- Staggering the disk writes is clearly a good idea, as it drastically reduces the processors' serialization at the disks. When the program's behavior is not characterized by synchronization or large messages, this staggering can be effected by the Network Sweeping algorithm much more so than the other algorithms. When the main memory checkpoint fits into physical memory, any algorithm can stagger the writes, regardless of the behavior of the target program.

- Although we have not yet implemented the CLL algorithm, the results of the multiprocessor implementation suggest that it would be doubly beneficial to any of the checkpointing algorithms. First, since it imposes a small initial stop time, it would not be penalized by programs with tight synchronization or large messages. Second, since it is an algorithm that naturally spreads out the disk writes over time, it should help in reducing contention for the disk. Thus, a combination of the Network Sweeping algorithm and the CLL algorithm should prove ideal for multicomputer checkpointing.

- Compression is good for certain programs, and bad for certain programs. Users should be able to predict when compression will benefit their programs, and accordingly should only use it in those cases.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The preceding chapters have presented a thorough discussion of checkpointing algorithms and speed optimizations for multiprocessors and multicomputers. With the implementations in Chapter 4, we have shown that we can provide efficient fault-tolerance for MIMD architectures, thereby enhancing their general usability. Specifically, we draw the following conclusions about checkpointing algorithms and their speed optimizations:

5.1.1 Algorithms

For multiprocessor checkpointing, there is only one basic algorithm for checkpointing, which proves successful in providing fault-tolerance on the Firefly. For multicomputer checkpointing, we studied three basic algorithms: Sync-and-Stop, Chandy-Lamport, and Network Sweeping. The latter two algorithms outperform the Sync-and-Stop algorithm in all cases as they do not freeze up the entire system like the Sync-and-Stop algorithm does. The Network Sweeping algorithm is more efficient than the Chandy-Lamport algorithm when the target program benefits from staggering the checkpoints, as Network Sweeping staggers them to a greater degree. As the number of processors

grows, the Network Sweeping will further outperform the Chandy-Lamport algorithm, as it sends far fewer marker messages.

5.1.2 Reducing Checkpoint Time

As stated in the Introduction, the main way to decrease checkpoint time is to reduce the amount of data written to disk. Previous research [FB89, EJZ92] shows that incremental checkpointing is one way to reduce the checkpoint size. Our implementation shows that on-line compression is another alternative. In our multiprocessor implementation, the CLL algorithm reduces the checkpoint time from the other optimizations by starting the disk writes as soon as possible, instead of waiting for a complete copy of the checkpoint to exist in main memory.

The other optimizations all increase the checkpoint time in order to decrease the overhead or latency. The experimental results reported in Chapter 4 show that this is a reasonable tradeoff. Except for when the optimizations thrash virtual memory, they never increase the checkpoint time by a large factor. Even when writing 128 megabytes of checkpoint data to disk as in the Iterative Equation Solver of Section 4.2.9, the checkpoint time never increases to over two minutes. As we envision that users will checkpoint in sparse intervals, such as once every ten minutes or once an hour, these checkpoint times are indeed adequate.

5.1.3 Reducing Checkpoint Overhead

Checkpoint overhead is the most important component of a checkpointer's speed, as it measures how much the checkpointer affects the target's running time. The main-memory optimization, especially as implemented in the CLL algorithm, is the most effective at reducing the checkpoint overhead, as it substitutes memory-to-memory copy time for disk write time as the major component of the overhead. In multicomputer checkpointing, staggering the disk writes to reduce contention on shared resources is also a good way of

reducing overhead. The Network Sweeping algorithm allows for effective staggering with certain types of program behavior (loose synchronization, small messages). Main memory checkpointing allows for complete staggering when the checkpoint fits into physical memory. The CLL algorithm should be effective in staggering disk writes in all cases.

5.1.4 Reducing Latency

Latency is a measure of the real-time invasiveness of a checkpointer. The copy-on-write and CLL optimizations both use a machine's virtual memory hardware to intersperse the taking of the checkpoint with the running of the target program. Both optimizations are successful in interrupting the target only for small, fixed periods of time. In multiprocessor implementation, the latency was kept under a tenth of a second. The copy-on-write and CLL optimizations should both improve multicomputer latency as well.

