on every node. In both cases performance will be affected by having to copy messages to memory and because some fraction of main memory is needed to maintain message logs.

To reduce the local memory usage with sender-based message logging [John87], processes may have to checkpoint frequently in order to limit the size of the message logs. In the case of n-fault-tolerant logging [Stro88], the amount of information to be logged to disk is reduced by discarding some messages and predicting the arrival order of messages (using a deterministic heuristic). Performance will depend on how well the heuristic performs, on how many messages can be discarded, and on how much compute time and how many extra messages are required to run the heuristic. We do not itemize the overhead of sender-based message logging because we are restricting our attention to n-fault-

tolerant algorithms. We do not itemize the overhead of n-fault-tolerant logging because a complete algorithm was not given (many details were omitted including where and how dependency vectors are maintained, where and how $<SSN, RSN>$ pairs are logged, etc). Many of these omitted details are essential to both the performance and correctness of the approach.

Two more algorithms demand closer examination: Manetho[Elno92a] and Wang and Fuchs' optimistic message logging[Wang92], which is not really a ''true'' message logging technique. In Manetho messages are logged in the local volatile memory of the sender and antecedence graphs are used in place of dependency vectors. An antecedence graph is simply an extension of the dependency vector in which *all* the events (state intervals) upon which a state interval (or message) depends are recorded. This eliminates orphan processes and orphan messages since all messages sent by non-failed processes have (in the sender's local memory) all of the ordering information which is needed to regenerate the state interval during which that message was sent. Furthermore, a message may be committed to the outside world as soon as its antecedence graph has been (synchronously) logged to stable storage.

The cost is that an antecedence graph can grow well beyond the size of a dependency vector and it must be transmitted with every application message just as a dependency vector is. The size of a process' antecedence graph is reduced by periodically logging it to stable storage and then transmitting and updating only the unlogged (''current'') portion. Updating the receiver's antecedence graph is a more complex operation than updating a dependency vector and each node in the graph has five fields instead of three. Additionally, it must be possible to be able to

record, as part of the antecedence graph, all the information necessary to regenerate an identical internal process event during a recovery session. Other message logging techniques do not have this restriction. The overhead breakdown of this technique is as follows:

a) *computation time lost during checkpointing and normal operation*: Processes checkpoint independently and asynchronously. In addition to the process' state, the volatile message log and current antecedence graph are also saved to stable storage during the checkpoint session. Throughout normal operation processing is required by the sender to append the antecedence graph onto each application message and to save a copy of the message and graph in local volatile memory. The receiver must check the sequencing of the message and update its local antecedence graph. All communication must be through a reliable protocol. The antecedence graph may be periodically logged to stable storage during normal operation.

b) *load on the communication network*: Includes checkpoints (plus current antecedence graph and volatile message logs), the overhead of the reliable communication protocol, and the transmission and logging of bookkeeping information (e.g. antecedence graph).

c) *disk bandwidth required*: Similar to (b). All volatile message logs are written to stable storage (with antecedence graphs) so bandwidth needs may be substantial as discussed above.

d) *disk storage required*: At most two checkpoint states per process plus the volatile message logs (saved to stable storage during each checkpoint) and the antecedence graphs. Generally, the storage requirement might be one checkpoint

state per process plus the stable message logs. If desired, processes can be forced to checkpoint so that older messages logs and antecedence graph nodes can be reclaimed. In general, garbage collection should be straightforward and storage needs should not be great (despite the fact that bandwidth requirements may be quite significant).

e) *storage required in local memory*: Local storage requirements include volatile checkpoints, the volatile message logs including antecedence graphs, the local current antecedence graph and a small amount of additional bookkeeping information. Since all messages are logged in the local memory of the sender, local memory requirements may be substantial, as discussed in Section 6.1.2.1, and adversely impact performance.

f) *special processing during recovery*: All interacting processes (senders) participate in recovery. (More accurately, remote procedure calls are invoked, by the recovering process, on the node of each process which had sent message(s) to it prior to the failure). Each of these processes (synchronously) log their antecedence graphs to stable storage and then search their volatile message logs for any messages needed by the failed process(es) to restore a state consistent with the non-failed processes in the system.

g) *computation time lost due to rollback*: Only the failed processes need to roll back. Significant processing time is needed to retrieve the volatile message logs, log all the antecedence graphs to stable storage, and to replay the log.

h) *delay in committing outputs to the outside world*: A message can be committed to the outside world after a single synchronous disk operation which logs the message's antecedence graph to stable storage.

This algorithm is tolerant of n process failures, requires communication to be reliable, and assumes that processes are deterministic.

Wang and Fuchs proposed an optimistic message logging algorithm which is not really a ‘‘true’’ message logging technique. Unlike other message logging algorithms processes can be nondeterministic. Like coordinated checkpointing techniques this algorithm records (logs) the messages sent along ‘‘channels’’ between processes, but checkpointing is independent, not coordinated, so this information can be significant. One of the main assumptions of this approach is that there is no access to the communication facilities in the multicomputer and thus that it can not be determined, by the application and checkpointing routines, if a particular message has arrived (correctly or not).

The basic approach is to log all messages in local volatile memory at the receivers and then determine, during the next checkpointing session, which of these messages actually belong to ‘‘channels’’ (as in coordinated checkpointing techniques; here they are called ‘‘state messages’’). These messages are then logged to stable storage. Since checkpointing is not synchronized it is possible that a large number of messages will be ‘‘state messages’’ and thus have to be logged to stable storage. To determine which messages are state messages, each application message must be tagged with the process’ current checkpoint interval (incremented at the beginning of a checkpoint session), a processor (process) identifier, and a send sequence number. Each processor (process) also maintains a communication information table (on stable storage) which records the dependencies between the checkpoint intervals of interacting processes. During recovery these tables can be used to determine a recovery line. Interacting

processes will thus be rolled back to this recovery line (somewhat similar to coordinated checkpointing techniques). Checkpoints are reclaimed in two ways: 1) checkpoints taken earlier than the most recent global recovery line can be discarded and 2) a reclamation algorithm can be run to determine which checkpoints can never belong to a recovery line and thus can be discarded.

In general this algorithm, similar to [Bhar88], does not guarantee that the recovery line will progress. Depending on the communication behavior of the application and the likelihood of ‘‘accidentally’’ saving a recovery line, some number of checkpointed states may actually never become become part of any recovery line and thus are ‘‘wasted overhead’’.

a) *computation time lost during checkpointing and normal operation*: Processes checkpoint asynchronously and independently, requiring no coordination overhead. Processing overhead is required to append a checkpoint interval number, send sequence number, and processor identifier onto each message. The receiving process logs all incoming messages into its local volatile memory. The communication information table is updated during each checkpointing session and all state messages are logged to stable storage. Additional processing overhead is incurred to determine which messages are state messages.

b) *load on the communication network*: Additional load on the communication network is required for the message tags, volatile checkpoints, transmission of state messages to stable storage, and to update the communication information table.

c) *disk bandwidth required*: Similar to (b).

d) *disk storage required*: Each process has, on stable storage, one communication information table, one or more checkpoint states, and the corresponding state

messages. In the worst case, all (‘‘non-obsolete’’) checkpoint states may have to be retained until the application completes. A reclamation algorithm can be used to determine which checkpoint states can never belong to a recovery line (and thus may be reclaimed).

e) *storage required in local memory*: Local memory is needed for volatile checkpoints, a small amount of bookkeeping information, and the volatile message log. All messages are logged in local volatile memory until the next checkpoint session. During the checkpoint session some of these messages are discarded and the rest are logged to stable storage. As discussed in Section 6.1.2.2, logging a large number of messages in local memory can adversely impact application performance.

f) *special processing during recovery*: During recovery significant processing is required; all non-failed processes participate in the determination of the recovery line by using communication information tables to construct an extended checkpoint graph.

g) *computation time lost due to rollback*: The recovery line can include the current states of non-failed processes but multiple non-failed processes may still need to be rolled back. The processes which do roll back may have to roll back to states older than their most recent checkpoint. In fact, the algorithm does not guarantee any progress of the recovery line, thus processes may have to roll back to their initial states.

h) *delay in committing outputs to the outside world*: Messages may be committed to the outside world only after the establishment of the next recovery line (which may not occur until the application completes). Checkpoints taken prior to the

recovery line may be discarded and a special checkpoint reclamation algorithm can be used to garbage collect checkpoint states that can *never* be part of a recovery line.

This algorithm is tolerant of n process failures and, as in[Bhar88], the worst case scenario could be avoided if it is possible to periodically invoke a coordinated checkpointing algorithm to establish a recovery line on stable storage (e.g. using the packet-switching approach, Section 4.3.4, and a high-level approach to reliable communication since access to low-level communication facilities is assumed to be impossible).

6.1.3. Overhead Of Coordinated Checkpointing of Interacting Sets

Unlike most other error recovery schemes, coordinated checkpointing of interacting sets of processes does not assume that the message delivery system is made reliable via sliding windows (or two-phase commit protocols) and error detection bits for two reasons. First, the use of sliding windows has the stringent requirement that message transmission delays must be bounded while the use of two-phase commit protocols imposes a high level of overhead on every message. Secondly, communication errors occur very infrequently relative to the frequency at which messages are sent. Hence, it is not desirable to penalize the system continuously by checking for errors on every message. Instead, detection of communication errors (and recovery from them) is incorporated into the checkpointing/recovery algorithms[Tami84, Tami89] (Chapter 4).

In[Tami89, Tami91] ‘‘normal’’ interprocess messages have no error detection bits, no bookkeeping information, and require no acknowledgments. Instead

signatures (accumulated by Linear Feedback Shift Registers, see Section 2.4), which are updated in parallel with message transmission, are maintained for each pair of communicating processes. For example, if virtual circuits are used for interprocess communication then signatures are accumulated for each incoming and outgoing virtual circuit. Communication between processes in the interacting set can be checked during a checkpoint session by performing ‘‘end-to-end’’ checks on all the virtual circuits between processes in that set. If a ‘‘mismatch’’ is detected then the interacting set containing the processes on both ends of the virtual circuit are rolled back. Messages which are used to coordinate the creation of checkpoints and for error recovery must be verified before they are used and hence carry error detection bits. This reduces the total overhead incurred by the error recovery scheme during normal operation.

