Chapter 3

Optimizations

This chapter outlines several optimizations that can be made to the basic algorithms presented in Chapter 2. These optimizations are meant to improve the algorithms' performance on the three speed metrics described in the introduction: checkpoint time, overhead, and latency.

While the optimizations themselves are quite different, they have similar approaches with respect to improving each metric:

- **Checkpoint time:** The best way to reduce checkpoint time is to decrease the amount of data written to disk. As mentioned in the introduction, incremental checkpointing has the potential to improve the checkpoint time in this way, as does compression, mentioned in Section 3.5 below. On the other hand, checkpoint time can be adversely affected by the checkpointer performing excess work for other optimizations. As we shall see in Chapter 4, there can be situations where the main memory and copy-on-write algorithms described below fall into this trap.

- **Checkpoint overhead:** To reduce the overhead of checkpointing, one must increase the concurrency. Thus, optimizations for this metric attempt to overlap as much checkpointing as possible with the execution of the target program. Main memory checkpointing, outlined in Section 3.1 below, is an example of this technique. Another source of overhead is

serialization of the processors in the checkpointing algorithm. Finding bottlenecks and eliminating them is a second technique of reducing overhead.

- **Latency:** The main way to reduce latency is to break up the checkpoint into small pieces, and then to perform work on a piece at a time only when it is absolutely necessary. We employ copy-on-write to perform this breaking up at the page level. Sections 3.2 and 3.3 detail how copy-on-write is incorporated into checkpointing.

The remaining sections of this chapter outline individual optimizations for checkpointing. Each optimization has been implemented either for multiprocessors or multicomputers. The results are given in Chapter 4.

3.1 Main Memory Checkpointing

Main memory checkpointing is a simple way of increasing the concurrency of checkpointing by overlapping the execution of the target program with the writing of the checkpoint to disk. We first explain it in terms of multiprocessor checkpointing. In the sequential multiprocessor checkpointing algorithm outlined in Section 2.1, the processors are all frozen while the checkpoint is written to disk. This is wasteful as writing the checkpoint to disk is time consuming, and on most multiprocessors it is a sequential activity (for example, the Firefly has one I/O processor through which all disk traffic must pass). Thus, a solution is to copy the checkpoint to another location in main memory while the processors are frozen, and then write the copy to disk once the processors are unfrozen. The algorithm is illustrated in Figure 3.1, and outlined below:

- Freeze all the processors.
- Allocate a second address space in main memory.
- Copy the state of the processors to this second address space.

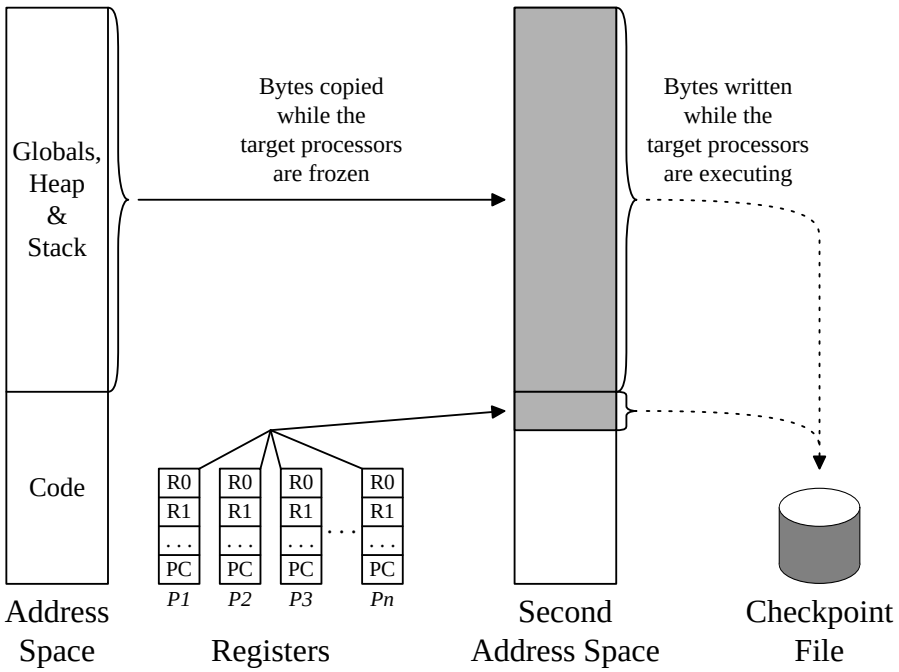

Figure 3.1: Main Memory Checkpointing for Multiprocessors.