5.1.5 Uses

Both the multicomputer and multiprocessor implementations show that our checkpointing algorithms can be useful for fault-tolerance. For example, suppose a user's iPSC/860 program runs on 32 nodes for a week, and uses four megabytes per node. Regardless of the algorithm used and the behavior of the program, the checkpoint time and overhead will be less than two minutes. Therefore, if the user checkpoints once every hour, then the program's running time will be increased by no more than five and a half hours. If the checkpoint can be compressed into main memory, then the data from Figure 4.15 suggests that the checkpoint overhead will be around twenty seconds per checkpoint. This means that the running time of the program will be increased by under an hour, which is less than 0.6% of the total running time.

For job-swapping, the data presented in Chapter 4 suggest that our checkpointers should be useful for coarse-grained job-swapping. On the iPSC/860, this is very beneficial, as there are no other facilities provided by NX/2 for

job-swapping. Without ICKP, users are very reluctant to try running long programs, as most machines are widely shared, and hogging a significant portion of processors for several hours is usually discouraged. ICKP makes programs like the genetic algorithm tool easier to use in a considerate manner as it enables them to be swapped off the iPSC/860 during periods of high use.

Fine-grained job-swapping requires low latency. On multicomputers, a combination of the network sweeping algorithm and the CLL algorithm can be used to implement such a swapping mechanism. Once the target address spaces have been protected, the processor can start serving a process of the second program. Protected memory pages can be moved to the CLL buffer and then to backing stores on demand. Thus, the latency of job-swapping is similar to the checkpoint latency. The overhead depends on memory requirements.

Program migration is much like job-swapping, except that the checkpoint need not be written to disk. Instead it must be sent as a message to the second machine. This is a much faster operation than disk writing. For example on the Paragon, the bandwidth of messages is 200 Mbytes per second, whereas the speed of CFS disk writes is around four Mbytes per second. These fast speeds combined with main memory or CLL checkpointing should reduce checkpoint time and overhead significantly enough for checkpointing to be an effective means of program migration.

5.2 Future Work

We anticipate doing two kinds of research in the future: modifications and extensions of work done in this dissertation, and working on new types of checkpointing. In the former case, we plan to implement incremental checkpointing on all of our platforms, and to install the copy-on-write and CLL optimizations into the multicomputer implementation. This will give us a more complete picture of how the checkpointing optimizations work and interact with each other.

More importantly, we plan to port the iPSC/860 implementation to larger multicomputers, such as the Delta and Paragon, which can have up to 570 and 1024 processors respectively. Here, both file compression and incremental checkpointing will be imperative, as the amount of data comprising each global checkpoint will be staggeringly large—up to 16 gigabytes. Moreover, the difference between the Chandy-Lamport and Network Sweeping algorithms will become more evident.

Another platform on which to test checkpointing is *Shared Virtual Memory (SVM)* [LH89]. Here, the underlying machine is a multicomputer, which supports only message passing, but the programming environment implements shared memory using the page tables and fault handlers of each machine to manage the shared address space. Paging is performed from processor to processor instead of from processor to disk. Thus, a checkpointer can take advantage of the shared memory model to checkpoint as on a multiprocessor, even though the underlying machine is a multicomputer. Moreover, replicated pages on different processors may be identified by the checkpointer, and only one of them need be written as part of the checkpoint file. Thus, global checkpoints should be both smaller and simpler when taken as part of an SVM system than as a generic application on the multicomputer.

5.2.1 $N + 1$ Parity

$N + 1$ parity is an idea that has been used to add reliability to disk arrays [Gib90]. It can also be used to checkpoint a multicomputer system given the stipulation that only one processor fails at any point in time. The definition of $N + 1$ parity is as follows: Suppose there are N pages of memory, $P_1, P_2, \ldots P_N$, all of which are m bytes long. We designate the i-th byte of page P_j as P_j^i. The *parity page*, P_{N+1}, of these N pages is defined to be the bytewise exclusive or of P_1 through P_N. Specifically:

$$P_{N+1}^i = P_1^i \oplus P_2^i \oplus \ldots \oplus P_N^i \text{ for } 1 \leq i \leq m.$$

Now, if we define the set P to be $P_1 \cup \ldots \cup P_N \cup P_{N+1}$, then P has the following interesting property: Any subset of P with exactly N pages can be used to construct the remaining page. Specifically:

For any j between 1 and $N+1$, $P_j = P_1 \oplus \ldots \oplus P_{j-1} \oplus P_{j+1} \oplus \ldots \oplus P_N \oplus P_{N+1}$.