The overhead of *synchronous* coordinated checkpointing can be generally characterized as follows:

a) *computation time lost during checkpointing and normal operation*: The interacting set of processes is suspended until the checkpointing session has terminated. The interacting set may contain, in the worst case, all of the processes that constitute the application. If the nodes are time-shared, they may continue to execute other processes. Executing the checkpoint handlers and coordinating their activities also results in lost computation time. During normal operation, updating the *Buddies* list may require some very small processing overhead.

b) *load on the communication network*: Increased load on the communication network results from sending checkpoints and the current contents of the message queues to disk. This traffic may be ‘‘bursty’’ because multiple processes are

checkpointed ‘‘simultaneously’’. Additional overhead is minimized since i) no additional check bits are added to normal messages, and ii) there is no need to log messages to stable storage. During checkpointing there is increased load on the communication network due to coordination messages.

c) *disk bandwidth required*: Only process checkpoints are sent to disk. This bandwidth is thus entirely dependent on application size and checkpoint frequency. Timers can be used to fine tune this amount of checkpoint state on a ‘‘per application’’ basis (true for any checkpointing technique).

d) *disk storage required*: One checkpoint per process (two during checkpointing). Only the most recent committed recovery line needs to be retained.

e) *storage required in local memory*: ‘‘Minimal’’: only dynamic communication information (*Buddies* list) needs to be stored.

f) *special processing during recovery*: Recovery of a failed process requires the coordination of the interacting set of processes containing the failed process. Messages may need to be flushed and discarded and the checkpoint states need to be restored.

g) *computations lost due to recovery*: All processes directly affected by an error *and* their interacting set(s) must be rolled back. In the worst case the interacting set(s) contains all of the processes in the application which must then be rolled back to their most recent checkpoints.

h) *delay in committing outputs to the outside world*: Messages can not be committed until the next checkpointing session completes.

The overhead of *asynchronous* coordinated checkpointing can be generally

characterized as follows:

a) *computation time lost during checkpointing and normal operation*: A process is suspended until the volatile checkpoint has been taken and *Checkpoint* messages have been queued to all processes with whom direct communication has taken place since the last checkpoint (the *Buddies* list). Processes may have to be suspended again during the checkpoint session if a synchronous send is executed and the target process might be in *or* outside of the interacting set. As soon as it has been determined whether or not the target process is part of the interacting set the suspended process may resume executing. Asynchronous sends never *directly* result in the process being suspended, although the message will not actually be sent until it can be determined whether the receiving process is part of the interacting set of the sender. Thus, the sender may be suspended if it expects an immediate response to the message. Executing the checkpoint handlers and coordinating their activities also results in lost computation time. During normal operation updating the *Buddies* list may require some very small processing overhead.

b) *load on the communication network*: Same as for synchronous coordinated checkpointing.

c) *disk bandwidth required*: Same as for synchronous coordinated checkpointing.

d) *disk storage required*: Same as for synchronous coordinated checkpointing.

e) *storage required in local memory*: ‘‘Minimal’’: dynamic communication information needs to be stored during normal computation as well as storage for volatile checkpoints and holding buffers during checkpoint sessions.

f) *special processing during recovery*: Same as for synchronous coordinated

checkpointing.

g) *computations lost due to recovery*: Same as for synchronous coordinated checkpointing.

The proposed synchronous and asynchronous coordinated checkpointing schemes are n-fault-tolerant, do not require communication to be reliable, and make no restrictions on application behavior.

6.2. Performance Evaluation Using DERT

The performance overhead of error recovery techniques can be broken into three parts: 1) overhead incurred during normal operation (not including the cost of checkpointing process states); 2) the cost of checkpointing process states; and 3) the cost of recovery when failures occur.

Since failures in the target system are assumed to be infrequent, the performance analysis in this dissertation is limited almost exclusively to examining the overhead that error recovery techniques impose on the system during normal operation (with checkpointing) relative to a system that has no logging or checkpointing. We examine asynchronous message logging in detail (as presented in Section 3.5 and[Stro85]), and then asynchronous coordinated checkpointing (as presented in Section 4.5) along with an ‘‘optimal’’ independent checkpointing algorithm.

We focus on the impact of the checkpointing algorithms on application execution time relative to the execution time of the application without any checkpointing. Thus we isolate (a) *computation time lost during checkpointing and normal operation* and make sure that the (simulated) (b) *network bandwidth*

required and (c) *disk bandwidth required* are more than sufficient to support the error recovery schemes and the applications. These execution time measurements also ignore the (d) *disk storage required* and (e) *local memory storage required* by the recovery schemes, as was discussed in the previous section. We will examine (b) and (c) in the case of asynchronous message logging and asynchronous coordinated checkpointing and briefly address the issue of (h) *delays in committing outputs*. Simulations were run to test the correctness, but not the performance, of the recovery algorithms (f and g).

If the communication network and/or disk bandwidth can not support the increased load caused by the use of a particular error recovery scheme, then the execution time of the application may be directly or indirectly affected. We ignore any additional overhead that might result from limited available network and disk bandwidths by ensuring that both are more than adequate. For the applications we used and the recovery schemes we examined this was easily satisfied by the default 20 Mbyte/sec links in our testbed and by simulating a dedicated 3 Mbyte/sec disk at each node. This dedicated disk is used solely for the checkpointing of process states and logging of application messages. In most cases much less bandwidth is actually needed to support these activities.

The amount of local memory storage required by an error recovery scheme can also impact the execution time of the applications. This overhead is ignored by our testbed; no paging activity is simulated (i.e. all applications are completely resident within a node's local memory) and the available local memory is assumed to be ''infinite''. Available disk storage is also assumed to be infinite.

6.2.1. Performance of Asynchronous Message Logging

Our research addresses the design and performance of error recovery schemes on massively parallel systems. Thus we have focused our attention on scalability issues and on examining applications that exhibit different communication/computation ratios. The trend in multicomputers is toward systems that have communication protocols with very small overhead (e.g. Transputer, Paragon, CM-5[Sper93], J-machine[Nuth92, Noak93], AP1000[Hori93], active messages[Eick92], etc). Previous performance analyses of message logging techniques have looked at small-scale multiprocessors (16 nodes or less) where communication is relatively expensive[John90b, Wang92]. It is not surprising that the additional overhead incurred by message logging, in terms of operating system processing during normal computation and per-message overhead, may not be significant in such systems (especially for ethernet-based multiprocessors).

We limit our evaluation here to the asynchronous message logging algorithm presented in this dissertation (based on[Stro85]) and a few simple optimizations. In all of the simulations processes are checkpointed independently. Checkpointing a process involves 1) copying *only the modified 1K byte pages* of the process state in local dedicated buffer space, then 2) waking up the application, and 3) transferring the process state, in the background, to stable storage. This overhead is isolated in the graphs (see *Chkpt_Ovrhd* below).

The rest of the overhead stemming from the message logging scheme is incurred during normal operation. A small amount of overhead results from the periodic reclamation of disk storage (checkpoints and message logs) and the

periodic broadcast of each process' logging status. The remaining overhead is determined by the amount of per-application-message processing that must be performed by the operating system (and/or hardware). Thus, this potion of the total overhead is highly dependent on the communication behavior of a particular application. Table 6.1 itemizes the per-application-message processing, in terms of the number of assembly instructions required to perform each function. We assume that the base communication protocol used by the target multicomputer is quite inexpensive (ten assembly instructions to perform either a send, a ''forward'' through an intermediate node, or a receive).

AML is the asynchronous message logging algorithm presented in Chapter 3. The processing performed by the operating system, in order to *send* an application message, entails: copying the O(n) dependency vector onto the message, sending the message from the node and then saving a copy of the message and its dependency vector in local volatile memory. The operating system processing required to *receive* an application message entails: checking the send sequence number of the message, determining if the message is an orphan, updating the receiver's dependency vector, delivering the message to the application, and then sending an acknowledgment to the sender and logging the message to disk. There are additional costs associated with 1) each link transfer, 2) each intermediate node, 3) the processing, by the sender, of the acknowledgment from the receiver and 4) the processing, by the receiver, of the acknowledgment from the disk. These costs, listed in Table 6.1, are all incorporated into the simulator.

Looking at the first half of Table 6.2 we can see how the operating system processing overhead, associated with each application message, increases as the

	AML	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:	$\frac{size+N+4}{2}$	$\frac{size+N+4}{2}$	$\frac{size+X+4}{2}$	$\frac{size+N+4}{2}$	$\frac{size}{2}$
Intermediate 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 6.1: Breakdown of operating system processing overhead incurred per application message in terms of assembly instructions executed. N is the number of processes and $X <= N$.
A one word checksum is sent with each message (and the message is saved at the senders) to satisfy the reliable message transmission requirement. Check bit processing is assumed to be free. Each dependency vector entry (incarnation number and *RSN*) is (optimistically) assumed to be contained in a single byte. Additionally, there is no cost to send the send sequence number (just to check the sequencing). There is additional overhead in the form of a periodic broadcast of logging status and a periodic (coordinated) attempt to reclaim old checkpoints.