- Copy the state of the memory to the second address space.
- Unfreeze the processors.
- Start up a new program thread, which writes the contents of the second address space to disk.

The checkpoint produced by this algorithm is the same as in the sequential algorithm described in Section 2.1. However, the values of the speed metrics should be different. The checkpoint time should increase: It should be equal to the sequential checkpoint time plus the time to make the memory-to-memory copy of the address space. The checkpoint overhead, however, should decrease from being equal to the checkpoint time to being equal to the memory-to-memory copy time. Finally, the latency should decrease to equal the checkpoint overhead.

This optimization also applies to multicomputer checkpointing. It is performed by each processor when it takes a local checkpoint: Instead of writing

the checkpoint directly to disk, a copy is made in main memory, and then after the target is restarted, the copy is written to disk. This should reduce the overhead of checkpointing significantly for all three algorithms described in Chapter 2 for the same reason as in the multiprocessor case: The processors are not frozen during the time-consuming activity of writing to disk.

One of the underlying assumptions of this optimization is that there actually be enough physical memory to hold the main memory checkpoint. If not, and the machine has a backing store, then the optimization will force pages of memory to be swapped to disk. This will degrade both the checkpoint time and the overhead. If the machine does not support paged virtual memory, then the optimization must be modified so that the part of the main memory checkpoint that does not fit in physical memory gets written directly to disk while the processors are frozen. This should not have such an adverse effect on the checkpoint time, but it will not reduce the checkpoint overhead as much as if the entire checkpoint does fit into main memory. Section 3.3 discusses this issue further.

3.2 Copy-on-write Checkpointing

The main memory optimization should decrease both the overhead and latency of checkpointing. However, it still might take on the order of seconds to make an in-memory copy of the address space. This next optimization attempts to improve the latency of the main memory optimization by using *copy-on-write* [TLC85, FR86] to make the main memory checkpoint. Copy-on-write is a technique that uses a processor's virtual memory hardware to make a memory-to-memory copy with low latency. For the multiprocessor case, a copy-on-write checkpointing algorithm works as follows:

First, it freezes the processors and copies their registers to the second address space. Also at this time, it sets the page protection bits of all pages in main memory to be "read-only." Next, it unfreezes the processors, and starts a separate thread of execution, which we call the *"copier thread"*, that

copies pages to the new address space and resets the pages' protection to "read-write." If a thread of the target program generates a page access violation, then it must copy that page to the new address space before setting its protection to "read-write" and restarting. When the copier thread is done copying, the main memory checkpoint is complete, and the copier thread writes the checkpoint to disk. This algorithm is illustrated in Figure 3.2.

