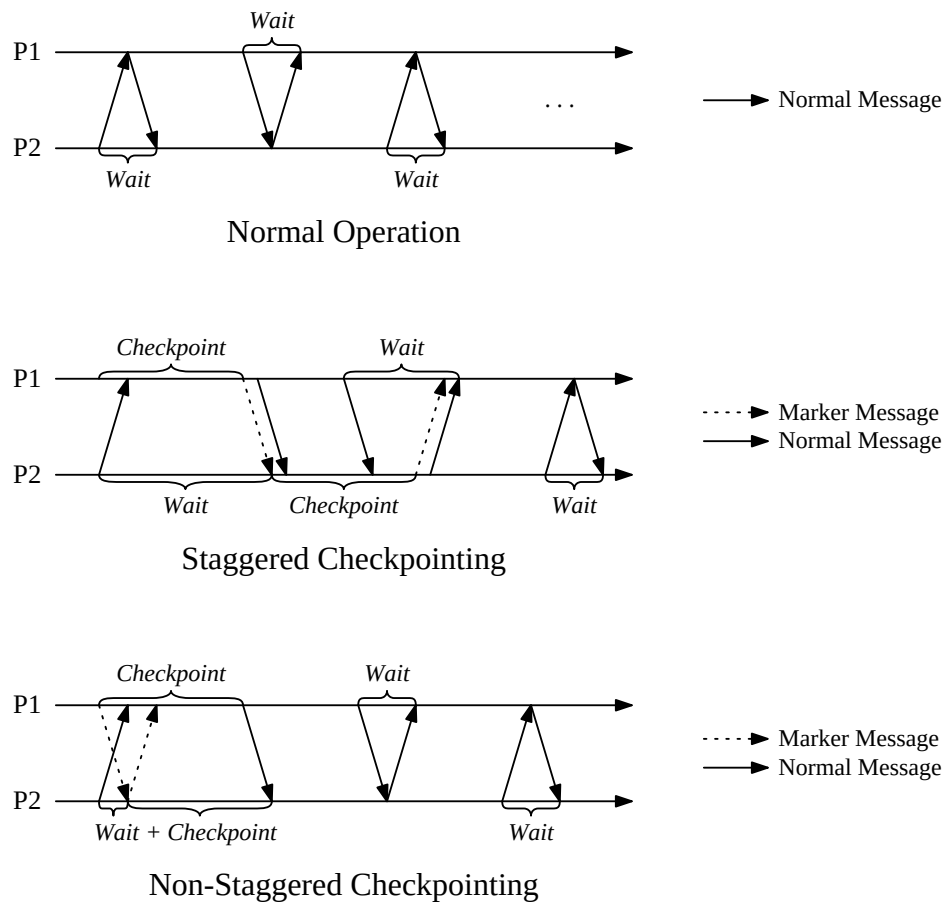

Figure 3.7: Staggering Checkpoints with a Synchronizing Pair of Processors.

pointed with a staggered algorithm, there can be cases when a processor has to wait for the other to finish taking its local checkpoint before it receives its acknowledgement. Thus, even though the checkpoints to disk are local, their duration can possibly cause overhead on another processor that is waiting for it to send a message. With non-staggered checkpoints, the possibility of this kind of overhead is diminished, as the local checkpoints occur roughly at the same point in time.

There are some situations where staggered checkpoints should cause less overhead than non-staggered ones. Consider the case when the number of processors greatly outnumbers the number of disks. For example, on the

iPSC/860, there may be up to 128 processors, but only four disks. If all the processors try to write their local checkpoints at the same point in time, then there is going to be heavy contention for the shared disks, and some processors are going to be delayed while waiting for other processors to write to disk. In this case, the staggered algorithms should cause less overhead than the non-staggered ones, since the writing of checkpoints is staggered. Moreover, the Network Sweeping algorithm should show the best performance because its spreads the checkpoints over a greater period of time than the Chandy-Lamport algorithm.