No. Nodes	4	8	16	32	64	128	256	dynamic
O/S overhead per application message (in assembly instructions executed)								
AML:	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)
No_Chkpt:	20	20	20	20	20	20	20	20
Optimal:	20	20	20	20	20	20	20	20
Avg. No. of assembly instrs executed by each application process between sends								
Extract:	1.71M	1.01M	.498M	-	-	-	-	-
OLF:	-	128K	64.9K	39.4K	27.7K	21.5K	18.0K	-
FFT:	-	-	63.0K	26.0K	11.2K	5.04K	2.37K	-
QR:	14.8K	14.8K	14.8K	14.8K	-	-	-	-
Place:	475K	28.4K	5.28K	-	-	-	-	-
Router:	2000	917	492	-	-	-	-	-

Table 6.2: Total assembly instruction counts; for each of the five schemes (*AML*, *Orph_roll*, *DV_diffs*, *No_Chkpt*, and *Optimal*) the total number of assembly instructions executed by the operating system *per application message* is divided into a static part (dependent on the number of nodes/processes) and a dynamic part. The dynamic part must be added to *each* of the static totals. *size* represents the size of the application message, in bytes. The second half of the table shows, for each of the six applications, the average number of assembly instructions executed between sends (from the same process). These numbers are readily calculated from the information given in Table 5.1.

number of application processes is scaled. In addition to the presented *AML* algorithm we also looked at several optimized versions of the algorithm which attempt to reduce the amount of per-application-message processing required: *Orph_roll*, *DV_diffs*, and *Optimal*. The different recovery schemes simulated were:

- *AML* is the presented asynchronous message logging scheme.
- *No_Chkpt* is the application without any message logging or checkpointing

and with an inexpensive send and receive protocol (see Table 6.1). In Figures 6.1 and 6.2 *No_Chkpt* is always 1.0. All other schemes are graphed relative to *No_Chkpt*.

- *Msg_Ovrhd* is the application without checkpointing or message logging but with expensive send and receive primitives equivalent to the computation time performed by the operating system of the *AML* scheme. This isolates the contribution of operating system processing, by the sender and receiver, to the overall overhead for the *AML* scheme.
- *Chkpt_Ovrhd* assumes that the operations listed in Table 6.1 are performed ‘‘for free’’ in parallel with normal processing by special-purpose hardware. This isolates the overhead contributed by checkpointing process states plus a small periodic cost for broadcasting log vectors and reclaiming checkpoints. Thus, $Msg_Ovrhd + Chkpt_Ovrhd = AML$.
- *Orph_roll* is the presented asynchronous message logging scheme with the following two optimizations: the orphan detection filter is eliminated and the update of the dependency vector is performed *after* the message is delivered to the application process (see Table 6.1). Since the process’ dependency vector *is* updated with the dependency vector on each application message then, when a *Recovery* message is received the recovery handler can still determine that the application’s state has been orphaned. The cost is the potential rollback of an additional process(es) that could have been avoided by detecting the orphan message earlier. However, if failures are infrequent relative to the frequency of communication then this cost may be justified. The second optimization delivers the message to the application as soon as it

passes the SSN protocol. The dependency vector can be updated *after* the message is delivered to the application as long as the operating system can not be interrupted to deliver a new message from the application *before* completing the update.

- *DV_diffs* is an optimization, previously proposed by others, that computes the difference between the dependency vector previously sent from a process S to a process R and S's current dependency vector - which is then sent with the application message (i.e. X entries instead of N). These simulations *include* the *Orph_roll* optimizations.

Table 6.1 shows the breakdown of operating system processing overhead required, per application message, for each of these simulated algorithms. Table 6.2 shows the change in the total number of assembly instructions executed by the operating system, per application message, (from Table 6.1) as the hypercube and application are scaled. The second half of Table 6.2 indicates, for each application, the frequency with which processes communicate with one another (given as the average number of assembly instructions an application process executes between sends).

Figures 6.1 and 6.2 show the overhead of the different recovery schemes in terms of increased application execution time for the applications *fft*, *Router*, *Placement*, *Matrix QR*, *Olfactory*, and *Extract*. The checkpoint interval is just under 500 msec for *fft*, *Router*, *Placement*, matrix *QR*, *Olfactory* and just under 1500 msec for *Extract*. The default disk bandwidth of 3Mbytes/sec was insufficient to support the checkpointing of *Extract*'s large process state every 500 msec. The y axis in these graphs is the application's execution time with a

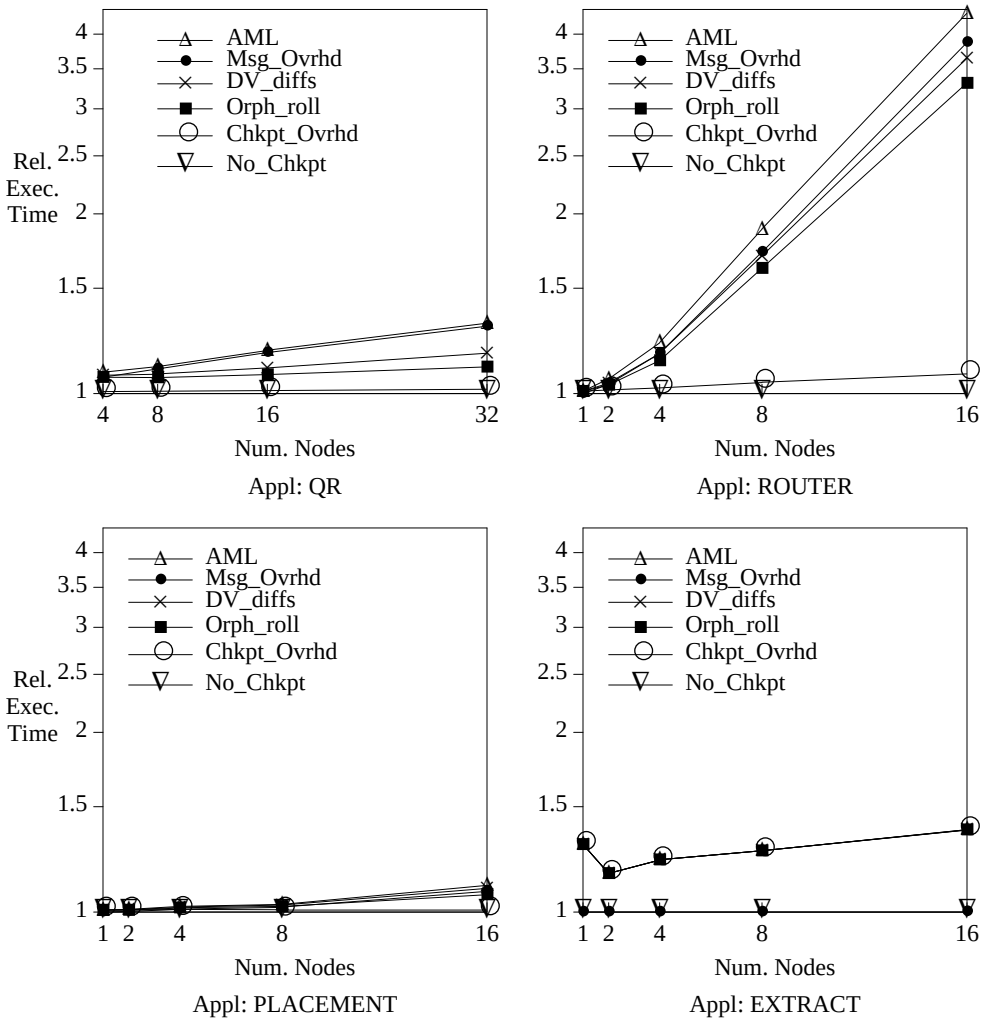

Figure 6.1: Application execution time with message logging relative to execution time without any recovery scheme. Checkpoints are taken every 4600000 instructions (14600000 instructions for *Extract*).

particular checkpointing algorithm relative to the application's execution time without any checkpointing. This axis is given in logarithmic scale so that all of the graphs would be both readable and use the same scale. The *x* axis indicates the number of processes that constitute the application and ranges from 1 to 256 nodes (depending upon the application). In all cases the problem size is held constant

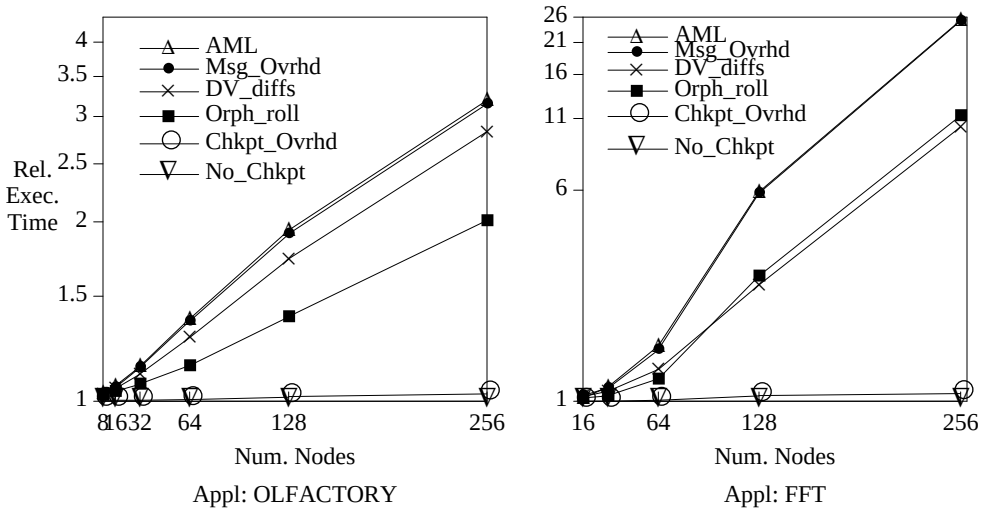

Figure 6.2: Continuation of Figure 6.1. Application execution time with message logging relative to execution time without any recovery scheme. Checkpoints are taken every 4600000 instructions. (Note change in Y axis for the *fft* graph).

while the application is scaled.

fft has very little process state to checkpoint (two to four 1Kbyte pages, see Table 5.1 in Section 5.5) therefore almost all of the recovery scheme overhead is due to processing by the operating system, which is confirmed by the *Msg_Ovrhd* simulations. This overhead scales quite poorly — overhead is only 3 percent and 13 percent when *fft* is comprised of 16 and 32 processes, but climbs to 60 percent, 592 percent and 2535 percent when the application is scaled to 64, 128, and 256 processes. It is worth noting that, in our simulator, the application and operating system do not need to be context switched since they are simulated as executing in parallel. Having to perform these context switches would further increase the total overhead incurred by operating system processing. Table 6.2 helps to clarify the poor scalability of the recovery algorithm. This table shows that the amount of

processing performed by the application between message sends decreases as the application scales while, at the same time, the amount of operating system processing required per message increases, due to the increasing size of the dependency vector.