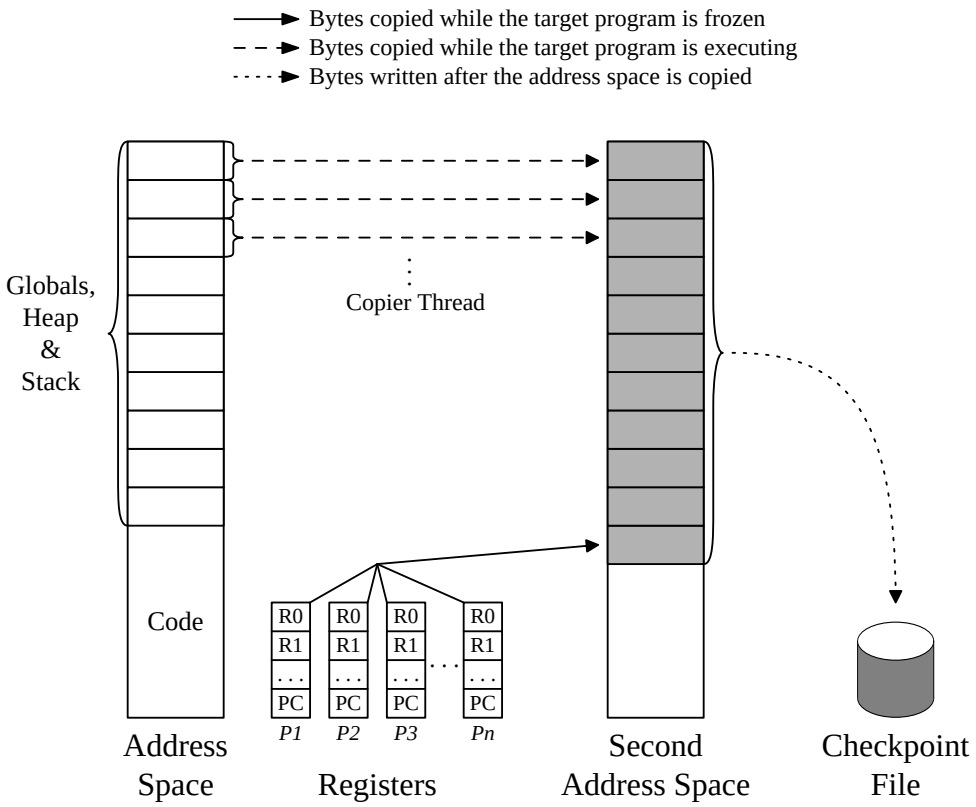

Figure 3.2: Copy-on-write Checkpointing for Multiprocessors.

The multicomputer version of this algorithm is similar: Each processor performs copy-on-write to make the main memory copy of its local checkpoint.

The copy-on-write algorithms produce the same checkpoint files as the main memory algorithms. Moreover, the checkpoint time should be further increased, as there is an extra action in protecting the address space, and the

memory-to-memory copy is done in chunks, rather than all at once. In the multiprocessor case, the checkpoint overhead should decrease once more, because some of the target program's processors are running while the copier thread copies pages, as opposed to the main memory algorithm in which all processors are halted while the address space is copied. The latency of these algorithms is equal to the maximum of the time it takes to copy the registers and protect the address space, and the largest time to process a page fault. Both of these should be significantly smaller than the main memory optimization's latency, which is the time to copy the entire address space.

One variable of this optimization that can have a major effect on the latency is the page size of virtual memory. Although this value is fixed by a machine, one can emulate larger page sizes by increasing the number of bytes that are copied to the second address space by the copier thread and by the page fault handler. Larger pages should increase the time to handle a fault, thereby increasing latency, but they will also decrease the number of faults, and because of locality of reference, they may also decrease the rapidity at which faults occur.

3.3 The CLL Optimization

The copy-on-write optimization can be improved further by adding buffering, a standard operating systems technique usually used to hide disk latency during file system writes [SP88]. What the buffering does in this case is allow the checkpoint to be written to disk at the same time as it is being copied from memory. We call this the *CLL* optimization, which stands for "Concurrent, Low-Latency." For multiprocessors, the CLL optimization works as follows: It first allocates a fixed-size pool of pages (the buffer) in the second address space that the copier thread and page fault handlers fill, and that a new thread called the *"writer thread"* empties by writing the pages to disk. The writer and copier threads are both started immediately after the processors are unfrozen. The algorithm is illustrated in Figure 3.3.