This last situation is not as important for taking main memory checkpoints. The reason is because once the checkpoint is copied to main memory, the target program is restarted and the checkpoint may be written to disk at time, not necessarily immediately. Thus, the processors may coordinate so that only one is writing to disk at a time, thereby eliminating any contention for the disks.

3.5 Compression

A rather obvious optimization to checkpointing is to use compression to decrease the number of bytes that get written to disk, thereby reducing the checkpoint time. File compression has seen widespread use in computer systems, as most user and computer generated files have enough structure for compression algorithms to shrink the file size by at least a factor of two. Thus, compression should also benefit checkpointing if the address spaces have the kind of structure of which compression algorithms can take advantage.

A concern of compression optimizations is the speed of the compression algorithm versus the amount of the checkpoint that gets compressed. If the compression algorithm is several times slower than a memory-to-memory copy, and it does not shrink the size of the checkpoint file considerably, then it is likely to increase both the checkpoint time and overhead rather than decrease them. To evaluate this tradeoff, we define the following quantities:

$$
\begin{aligned}
S &= \text{The size of the checkpoint file} \\
m(S) &= \text{The time to make a memory-to-memory copy of } S \\
c(S) &= \text{The time to compress } S \\
cf(S) &= \text{The size of } S \text{ compressed} \\
w(S) &= \text{The time to write } S \text{ to disk}
\end{aligned}
$$

The checkpoint times of sequential, main memory, and compressed checkpointing should then be as follows (This is assuming the main memory checkpoint fits in physical memory):

$$
\begin{aligned}
\text{Sequential:} &\quad w(S) \\
\text{Main memory:} &\quad m(S) + w(S) \\
\text{Compressed:} &\quad c(S) + w(cf(S))
\end{aligned}
$$

And the checkpoint overhead as follows:

$$
\begin{aligned}
\text{Sequential:} &\quad w(S) \\
\text{Main memory:} &\quad m(S) + \alpha w(S) \\
\text{Compressed:} &\quad c(S) + \alpha w(cf(S))
\end{aligned}
$$

In the above, α is a constant that denotes the overhead of concurrently writing a chunk of memory to disk. That is, even when a disk write is done at the same time as the execution of the target program, it affects the running of the target via cycle stealing and memory bus utilization. α is a measure of this overhead.

Thus, we expect compression to outperform sequential and main memory checkpointing whenever the gains of writing a smaller checkpoint file to disk outweighs the penalty of the compression time. That is, if the difference between $w(S)$ and $w(cf(S))$ is enough to offset the difference between $c(s)$ and $m(S)$, or $c(s)$ and 0.

If the main memory checkpoint does not fit into physical memory but the compressed checkpoint does, then it is obvious that compressed checkpointing should be better at decreasing overhead: No disk writes need be performed while the target threads are frozen and there need be no thrashing. If the compressed checkpoint does not fit into physical memory, then compression

can still be used to improve the checkpoint time of sequential and main memory checkpointing, by compressing and writing the checkpoint in chunks that will fit in main memory. All these cases are tested in our implementation.

Chapter 4

Implementations and Experiments

We have implemented checkpointing and recovery on a multiprocessor (the DEC Firefly) and a multicomputer (the Intel iPSC/860). These implementations encompass all algorithms and optimizations described in Chapters 2 and 3. The first three optimizations of Chapter 3 are implemented on the multiprocessor, and the rest are implemented on the multicomputer. The results of the experiments presented in this chapter demonstrate that we are able to implement checkpointing efficiently on both of these machines.

Before going into the details of the implementations, we reiterate our goals concerning the speed of checkpointing. First, we desire that the checkpoint time be near optimal. That is, it should be a small constant factor from optimal, like twice or three times. Second, the checkpoint overhead should be as low as possible. This means that the checkpoint has less effect on the target's running time. Finally, we would like to get latency down to a level where a user watching the program can always perceive that the target is getting work done. As stated in Chapter 1, we set this value at a tenth of a second.