The optimizations *Orph_roll* and *DV_diffs* offer significant improvement but still scale poorly (reaching overhead levels of 1026 and 1130 percent, respectively, when *fft* is scaled to 256 processes). Extensive hardware support, if feasible, could reduce or eliminate this problem as evidenced by the *Chkpt_Ovrhd* simulations. All overhead above the *Chkpt_Ovrhd* curve is *completely independent of the checkpoint frequency* (for these simulations where operating system does not need to block waiting for disk accesses to complete) and is dependent on the communication frequency of the application and the size of the messages sent.

Like *fft*, *QR* also has a small process state (one to two pages) and the majority of its overhead comes from operating system processing. With 32 processes, 95 percent of the 31.2 percent total overhead can be attributed to the per-application-message operating system processing. In general, we found that the *Orph_roll* optimizations perform well (with 32 processes, the total overhead reaches only 10.8 percent for *QR*). Adding the optimization of sending only the changes to the dependency vector, in addition to the *Orph_roll* optimizations, actually increases the overhead for *QR*, *Router*, *Placement*, and *Olfactory* applications. This is because the dependency vector difference must be computed on-the-fly as part of the send primitive. Thus, this optimization does not appear to be worthwhile for these applications and our assumptions.

Router is the most communication-intensive application (see Table 6.2) and

the operating system processing overhead incurred by message logging again scales very poorly. The optimized algorithms show significant improvement but hardware assistance is needed to eliminate the trend. *Placement* exhibits the same trend as it is scaled but to a significantly reduced degree (overhead is less than ten percent for all simulations). The overhead of the computation-intensive *Extract* application is fairly ‘‘steady’’ (between 16 and 37 percent) and is *completely* dependent on the checkpoint frequency. This is not surprising since messages are sent only at a rate of one per 0.5 to 1.7 *million* instructions executed. Lastly, *Olfactory* simulations show results similar to *fft* and *Router*.

These simulations have led us to conclude that for applications, like those used, that run in a multicomputer environment and are medium- or fine-grained, the operating system message handling overhead required by asynchronous message logging algorithms may be prohibitive. This problem worsens considerably as the applications are scaled. *Chkpt_Ovrhd* simulations suggest that hardware support, if feasible, can have a dramatic impact on overhead.

6.2.1.1. Overhead Due to Network Bandwidth

In our simulations the message logging algorithm’s network bandwidth requirements did not contribute to increased application execution time. This is an optimistic result for two main reasons —

1) applications are assumed to be completely resident in the node’s local memory and thus no paging activity occurs when the application is not being checkpointed, and

2) every node has a dedicated link to a dedicated disk so that paging activity due

No. Nodes	Total Messages	Percent Recovery	Percent Application	No. Nodes	Total Messages	Percent Recovery	Percent Application
Router				*FFT*			
1	372	93.3	6.7	16	155012	79.3	20.7
2	269378	75.0	25.0	32	340132	76.4	23.6
4	508067	75.0	25.0	64	971460	80.2	19.8
8	1312325	75.1	24.9	128	826383	87.3	12.7
16	2868440	75.1	24.9	256	2003983	88.1	11.9
Placement				*Olfactory*			
1	2659	65.5	34.5	8	13544	79.2	20.8
2	62774	74.7	25.3	16	35477	78.9	21.1
4	66492	74.8	25.2	32	82096	77.2	22.8
8	151004	76.1	23.9	64	272913	83.6	16.4
16	530161	75.2	24.7	128	826383	87.3	12.7
				256	2003983	88.1	11.9
Matrix QR				*Extract*			
4	223493	75.2	24.8	1	943	91.3	8.7
8	223012	75.2	24.8	2	2460	86.6	13.4
16	229751	75.9	24.1	4	7841	83.4	16.6
32	277501	80.0	20.0	8	16805	82.3	17.7
				16	50881	84.4	15.6

Table 6.3: Total messages sent in each simulation. Also given is the percentage of this total which is attributable to the error recovery algorithm (*AML*) and the percentage attributable to the application.

to checkpointing does not require any network bandwidth either.

Thus only application messages and messages required by the recovery algorithm (acknowledgment and log vector broadcasts) require network bandwidth. Further, only one application runs on our testbed at a time, thus the network is hardly stressed to begin with.

In our system model (Chapter 2) we assumed that the disk nodes would be remotely located, not locally, and therefore, when examining the total number of messages required by the recovery algorithm we included the additional messages

required to log application messages to disk along with the disk acknowledgment messages. In our simulator both of these types of messages actually use only the dedicated link to disk and thus have no impact on network traffic. The total number of messages sent in each simulation are listed in Table 6.3 and are ‘‘worst-case’’ in the sense that acknowledgment messages and log messages are not piggybacked onto application messages, but are transmitted separately. From this table we can see that from 65.5 to 93.3 percent of the total messages sent are due to the asynchronous message logging scheme. The information in these tables was taken from the same simulations that produced the graphs in Figures 6.1 and 6.2.

6.2.1.2. Overhead Due to Disk Bandwidth

The disk bandwidth requirements of asynchronous message logging were not a significant problem for several reasons:

1) The process states were small — ranging from 80 Kbytes (*fft*) to approximately 3 Mbytes (*Extract*).
2) Only the portion of the process state modified since the previous checkpoint was checkpointed.
3) Each node was simulated as having its own disk connected by a dedicated link.
4) The applications were not very communication-intensive (except for *Router*).

Table 6.4 indicates the percentage of application execution time that each disk was kept busy logging messages and checkpoints. It also shows the breakdown (by percentage) of time spent on each activity. Table 6.5 shows the *total* disk

No. Nodes	Percent Exec. Time	Relative Checkpoint	Percent Logging	No. Nodes	Percent Exec. Time	Relative Checkpoint	Percent Logging
Router				*FFT*			
1	1.8	(99.5	00.5)	16	10.8	(5.2	94.8)
2	4.7	(22.3	77.7)	32	12.9	(4.2	95.8)
4	7.6	(9.2	90.8)	64	13.3	(2.7	97.3)
8	10.9	(3.8	96.2)	128	10.3	(1.1	98.9)
16	9.1	(2.5	97.5)	256	8.0	(0.5	99.5)
Placement				*Olfactory*			
1	5.1	(99.7	0.3)	8	11.1	(6.8	93.2)
2	5.5	(83.4	16.6)	16	16.4	(4.3	95.7)
4	3.5	(72.5	27.6)	32	20.5	(2.8	97.2)
8	2.8	(58.6	41.4)	64	21.8	(2.6	97.4)
16	3.4	(42.7	57.3)	128	19.5	(1.7	98.3)
				256	16.3	(1.3	98.7)
QR Matrix				*Extract*			
4	19.5	(0.3	99.7)	1	5.1	(99.6	0.4)
8	18.9	(0.3	99.7)	2	3.1	(98.9	1.1)
16	17.7	(0.3	99.7)	4	3.2	(95.5	4.5)
32	14.9	(0.3	99.7)	8	2.6	(92.4	7.6)
				16	2.0	(85.4	14.6)

Table 6.4: Percentage of application execution time that the disk(s) was (were) busy (on average).
This percentage is then broken into time spent logging messages and time spent checkpointing process states.

bandwidth required, for each application, in order to support the checkpointing of process states and the logging of messages. The information in these tables was taken from the same simulations that produced the graphs in Figures 6.1 and 6.2.

Total Disk Bandwidth Required (Mbytes/sec)			
# Nodes		# Nodes	
	Router		*FFT*
1	4.205	16	69.113
2	6.059	32	91.532
4	9.384	64	156.171
8	24.783	128	400.399
16	50.725	256	1408.379
	Placement		*Olfactory*
1	17.773	8	12.656
2	41.448	16	26.606
4	30.530	32	55.556
8	56.717	64	121.335
16	114.864	128	280.713
	Matrix QR		*Extract*
4	56.224	1	323.120
8	58.167	2	272.108
16	61.680	4	282.487
32	64.375	8	331.927

Table 6.5: The total disk bandwidth required to support asynchronous message logging (in Mbytes/sec).

6.2.1.3. Reducing the Number of Orphan Processes

In the proposed message logging algorithm (Section 3.5) we introduced the optimization of logging the *RSN* at the sender in order to reduce the number of orphan processes that result from a failure. This could be useful in a system that requires a probabilistic bound on the time it takes for the system to recover from a node failure (e.g. soft real-time). Any improvement in the number of orphan processes by this optimization relies on the assumption that the logging of the *RSN* at the sender proceeds faster and/or more robustly than the logging of messages

(and *RSNs*) to stable storage. This may be a reasonable assumption since logging the *RSN* at the sender requires the transmission of a very small message that can be sent immediately after the *RSN* is calculated. In contrast, logging to stable storage requires a copy of the entire application message and its dependency vector to be sent to stable storage.

Further, logging the *RSN* at the sender is a ‘‘free’’ operation if an acknowledgment is already needed by the reliable message delivery protocol. Ideally, if the *RSN* could be guaranteed to reach the sender then orphans would never occur as the result of a single process failure †. Also, reducing the probability that *any* orphan will occur is important because orphan messages can continue to propagate, creating more orphan processes, up until the time that recovery information (the new IST entry) has been received throughout the system (see Table 6.6).

Table 6.6 shows that, in our simulations, orphans are, in general, not likely to occur. This is because ample disk bandwidth, with no network congestion, is available to log messages as fast as they arrive. However, in the unlikely event that a single orphan does occur, *many* orphans can occur. Thus, in some systems, logging the *RSN* temporarily at the sender, in addition to logging it on stable storage, may be beneficial to avoiding an extended recovery session involving the rollback of multiple (orphaned) processes.

In *non*-soft-real-time systems this optimization can have a significant

---

† As in *synchronous* message logging algorithms which, by not allowing process execution to proceed until the *RSN* has been logged, guarantee that orphans will not occur in the event of a single process failure.

beneficial affect if any of the following conditions are true:

1) Techniques for reducing the number of messages logged to disk are used[Stro88, Wang92]. These techniques log *all* messages in local memory and delay the logging of (a smaller number of) messages to disk.

2) If it is desirable to log messages in groups as a single unit (as mentioned in[Stro85, Wang92]) instead of individually as we did.