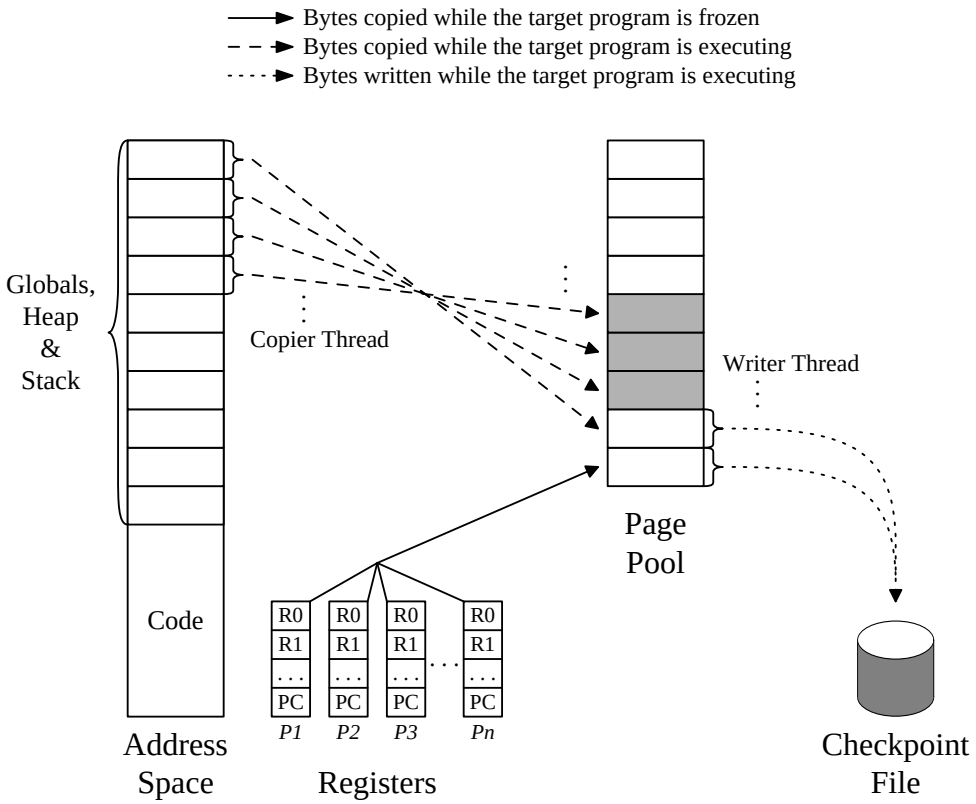

Figure 3.3: The CLL Optimization for Multiprocessors.

As in the copy-on-write optimization, one can apply the CLL optimization to multicomputers by having each processor allocate, fill, and empty its own buffer for its own local checkpoint.

The improvements of the CLL optimization over the copy-on-write optimization should be twofold. First, the extra memory requirements of this scheme are fixed—they are the size of the page pool. The copy-on-write scheme needs extra space that is the same size as the target program's address space, which is more likely to overflow physical memory and cause thrashing. Second, the overall checkpoint time of a CLL checkpointer should be less than that of a copy-on-write checkpointer. This is because the CLL optimization starts

writing to disk as soon as the processors are restarted, instead of waiting until a complete main memory checkpoint has been made.

A possible concern of the CLL optimization is what happens when the page pool fills up. Then pages in the pool are freed only as fast as they can be written to disk. If the size of the page pool is chosen to be the amount of available physical memory, then the CLL optimization should outperform the copy-on-write optimization for the following reason: In the copy-on-write case, pages might be swapped to the swap area on disk so that the checkpoint can fit into main memory. If these pages are part of the checkpoint, then they must be swapped back into memory to be written to the checkpoint file. If they are part of the target program, then they will eventually have to be swapped back into memory when the program needs them. In either case, the copy-on-write scheme performs an extra disk write and read for each swapped page, while the CLL scheme needs no swapping, and therefore performs no extra disk writes. This extreme case for both optimizations is tested in our implementation.

3.4 Staggering Writes

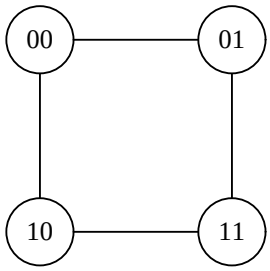

Figure 3.4: Simple multicomputer with Hypercube Routing.

One can change the behavior of the Chandy-Lamport and Network Sweeping algorithms significantly by altering the time at which the m_{CL} messages are sent. For example, in the Chandy-Lamport algorithm, processors can broadcast their m_{CL} messages immediately after taking their local checkpoint instead of immediately before. Similarly, in the Network Sweeping algorithm, the co-

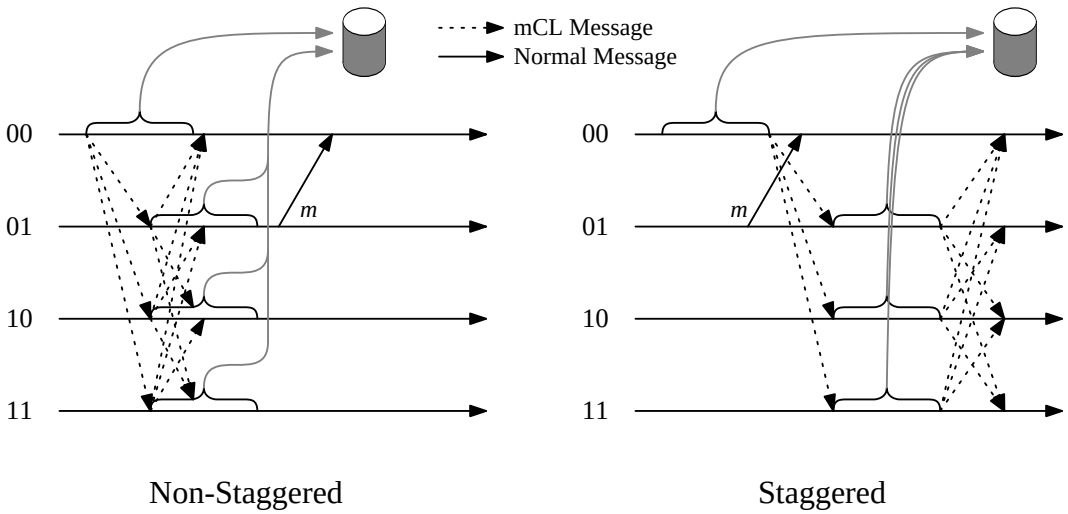

Figure 3.5: Two versions of Chandy-Lamport Checkpointing.

ordinating processor p_c need not broadcast m_{CL} messages immediately before taking its local checkpoint. Instead, the other processors can be notified to start checkpointing by the m_{NS} messages and the other m_{CL} messages.