4.1 Multiprocessor Implementation

We have implemented four checkpointing algorithms—sequential, main memory, copy-on-write, and CLL—as well as recovery on a DEC Firefly, an experimental shared-memory multiprocessor developed at the DEC System Research Center [TS87]. Fireflies consist of up to seven CVAX processors, each with a floating point unit and a direct-mapped 64 Kbyte cache attached to a central bus. The caches are snoopy caches, and thus listen to the shared bus to keep their caches coherent with the shared memory. The operating system for the Firefly is Taos [MS87], a version of Ultrix with threads and cheap thread synchronizations.

The Firefly upon which we tested our algorithms has four processors that share 16 Mbytes of physical memory. It also has a separate I/O processor that shares memory with the other four processors and has a separate bus connecting the Firefly to I/O devices and the outside world. In our system, the I/O processor is connected to a RD54 disk drive, which writes at a speed of 320 Kbytes per second.

The implementation is written in Modula-2+ [RLW85], an extension of Modula-2. The user interface to the checkpointing routines is a single call to *setup_checkpoint()* at the beginning of the user's program. This call may be used either to restore the program's execution to a saved recovery point, or to specify how long the checkpointer should wait before interrupting the program to take a checkpoint. Freezing the processors is provided by Taos through a system call. As stated in Section 2.1, in the absence of such a system call, one could freeze the processors either through interprocessor interrupts, or by protecting all the pages in memory to be "no access," and gaining control of the processors in the page fault handlers.

As mentioned in Section 3.2, one of the variables for the copy-on-write and CLL optimizations is the machine's page size. Although the actual page size on the Firefly's memory management unit is 512 bytes, we emulate pages sizes from 1K to 16K in our experiments.

4.1.1 Experiments with Merge Sort

For our initial experiments, we tested all four checkpointing algorithms on a parallel implementation of merge sort. We also checkpointed four other parallel programs: traveling salesman, matrix multiplication, pattern matching, and bubble sort. Since the results from the experiments with these programs are so similar to the results with merge sort, we only present merge sort in detail. The program sorts 250,000 indexed records, where the record size can be changed to modify the size of the program's address space.

In all experiments, the four processors of the Firefly are partitioned so that the target program uses three and the checkpointer uses one. This is to measure the maximal concurrency of our checkpointing methods. The checkpointer waits for the target to run for ten seconds, and then it takes one complete snapshot. For the initial results, the page size is eight Kbytes, and the page pool size of the CLL algorithm is 1 Mbyte. All times represent wall-clock time when the target program and checkpointer have exclusive use of the system. All times are averages of five or more runs.

Checkpoint Time and Overhead

Figures 4.1 and 4.2 display the checkpoint time and overhead imposed by the four checkpointing algorithms as a function of the size of merge sort's address space.[1] The total checkpoint time displayed in Figure 4.1 measures the elapsed time from the start of the checkpoint to its conclusion. The second graph in Figure 4.1 extends the first graph to include the checkpoint time when the address space approaches the size of physical memory. The checkpoint overhead in the first graph of Figure 4.2 is the amount of time by which the checkpoint increases the target's running time. The second graph displays the overhead as a percentage of the checkpoint's running time. This is equal to checkpoint overhead divided by the checkpoint time. As stated in Chapter 1,

[1]The raw data for all the graphs in this chapter are in Appendices B and C.