3) There is periodic congestion at the disk or logged messages must cross the network and there is periodic network congestion.

In all of these cases the logging of messages to stable storage is delayed, further increasing the chances that one orphan (and thus many orphans) may occur as the result of a failure.

	No. Simulations	No. Orphan Processes		No. Simulations	No. Orphan Processes
	Placement			*Olfactory*	
16 nodes	136	none	64 nodes	138	none
	14	15		1	1,2,4,5,18,32,63
				2	16
				3	8
	Router			*FFT*	
8 nodes	136	none	64 nodes	141	none
	2	1		1	9,13,16,21
	12	7		2	8
16 nodes	136	none		3	7
	1	1,13,14			
	13	15			

Table 6.6: The number of orphans that result when a node, running a single process, fails.

6.2.1.4. Committing Outputs to the External World

Using asynchronous message logging techniques, a message can not be committed to the outside world until it can be guaranteed that the message will not be ‘‘undone’’ if a failure occurs. In other words, the message can be released once it can no longer be orphaned — the state intervals upon which it depends have all been logged to stable storage. Two optimizations have been proposed[Stro88, Elno92a] to reduce the time it takes to commit messages to the outside world.

With our assumptions and the applications we used the optimizations presented in[Stro88] would *not* commit outputs in a more timely fashion than our algorithm. In[Stro88] all messages are logged in local volatile memory. By associating ‘‘NeedCounts’’ with each message it can be determined that some (possibly most) messages will not actually need to be logged to stable storage. This requires some cooperative processing by the operating systems on different nodes. A second optimization uses a heuristic to predict the arrival of messages. If the heuristic is correct then ordering information also does not need be logged to stable storage. As they state, this does not change the basic idea that a message can not be committed to the outside world until it is guaranteed to be reproducible. The argument in [Stro88] is that, by reducing the number of messages logged to stable storage, messages are logged more quickly and thus the latency of committing external messages would be reduced. This is a reasonable assumption when the disk lags behind the application when all messages are being logged. In our simulations disk bandwidth was not a bottleneck and thus these optimizations would not improve the commit time of external outputs. Also, it should be noted that even if it takes some (unacceptable amount of) time to log messages to stable

storage, null logging is still not necessarily beneficial under our assumptions. This is because coordinated, and potentially significant, extra operating system processing is required for maintaining NeedCounts.

In[Elno92a] antecedence graphs instead of dependency vectors are maintained for each process. This allows each process to know the ordering of *all* events upon which it depends. In contrast, dependency vectors only record the most recent event in every other process upon which a process depends. Thus, if antecedence graphs are used, committing outputs to the external world requires only a single synchronous disk operation to record the graph of the process sending the output message and no multi-node coordination. For systems that require the timely (and periodic) commitment of output messages this may be an attractive technique. However, the additional processing required to update the graph and to send the incremental portion of the graph on each message is likely to be significantly larger than the equivalent activities for a dependency vector. Thus it is a reasonable expectation that, for systems that fit our assumptions and for the applications we used, operating system processing overhead would be *significantly* higher than for the basic asynchronous message logging algorithm we simulated. In this case it is quite possible that outputs will not be committed any faster (wall clock time) due to the overall increase in application latency. Both techniques require further examination.

6.2.2. Performance of Coordinated Checkpointing

In this section we evaluate the overhead during normal operation (with checkpointing) that synchronous and asynchronous coordinated checkpointing techniques impose on the system relative to a system without any checkpointing. We also compare the coordinated checkpointing approach to a somewhat ‘‘optimistic’’ implementation of independent checkpointing.

In all of our simulations checkpointing the process state involves copying the *modified* 1K byte pages of the process state into local dedicated buffer space. This state is then transferred to non-local stable storage (*incremental checkpointing*). In coordinated checkpointing techniques all messages in transit between members of the interacting set are flushed to their destinations and then written to stable storage as part of the destination’s checkpoint. The performance, during normal operation, of coordinated checkpointing and independent checkpointing (without message logging) techniques is highly dependent on the cost of saving process states to stable storage. Asynchronous coordinated checkpointing and all independent checkpointing techniques (including message logging) can benefit from the additional use of the copy-on-write optimization (see Section 4.6). A comparison between incremental checkpointing (which checkpoints only the modified process state) and copy-on-write checkpointing was made in [Elno92b]. The copy-on-write optimization was also examined, using our simulation testbed, in Section 5.6.3. These results are orthogonal to the performance comparison made in this section and can be used to reduce the performance overhead incurred by any one of a number of asynchronous recovery schemes.

During normal operation the coordinated checkpointing techniques presented

in this dissertation incur only a very small amount of additional overhead in order to maintain the *Buddies* list on behalf of each application process. Independent checkpointing techniques generally incur some amount of per-application-message operating system processing overhead[Bhar88, Wang92]. In the ‘‘optimistic’’ independent checkpointing algorithm, we do not simulate *any* additional overhead during normal operation †. Also, even though the detection of message transmission failures has been incorporated into our coordinated checkpointing algorithms (a technique which can not be used by independent checkpointing algorithms), we have not incorporated this optimization into the coordinated checkpointing algorithms simulated on the testbed. Instead, coordinated and independent checkpointing are assumed to use the same reliable communication protocol. Both of these points favor independent checkpointing in our performance comparison.

The simulated error recovery schemes are:

- *No_Chkpt* is the application without any checkpointing scheme. In the figures *No_Chkpt* is 1.0 and all other schemes are graphed relative to *No_Chkpt*.
- *A_Synch* is the presented asynchronous coordinated checkpointing algorithm (see Section 4.5).
- *F_Synch* is (fully-)synchronous coordinated checkpointing in which processes do not resume execution until the new checkpoint states

† This algorithm is an incomplete recovery scheme; in the event of a failure, processes would always have to roll back to their initial states because this algorithm does not provide any support to make it possible to determine if some other recovery line exists on stable.

have been committed to stable storage and the checkpoint session has completed (see Section 4.3).

- *P_Synch* is (partially-)synchronous coordinated checkpointing where processes are allowed to resume execution as soon as the interacting set has been found.
- *Indep* is an ‘‘optimistic’’ version of independent checkpointing in which processes are checkpointed once per checkpoint interval, but no additional overhead (e.g. message sequence numbers and message logging) is simulated.

In all of the simulations for each of the recovery schemes processes start out with ‘‘staggered’’ checkpoint intervals to eliminate any ‘‘cold start’’ effect where processes run nearly synchronous with one another despite the lack of any external (hardware or software) synchronization.

Figures 6.3 and 6.4 show the overhead of several recovery schemes in terms of increased application execution time for the applications *fft*, *Router*, *Placement*, *Matrix QR*, *Extract*, *Olfactory*. The checkpoint interval is 14600000 instructions for *Extract*, and 4600000 instructions for all other applications. The x axis indicates the number of processes that constitute the application and ranges from 1 to 256 nodes (depending upon the application). The problem size is held constant while the applications are scaled. Unlike Figures 6.1 and 6.2 the y axis (the application’s execution time with a particular checkpointing algorithm relative to the application’s execution time without any checkpointing) is not given in logarithmic scale.

Tables 5.1 and 6.2 show statistics for each application in terms of

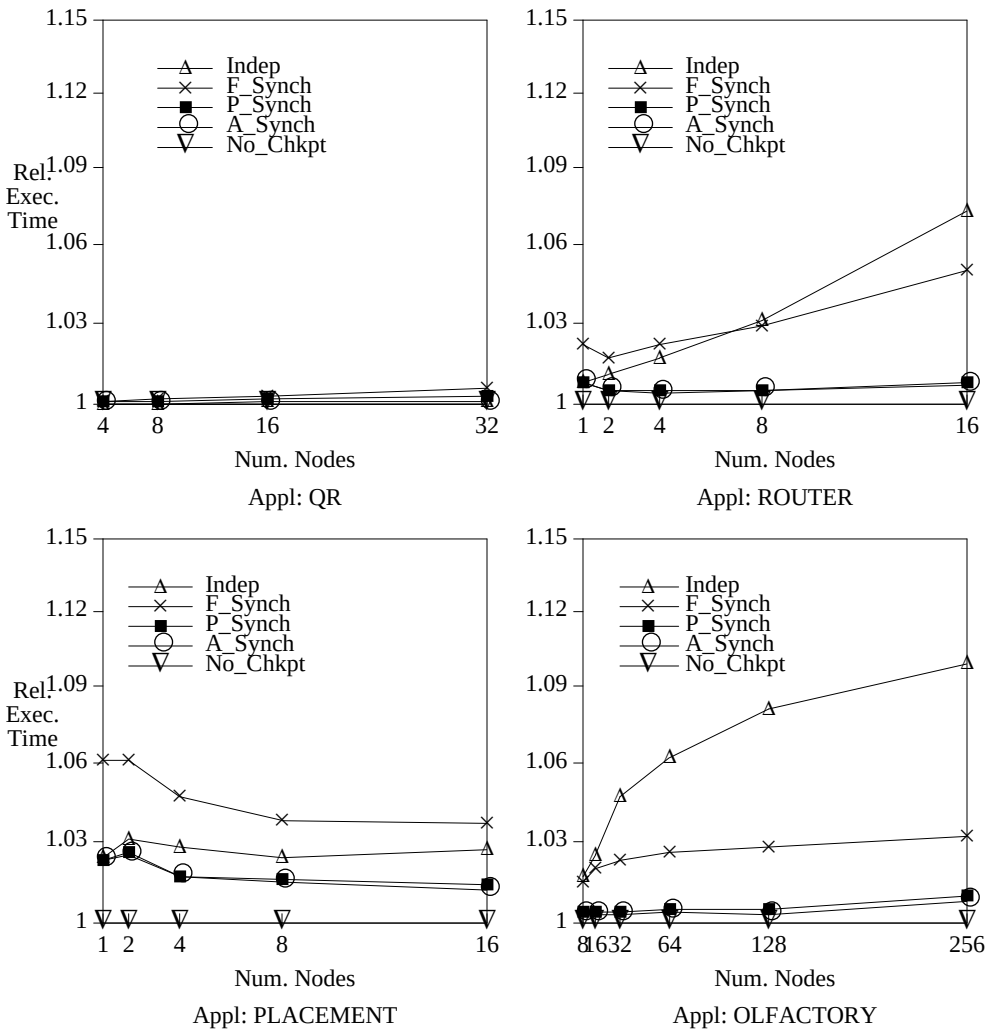

Figure 6.3: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints taken every 4600000 instructions.