$N+1$ parity can be used for checkpointing in the following way: Suppose that for every processor p in a multicomputer, p's local checkpoint fits into main memory. Then, when taking a global checkpoint, each processor simply creates its main-memory checkpoint, and then cooperates with the rest of the processors to make the *parity checkpoint*. This is done by having each processor pad its main memory checkpoint so that all checkpoints are the same size, and then making all processors cooperate to take the bytewise exclusive or of their checkpoints. The result is the parity checkpoint, which is analogous to the parity page above. The parity checkpoint is written to disk, and the processors retain their main-memory checkpoints in physical memory. If a processor fails, then its checkpoint can be recreated from the other processors' main memory checkpoints and the parity checkpoint.

If each checkpoint is m bytes long and there are N processors in the system, then $N+1$ parity writes only m bytes to disk, whereas normal checkpointing writes Nm bytes. Thus, $N+1$ parity should improve checkpoint time by a factor of N. This is at the expense of needing enough physical memory for each processor to keep a copy of its checkpoint resident in memory.

We plan to implement checkpointing using $N+1$ parity on the Intel Delta and Paragon.

5.2.2 User-Defined Checkpointing

One of the ways to simplify the task of checkpointing, remove the restrictions imposed by the operating system, and reduce the amount of information written to disk is to allow users to give the checkpointer directives concerning their program's behavior. For example, users could mark certain places in their code as "safe" places to checkpoint. These would be places where, for

example, there are no open files or sockets. Moreover, users could mark certain pieces of memory as temporary—that is, at the safe places to checkpoint, these pieces of memory need not be saved as part of the checkpoint.

The assumption here is that programmers often make use of large chunks of temporary storage, which at certain points in a program do not contain meaningful data. For example, they may get periodically re-initialized from disk, or recalculated from other program data. Moreover, there is an assumption that while there might be points in a user's program where the restrictions imposed by the checkpointer are broken, these points might be small in duration, and thus with a few hints from the user, the checkpointer may avoid them. Finally, with multicomputer checkpointing, the user may prohibit checkpointing to occur while sending large amounts of data, thereby decreasing the potential size of the message log.

We plan to define programming constructs and libraries for user-defined checkpointing, first for uniprocessor checkpointing and then for multiprocessor and multicomputer checkpointing. To implement these, we anticipate having to rewrite the memory allocator, enabling it to mark certain chunks of memory as temporary, and to reallocate specific chunks of memory on demand (for recovery). Some work has already been completed to write this allocator.

5.2.3 Shared Single-Level Store

A more long-range result of work in checkpointing is to develop a programming environment based on the concept of a *shared single-level store*. That is, for main memory to be both persistent and sharable by different processes and users. In such a system, file caching is taken to an extreme—files are simply mapped into main memory, and are persistent and sharable because main memory is persistent and sharable.

We envision being able to implement a shared single-level store on a multicomputer system using Shared Virtual Memory and checkpointing ideas to make memory both sharable and persistent.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[AEL88] A.W. Appel, J.R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *The 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, Washington, May 1990.

[Ahu89] Mohan Ahuja. Repeated global snapshots in asynchronous distributed systems. Technical Report OSU-CISRC-8/89 TR40, Ohio State University, August 1989.

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[BAD+92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, Georgia, October 1983.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrman, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions of Computer Systems*, 7(1):1–24, Feb 1989.

[BJLM92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. I believe this is a DEC SRC Tech Report – Kai says it will also be an ASPLOS 92 paper., 1992.

[BWC89] T. Bell, I.H. Witten, and J.G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–589, December 1989.

[CJ91] Flaviu Cristian and Farnam Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):3–75, February 1985.

[CT90] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report TR 90-1101, Cornell University, February 1990.

[Dal90] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.