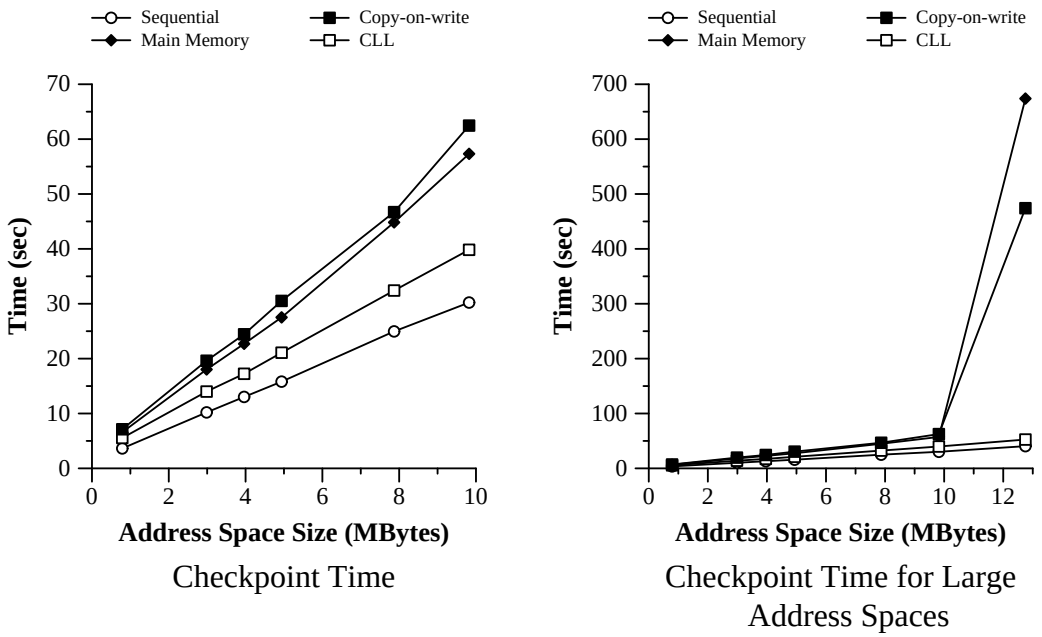

Figure 4.1: Checkpoint Time for the Multiprocessor Algorithms

concurrency can be calculated as:

$$concurrency = 100\% - overhead\ percentage.$$

It comes as no surprise that in all four algorithms, checkpoint time is proportional to the size of the address space. Also as expected, the sequential algorithm records the fastest checkpoint time, followed by the CLL algorithm. The other two algorithms take longer to record a checkpoint because they wait for a complete copy of the checkpoint to exist in main memory before writing the checkpoint to disk.

As shown in the second graph of Figure 4.1, when the address space approaches the size of physical memory, the main memory and copy-on-write algorithms exhibit severe thrashing, as their memory needs far exceed the size of physical memory. The other two algorithms keep their memory needs below the size of physical memory, and therefore do not suffer such rash penalties. It is worth noting that for all but the smallest address space tested, the pool

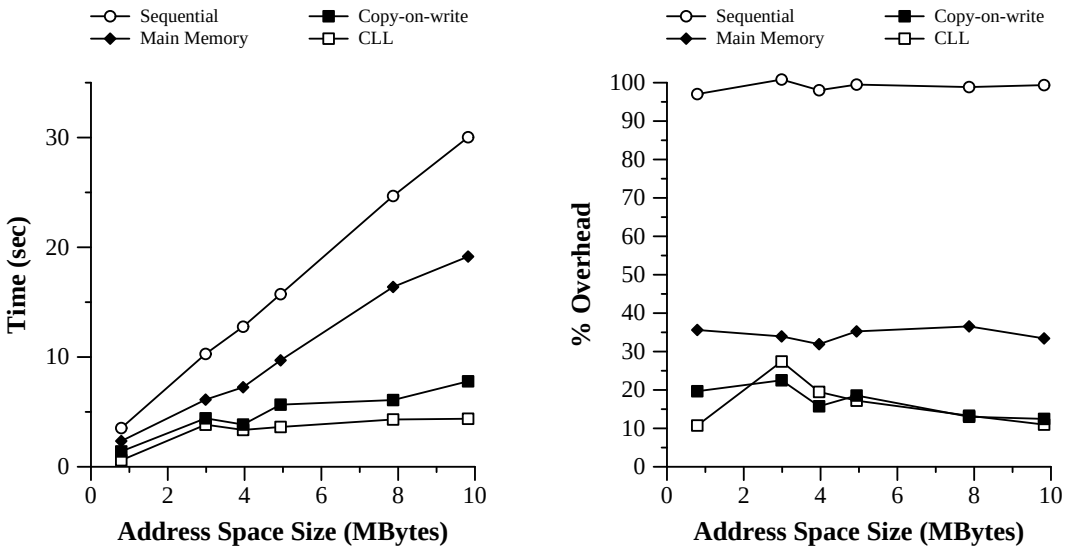

Figure 4.2: Overhead Data for the Multiprocessor Algorithms.

of pages in the CLL algorithm became completely filled. Therefore, some worst-case data is included in the graphs.