communication frequency, total number of application messages sent and message volume. For all of the graphs in Figures 6.3 and 6.4 the disk and network bandwidths are (more than) sufficient to support both the applications and the simulated error recovery schemes, such that they do not contribute to any increase in the application's execution time. Sections 6.2.2.1 and 6.2.2.2, below, briefly

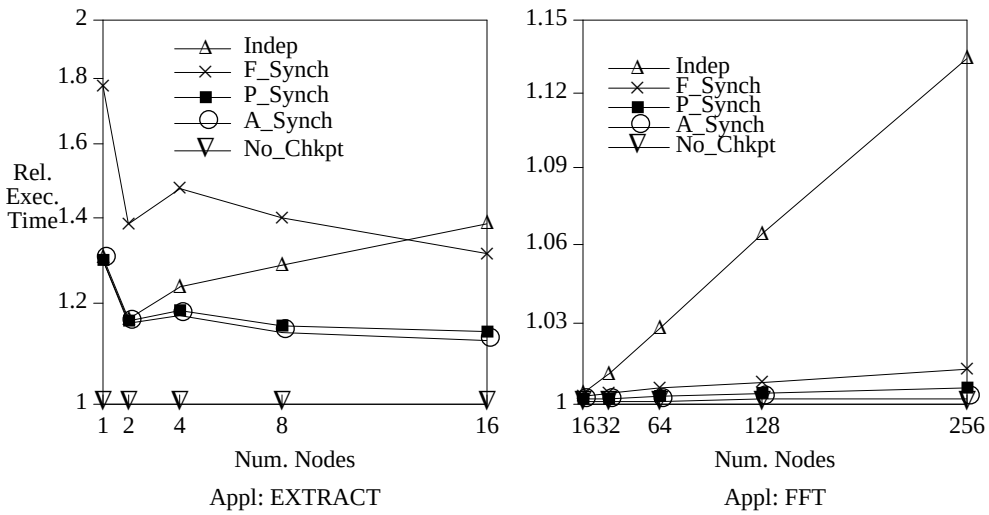

Figure 6.4: Continuation of Figure 6.3 (Note: with Extract checkpoints are taken every 14600000 instructions, instead of 4600000 instructions, and the y axis is different).

examine the network and disk bandwidth requirements of asynchronous coordinated checkpointing.

Table 6.7 indicates, for a given checkpoint frequency, the number of checkpoints taken by coordinated checkpointing schemes relative to *Indep*. By having processes checkpoint in groups, a checkpoint session is initiated by the first process that completes its checkpoint interval — resulting in checkpoints being taken, on average, more frequently than is dictated by the checkpoint interval. This table shows that, as the application is scaled, processes protected by coordinated checkpointing checkpoint more and more often compared to those protected by independent checkpointing. If the checkpoint interval can be tuned on a per application basis then this affect can be reduced. Table 6.7 also indicates the average size of the modified process state checkpointed by *A_Synch* and *Indep*; the checkpoint states are smaller for *A_Synch* because processes checkpoint more

	Checkpointing Statistics								
	Number of hypercube nodes								
	1	2	4	8	16	32	64	128	256
	Number of *A_Synch* Checkpoints / Number of *Indep* Checkpoints								
Extract†:	1.00	1.01	1.05	1.08	1.18	-	-	-	
OLF:	-	-	-	1.02	1.05	1.00	1.15	1.04	1.11
FFT:	-	-	-	-	1.05	1.12	1.18	1.20	1.48
QR:	-	-	1.01	1.03	1.16	1.21	-	-	-
Place:	1.14	1.13	1.24	1.74	1.81	-	-	-	-
Router:	1.00	1.03	1.10	1.11	1.18	-	-	-	-
	Avg. Size of Checkpoint State (1K Byte Pages) — *A_Synch* includes msg Qs								
Extract† *(Asy)*:	2.90K	1.60K	1.83K	1.55K	1.24K	-	-	-	-
(Ind):	2.86K	1.57K	1.85K	1.62K	1.42K	-	-	-	-
OLF *(Asy)*:	-	-	-	15.86	18.70	21.86	21.69	24.58	24.23
(Ind):	-	-	-	16.34	19.70	22.14	24.41	25.68	26.54
FFT *(Asy)*:	-	-	-	-	2.26	2.45	2.71	3.15	3.33
(Ind):	-	-	-	-	2.31	2.56	2.97	3.55	4.23
QR *(Asy)*:	-	-	1.16	1.31	1.52	1.88	-	-	-
(Ind):	-	-	1.15	1.30	1.58	2.07	-	-	-
Place *(Asy)*:	69.71	75.05	55.20	43.45	41.39	-	-	-	-
(Ind):	69.25	76.78	58.63	59.18	57.40	-	-	-	-
Router *(Asy)*:	25.67	22.88	27.47	33.66	42.08	-	-	-	-
(Ind):	25.53	23.52	30.00	37.45	48.99	-	-	-	-
	Avg. (Weighted) Size of Interacting Set (*A_Synch*)								
Extract† 2:	0.89	0.11	x	x	x	x	x	x	
4:	0.54	-	0.46	x	x	x	x	x	x
8:	0.38	-	-	0.62	x	x	x	x	x
16:	0.20	-	-	-	0.80	x	x	x	x
OLF 8-128:	size of interacting set = number of nodes								
256:	0.00	-	-	-	-	0.03	-	-	0.97
FFT 16:	0.34	-	-	-	0.66	x	x	x	x
32:	0.06	-	-	-	-	0.94	x	x	x
64:	0.10	-	-	-	-	-	0.90	x	x
128:	0.17	-	-	-	-	-	-	0.83	x
256:	0.20	-	-	-	-	-	-	-	0.80
QR,Place,Router:	size of interacting set = number of nodes (2, 4, 8, 16, or 32)								

Table 6.7: Checkpoint statistics for applications *Extract†* (checkpoint interval of 14600000 instructions) and *Olfactory*, *fft*, *QR*, *Placement*, *Router* (checkpoint interval of 4600000 instructions). Section one shows the number of checkpoints taken under *A_Synch* relative to the number of checkpoints taken with *Indep*. Section two gives the size of the checkpoint state for both schemes and Section three gives the average size of the interacting set (weighted by the total number of nodes) under *A_Synch*.

often. This mitigates, somewhat, the fact that processes checkpoint more often with coordinated checkpointing techniques. Lastly, the table shows the average weighted size of the interacting set in a checkpoint session. From this information we can see that, for the given checkpoint frequency, interacting sets tend to contain *all* of the processes in the application or, less frequently, just one of the processes.

Looking at Figure 6.3, none of the simulated error recovery schemes exhibit much overhead with QR. As can be seen in Table 6.7, QR has a very small working set: only one to two pages (per process) are checkpointed to stable storage and thus overhead is extremely low (less than one percent).

Router has significantly more state than *QR*, but overhead remains below ten percent for all simulated recovery schemes. A surprising result is that *Indep*, for 8 and 16 processes, performs worse than even *F_Synch*, the ‘‘fully synchronous’’ coordinated checkpointing scheme. It has generally been believed that independent checkpointing is a desirable feature, since each process need only be suspended the length of time it takes to copy its state — requiring no coordination with other processes. However, it is precisely this lack of coordination that results in greater overhead. As *Router* scales it becomes more communication intensive (see Table 6.2). As a result there is a greater probability that a process *P*, expecting a message from a process *Q*, will have to wait longer for that message because *Q* is in the midst of a checkpoint. With coordinated checkpointing *P* and *Q* have a high probability of checkpointing together (see Table 6.7) and thus overlapping much of their checkpoint overhead.

The impact of coordination upon the total overhead is quite significant since in *F_Synch* processes are suspended until all checkpoints in the interacting set have

been written and committed to disk, while with *Indep* each process is only suspended the length of time it takes to copy a single volatile checkpoint in local memory. In fact, in the ‘‘best case’’ our optimistic implementation of independent checkpointing would always perform at least as well as *A_Synch*, if we just use process suspension time as a predictor of checkpoint overhead. However our simulations show that this is erroneous and that it is in fact desirable for processes to coordinate their checkpointing. It is interesting to note that performance is good despite the fact that, with coordinated checkpointing, processes checkpoint more often than the checkpoint interval dictates. Further, table 6.7 shows that this shortening of the *actual* checkpoint interval gets worse as the application scales; the first process to reach the end of the checkpoint interval that initiates a checkpoint session — forcing other processes in the interacting set to checkpoint early. Nonetheless *A_Synch* overhead grows only slightly as the application scales and, if one can fine tune the checkpoint interval, this overhead can be reduced.

With *Placement* checkpointing overhead peaks at six percent for *F_Synch* and decreases as the application scales. All other schemes exhibit less overhead and show a similar trend. The reason for this is that the average number of dirty pages checkpointed to stable storage decreases as the application scales, which results in a stable or reduced level of overhead. This is also true for *Extract*, whose graph is similar to *Placement*. *Placement* is not communication-intensive compared to *Router* (see Table 6.2) and subsequently *Indep* performs better than *F_Synch* but still not as well as *P_Synch* and *A_Synch*.

Overhead with *Olfactory* is less than ten percent for all recovery schemes. *A_Synch* and *P_Synch* have very low overhead, and *Olfactory*’s rather small state

results in low overhead for *F_Synch*. Despite the fact that *Olfactory* is not nearly as communication-intensive as *Router*, *Indep* overhead goes up as the application scales and is always worse than even *F_Synch*. The overhead levels with *fft* are comparable. Overhead for the coordinated checkpointing schemes is very small due to *fft*'s tiny working set (see Table 6.7).