The result of these changes is that checkpointing is spread out over a larger period of time. In the original versions of the algorithms, p_c broadcasts to all processors to start checkpointing as soon as it starts checkpointing. The result is that all the processors start checkpointing at roughly the same time. With these changes, the processors start checkpointing at different times: The checkpoints are more staggered over time. Thus, we call these algorithms *staggered*, and the original versions *non-staggered*.

Figure 3.5 shows an example of non-staggered and staggered versions of the Chandy-Lamport algorithm on the example multicomputer of Figure 3.4 (which we assume to use the e-cube routing function described in Section 2.6.2). Similarly, Figure 3.6 shows an example of the staggered and non-staggered versions of the Network Sweeping algorithm on the same multicomputer (m_S messages are omitted in both figures for clarity). In the Chandy-Lamport example, there is one regular message m in the system, which gets logged in the staggered version of the algorithm, but not in the non-staggered version. In

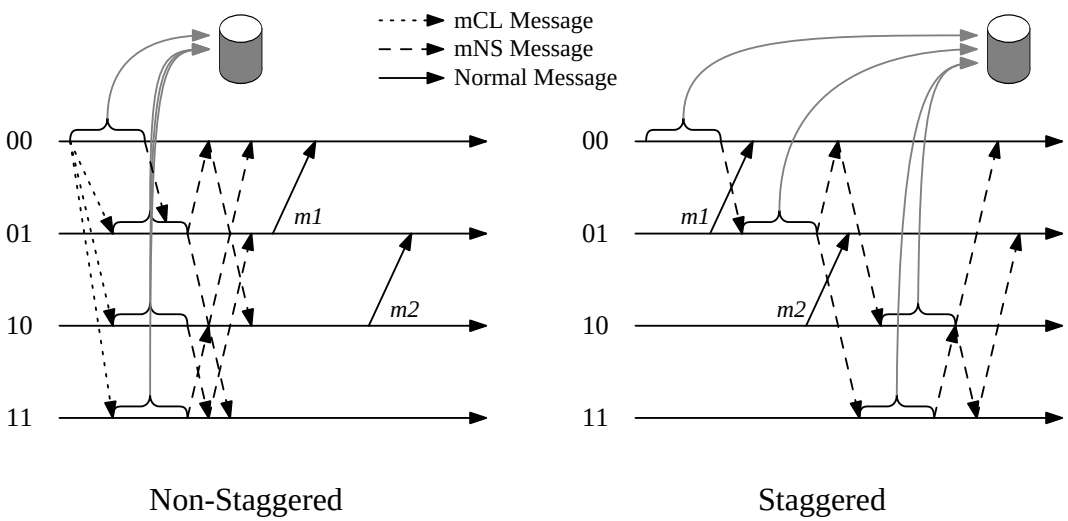

Figure 3.6: Two versions of Network Sweep Checkpointing.

the Network Sweeping example, there are two regular messages $m1$ and $m2$, both of which get logged in the staggered version of the algorithm, but not in the non-staggered version.

These figures demonstrate one of the differences between the staggered and non-staggered algorithms: Since the staggered ones spread the local checkpoints out over a period of time, they are more likely to cause cross-cut messages, which must be logged. If these messages are large, then the checkpointer must perform extra work writing the contents of the messages to disk, which means extra time and overhead for the checkpointer. Thus, in cases where the target program sends large messages, we anticipate that non-staggered checkpointing will perform better than staggered checkpointing.

Another time when non-staggered checkpointing might perform better than staggered checkpointing is when the target program does a lot of synchronizing. Consider the example in Figure 3.7. Here there are two processors that communicate in pairs: Whenever one sends a message to the other, the other acknowledges it with a second message. The first processor stalls until it receives the acknowledgement. In normal operation, neither processor spends much time waiting for acknowledgements. However, when the system is check-