Figure 4.2 shows that the two algorithms based on copy-on-write display the smallest overhead, and therefore the greatest concurrency. This is because these algorithms freeze the processors for the smallest amount of time. Taken as a whole, Figures 4.1 and 4.2 show that the CLL algorithm is clearly the best of the four in regard to the combination of checkpoint time and concurrency. The results that follow pertain only to this algorithm.

Latency Data

Figure 4.3 shows latency data for the CLL algorithm. The overhead of the algorithm is divided into two parts—the time that all the threads are stopped initially to protect the address space and save the threads' states, and the time that the target threads are trapped, waiting to process page faults. The first curve of the first graph in Figure 4.3 represents the initial stop time as a function of address space size, and the second curve represents the maximum time that any thread waits as a result of a page fault.

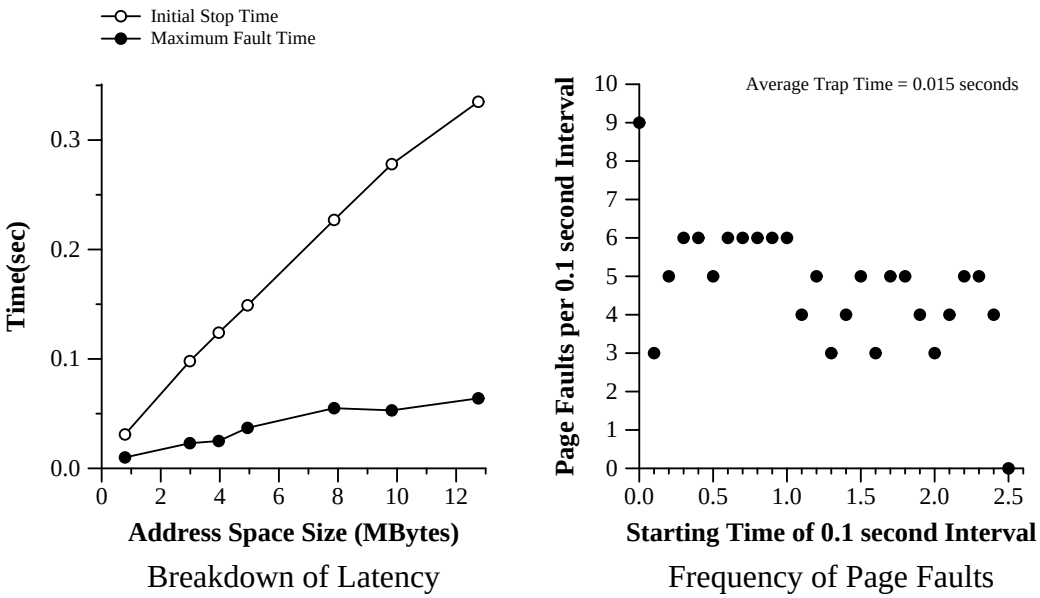

Figure 4.3: Latency Data for the CLL Algorithm.

For address spaces up to three Mbytes, the initial stop time is kept below a tenth of a second. Moreover, for all address space sizes, the maximum page fault time is well below our low-latency goal of 0.1 seconds. The second graph in Figure 4.3 displays the frequency of page faults over time for a run with a four Mbyte address space. In this graph, the checkpoint is broken into 0.1-second intervals, and the number of page faults in each interval is plotted. The purpose of the graph is to show that work is indeed being accomplished by the target threads during the initial phases of the checkpoint.

Note that after an initial burst of nine page faults, the trapping frequency steadies at six faults per tenth of a second for the first second. Then it slows to about four traps per tenth of a second until there are no more page faults. The average time to process a page fault is 0.015 seconds. Thus, during the first second of the checkpoint, the threads spend around 0.09 seconds per 0.1-second interval processing page faults. Since there are three target threads, this means that the threads spend only a third of their time processing page faults in the first tenth of a second; the rest of the time is devoted to completing the merge.