Extract's working set is 100 times the size of the other applications and this results in overhead of at least twelve percent and almost eighty percent in the worst case (*F_Synch* with one process). The checkpoint interval for *Extract* is 14600000 instructions and is limited by our assumption of 3 Mbytes/sec disk bandwidth per node; The disk can not handle a smaller interval. *Extract* is the least communication-intensive application but its large working set still results in *Indep* performing worse than *F_Synch* when run with 16 processes.

In all of the these simulations *A_Synch* and *P_Synch* performed almost the same. The main difference between the two algorithms is that with *P_Synch* processes are suspended until the interacting set is located. Thus process suspension time is dependent on network delays and the speed with which the remote operating system can respond to checkpoint requests. *A_Synch* overhead is independent of non-local activities. There are several reasons why this difference does not show up in our simulations. First, the operating system is modeled as a separate node *and* performs no duties aside from message routing and checkpointing — resulting in a quick response to checkpoint requests. Also, the network is very lightly loaded since only one application is running and there is no paging — resulting in fast message transmission. Thus the interacting set can be located quickly — amounting to less than one percent of total application runtime

in our simulations. However, we believe that independence from network delays and remote node computation is desirable. *A_Synch* will always perform at least as well as *P_Synch* with the only cost being an increase in algorithmic complexity during the checkpointing phase that locates the interacting set. Also *A_Synch* should scale well, while if network delays can be arbitrarily long *P_Synch* is not likely to scale as well. However, we do note that, if the network is congested, the application will take longer to execute and thus the *percentage* of overhead contributed by the error recovery scheme might *not* actually increase. We examine the performance of these recovery schemes under increased checkpoint frequency in Section 6.2.2.3.

6.2.2.1. Overhead Due to Network Bandwidth

Coordinated checkpointing techniques, as proposed in this dissertation, do not require additional network bandwidth during normal operation. In fact, if the detection of message transmissions errors is incorporated into the checkpointing algorithms as we have proposed, then less network bandwidth is needed compared to many alternative methods of implementing a reliable message delivery protocol (see Section 2.4). However, coordinated checkpointing does requiring the sending of many recovery scheme messages during checkpointing (and recovery) sessions. Table 6.8 indicates the total number of messages sent in each of the asynchronous coordinated checkpointing simulations. These totals are broken down into the percentage of application and recovery scheme messages. Recovery scheme messages make up anywhere from 0.5 to 23.4 percent of the total messages sent for *Router*, *Placement*, *fft*, *QR*, and *Olfactory* (excluding the single process case for

No. Nodes	Total Messages	Percent Recovery	Percent Application	No. Nodes	Total Messages	Percent Recovery	Percent Application
Router				*FFT*			
1	176	85.8	14.2	16	38235	16.1	83.9
2	67664	0.5	99.5	32	88088	9.0	91.0
4	127464	0.4	99.6	64	202268	4.9	95.1
8	327825	0.4	99.6	128	461165	2.7	97.3
16	718965	0.5	99.5	256	1047493	2.1	97.9
Placement				*Olfactory*			
1	873	0.0	1.0	8	3688	23.4	76.6
2	16549	4.0	96.0	16	8994	16.6	83.4
4	17544	4.6	95.4	32	21251	11.9	88.1
8	39567	8.8	91.2	64	50641	11.3	88.7
16	142084	7.7	92.3	128	115362	9.2	90.8
				256	262696	8.9	91.1
Matrix QR				*Extract*			
4	56757	2.5	97.5	1	193	57.5	42.5
8	56901	2.8	97.2	2	576	42.9	57.1
16	57197	3.2	96.8	4	1940	32.8	67.2
32	57445	3.5	96.5	8	4469	33.4	66.6
				16	11590	31.3	68.6

Table 6.8: Total messages sent in each simulation. Also given is the percentage of this total which is attributable to the error recovery algorithm (asynchronous coordinated checkpointing) and the percentage attributable to the application.

Router). The percentages are higher for *Extract* where the total number of messages sent is very low since *Extract* is very communication-unintensive. In general, the percentage of recovery scheme messages tends to decrease as the applications are scaled. No attempt to decrease or optimize the number of messages sent during checkpoint or recovery sessions has been made since these statistics have not proved to have any impact upon application or system performance under our assumptions.

Total Disk Bandwidth Required (Mbytes/sec)			
# Nodes		# Nodes	
	Router		*FFT*
1	4.434	16	1.142
2	1.846	32	1.415
4	1.893	64	1.919
8	3.003	128	2.838
16	5.498	256	5.166
	Placement		*Olfactory*
1	9.702	8	1.425
2	13.055	16	2.444
4	5.805	32	4.290
8	9.206	64	8.515
16	14.222	128	16.087
	Matrix QR		*Extract*
4	0.214	1	326.904
8	0.252	2	283.597
16	0.343	4	298.140
32	0.487	8	355.830

Table 6.9: The *total* disk bandwidth required to support asynchronous coordinated checkpointing (in Mbytes/sec).

6.2.2.2. Overhead Due to Disk Bandwidth

Since most of the applications have small process checkpoint states (see Table 5.1) the disk bandwidth requirements of asynchronous coordinated checkpointing are very low (see Table 6.9). *Extract* is the only exception (where each process checkpoint state is roughly 2.7 Mbytes). For *Extract* the disk bandwidth requirements of asynchronous coordinated checkpointing are quite similar to asynchronous message logging; for the rest of the applications

coordinated checkpointing requires *significantly* less disk bandwidth than message logging.

6.2.2.3. Simulations with Increased Checkpoint Frequency

To look more closely at the potential difference between the partially-synchronous (*P_Synch*) and asynchronous (*A_Synch*) coordinated checkpointing algorithms, within the framework of our simulator, we shortened the checkpoint interval to 50K instructions and then 5K instructions for the following four applications: *Matrix QR*, *Router*, *fft*, and *Olfactory*. This increases the total overhead of checkpointing and shows the contribution of interacting set determination time to this overhead. These simulation results are shown in Figures 6.5 and 6.6.

QR (with 50K instructions checkpoint interval) still has quite low overhead — no more than 4.5 percent (*F_Synch* with 16 processes) — due to a small process state. *Indep* and *A_Synch* have identical overhead at .4 percent for all *QR* simulations. *P_Synch* overhead increases somewhat, with the contribution of interacting set determination being .2, .7, 1.0, and 2.0 percent as the application scales through 4, 8, 16 and 32 nodes. When the checkpoint interval decreases to 5K instructions *Indep* stays at a constant 3.3 percent while *A_Synch* overhead starts at 3.9 percent for 4 nodes and goes up to 5.0 for 32 nodes. This is because 1) the small checkpoint interval results in smaller interacting sets (54 to 91 percent have only two processes) and thus there is less overlap of checkpointing overhead and more interference with normal processing and 2) the fact that processes checkpoint more often than the checkpoint interval dictates. *If* the checkpoint interval can be

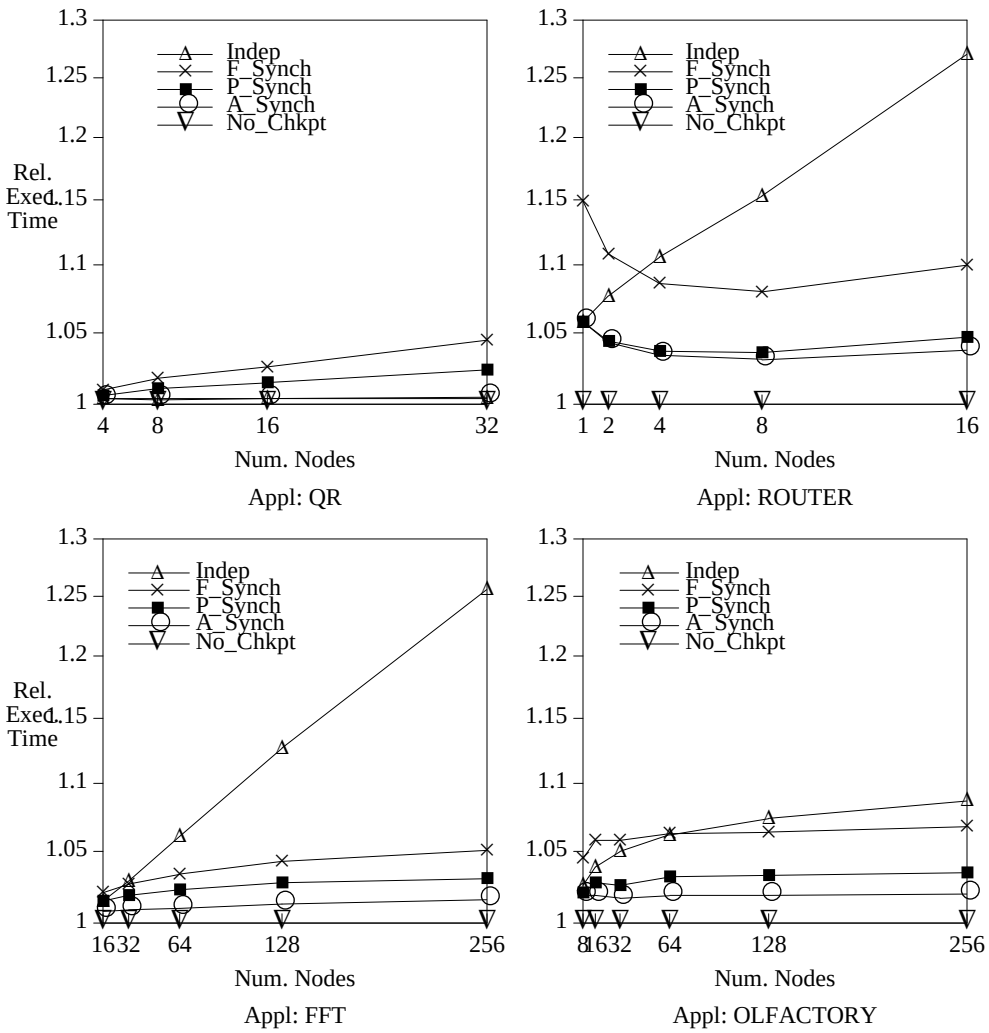

Figure 6.5: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints are taken every 50000 instructions.

tuned then, in the ‘‘worst’’ case, as the checkpoint frequency is increased, asynchronous coordinated checkpointing will ‘‘become’’ independent checkpointing (with interacting sets of only one process) and the overhead of the two algorithms will be comparable for this application.

As the checkpoint frequency increases *Router* results remain similar to our

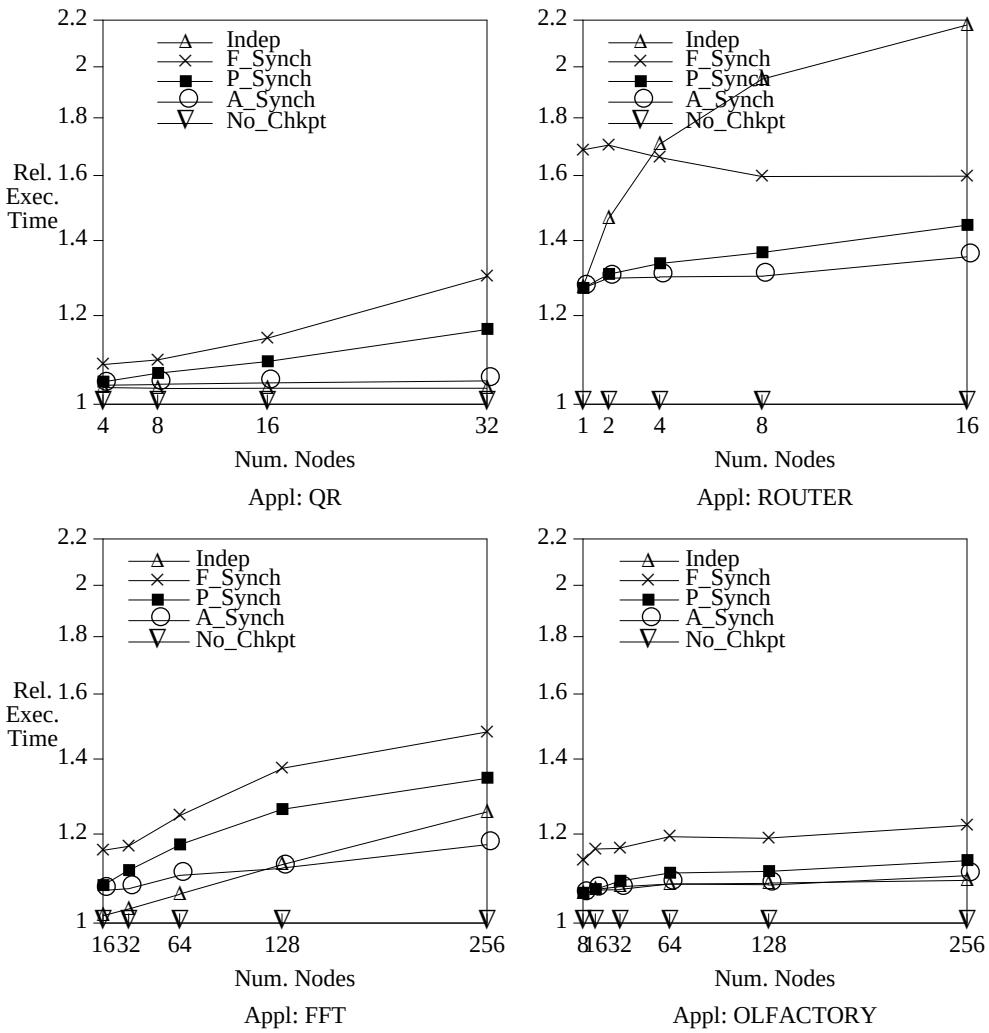

Figure 6.6: Application execution time with coordinated and independent checkpointing relative to application execution time without any recovery scheme. Checkpoints are taken every 5000 instructions.

original simulation results. Even with a 5K instruction checkpoint interval the interacting set contains all processes 87 to 100 percent of the time due to the high communication frequency of the *Router* application. Hence *A_Synch* performs best in all cases and *Indep* has the highest level of overhead. The overhead contribution of interacting set determination is at most 0.7 percent (50K interval with 16

	Checkpointing Statistics (Checkpoint interval of 5000 instructions)								
	Number of hypercube nodes								
	1	2	4	8	16	32	64	128	256
	Number of *A_Synch* Checkpoints / Number of *Indep* Checkpoints								
OLF:	-	-	-	1.00	1.01	1.00	1.03	1.02	1.11
FFT:	-	-	-	-	1.05	1.12	1.18	1.20	1.48
QR:	-	-	1.01	1.03	1.16	1.21	-	-	-
Router:	1.00	1.03	1.10	1.11	1.18	-	-	-	-
	Avg. Size of Checkpoint State (1K Byte Pages) — *A_Synch* includes msg Qs								
OLF *(Asy):*	-	-	-	2.00	2.01	2.01	2.02	2.04	
(Ind):	-	-	-	2.00	2.01	2.01	2.02	2.04	2.53
FFT *(Asy):*	-	-	-	-	2.26	2.45	2.71	3.15	3.33
(Ind):	-	-	-	-	2.31	2.56	2.97	3.55	4.23
QR *(Asy):*	-	-	1.16	1.31	1.52	1.88	-	-	-
(Ind):	-	-	1.15	1.30	1.58	2.07	-	-	-
Router *(Asy):*	25.67	22.88	27.47	33.66	42.08	-	-	-	-
(Ind):	25.53	23.52	30.00	37.45	48.99	-	-	-	-
	Avg. (Weighted) Size of Interacting Set (*A_Synch*)								
OLF 8:	0.71	0.19	0.09	0.01	x	x	x	x	x
16:	0.54	0.24	0.14	0.06	0.02	x	x	x	x
32:	0.40	0.30	0.09	0.10	0.09	0.02	x	x	x
64:	0.24	0.34	0.14	0.10	0.12	0.05	0.01	x	x
128:	0.17	0.33	0.15	0.09	0.09	0.09	0.06	0.02	x
256:	0.11	0.39	0.12	0.07	0.06	0.06	0.14	0.04	0.01
FFT 16:	0.53	0.17	0.26	0.04	-	x	x	x	x
32:	0.28	0.17	0.17	0.17	0.17	0.04	x	x	x
64:	-	-	-	0.31	0.18	0.17	0.34	x	x
128:	-	-	-	-	-	-	-	1.00	x
256:	0.01	-	-	-	-	-	-	-	0.99
QR 4:	0.89	0.11	x	x	x	x	x	x	
8:	0.54	-	0.46	x	x	x	x	x	x
16:	0.38	-	-	0.62	x	x	x	x	x
32:	0.20	-	-	-	0.80	x	x	x	x
ROUTER: 2:	-	-	x	x	x	x	x	x	x
4:	-	-	-	x	x	x	x	x	x
8:	-	-	-	-	x	x	x	x	x
16:	-	-	-	-	-	x	x	x	x

Table 6.10: Checkpoint statistics for *Olfactory*, *fft*, *QR*, *Router* (checkpoint interval of 5000 instructions). Section one shows the number of checkpoints taken under *A_Synch* relative to the number of checkpoints taken with *Indep*. Section two gives the size of the checkpoint state for both schemes and Section three gives the average size of the interacting set (weighted by the total number of nodes) under *A_Synch*.

processes) and 7.2 percent (5K interval with 16 processes) which is 15 and 17 percent of the total *P_Synch* overhead, respectively.

With *FFT* approximately half of *P_Synch*'s overhead (1 percent with the 50K checkpoint interval and 1 to 11 percent with the 5K interval) comes from the time it takes to determine the interacting set. The large relative percentage is due to *FFT*'s small process state. With a 5K checkpoint interval *A_Synch* has smaller interacting set sizes - especially for 16, 32 and 64 nodes, but *A_Synch* exhibits overhead comparable to *Indep* with 128 nodes and less with 256 nodes. *Indep*'s steeper curve is probably due to the fact that *FFT* becomes increasingly communication intensive as the application scales.

Olfactory is not very communication-intensive compared to *Router* and *FFT*. So while *Indep* does not perform as well as the coordinated checkpointing schemes with a checkpoint interval of 50K instructions, the breakdown of the interacting set size and larger number of checkpoints experienced by *A_Synch* evens up the overhead with *Indep* using a 5K checkpoint interval. Locating the interacting set contributes .5 to 2.5 percent to *P_Synch* overhead for 50K interval and .5 to 3.4 percent for 5K interval.

Table 6.10 is the same as Table 6.7 but for a checkpoint interval of 5000 instructions. This table indicates the number of checkpoints taken by coordinated checkpointing schemes relative to *Indep*; the average size of the modified process state checkpointed by *A_Synch* and *Indep*; and the average weighted size of the interacting set in a checkpoint session (which has broken down considerably compared to the checkpoint interval of 4600000 instructions).

While *Indep* exhibits overhead comparable to *A_Synch* for applications *QR*

and for the 5K checkpoint interval simulations using *Olfactory*, *A_Synch* has a more consistent level of overhead that appears to scale well even as the checkpoint interval is significantly decreased. This indicates that it may be useful in soft real-time systems due to rather predictable overhead levels. While *P_Synch* and *A_Synch* perform nearly the same in our original (46M instructions checkpoint interval) simulations, decreasing the interval shows that finding the interacting set contributes anywhere from a negligible amount to 11 percent of overhead on its own. We believe that *A_Synch* is a ‘‘robust’’ implementation of coordinated checkpointing whose performance will not degrade much with network congestion or scaling. If the checkpoint interval is short enough that most interacting sets are small relative to the number of processes in the application, then coordinated checkpointing is likely to perform similar to *Indep*.

As mentioned previously, with *A_Synch* the sending of a message may have to be temporarily delayed if the (checkpointing) process is attempting to send the message outside of the (locally known) interacting set, before that interacting set has been found (see Section 4.5.1). In our simulations very few application messages, generally less than 0.1 percent, 0.3 in the worst case, needed to be temporarily blocked. Since all of the six applications used nonblocking sends, this delay only affected how soon the message could be received by the destination process. This appears to have had an insignificant impact on the total overhead of asynchronous coordinated checkpointing.