Effects of Dynamic Delays of Staggered Checkpoints

Introduction:

When executing in large-scale computing environments, applications need to frequently save their state to reduce lost computation due to hardware failures, software issues, or policy decisions. Since these applications are typically comprised of independent processes that share data frequently, they must carefully save a consistent system state. A *consistent state* is one that that could have existed in actual execution, implying that no result of an event is saved unless that event's execution is also saved. This saved state is called a *checkpoint.* Currently each process pauses and saves its state at a predetermined location, and then it waits until every other process has saved its checkpoint before continuing. This technique causes two issues that we hope to address: first, when the processes are paused there is no work being done and second, since many processes will likely reach the save point at approximately the same time, the potential for significant contention on the network and file system exists. This network and file system contention can occur when the amount of data to save during a given time frame is greater than the network or file system's throughput handling capacity, and the likelihood of contention increases as the number of processes grows and the amount of memory used by each increases, both of which are current trends. This contention can lead to performance degradation and even system crashes.

A set of checkpoints from each process such that the set creates a consistent state is called a *valid recovery line.* In practice, each application may have many valid recovery lines, a number of which do not save the state of every process at the same point. This staggering of checkpoint locations reduces the likelihood that each process will save its state at the same time as many other processes, which reduces the quantity of data to be transferred to the file system at once, thus reducing individual process wait time and increasing overall performance. This idea is exploited by an alternative approach to saving process states that requires predetermining where processes share data [1], these actions are called *dependence-generating events,* and then staggering process checkpoint locations across those events such that the set of checkpoints create a valid recovery line. We plan to enhance this technique by allowing each process dynamic control of its checkpoint location, so the process may delay saving its checkpoint until system conditions are more favorable. We will estimate these system conditions by using real-time information about the network. Since each process will save its state before executing the next dependence-generating event, the recovery line remains valid.

Related Work:

All portions of this work are dependent on the staggered checkpoint algorithm and testing environment written by Dr. Norman for her PhD thesis [1] and related papers by Drs. Norman and Lin [2]. In their algorithm the compiler checks for dependence-generating events during compilation time and inserts checkpoint locations at those events, staggering process checkpoints at these locations. The goal of our work is to augment that algorithm to allow staggering between dependence-generating events on a per process basis using dynamic information.

Methodology:

To allow each process to independently delay its checkpoint while still ensuring a consistent state, we must make several changes to the existing code base. In this section we discuss the necessary changes made.

First, the system must define the window where saving each checkpoint is still guaranteed to produce a valid recovery line. The system currently places the checkpoint call immediately following the chosen dependence-generating event, meaning that the allowable window to commit each checkpoint is between the chosen dependence-generating event and the one following it. The window is thus defined by the number of instructions until the next dependence-generating event, and we obtain this knowledge by counting the total number of instructions between the initial dependence-generating event and the following event. We use the number of instructions between the events as a rough indicator of temporal separation and utilize that knowledge to create an approximately even temporal distribution of the dynamic checks for the potential checkpoints. While not a perfect temporal displacement, this method provides a good analogue presuming that each instruction takes approximately the same time to complete.

Second, we modify the checkpoint insertion algorithm to insert potential checkpoint locations evenly between the dependence-generating events at specific intervals based on instruction count. A number of interval sizes are tested to determine the best trade off between reduced network contention and the additional overhead for probing the network.

Third, at each potential checkpoint location, the process needs to dynamically determine if this time is advantageous for completing a checkpoint for this dependence-generating event, if it has not already been completed. We modify the potential checkpoint locations added in the last stage so that they are guarded by a check examining the network, and then we use that information, along with the size of the potential checkpoint, to weigh the potential cost of saving the checkpoint at that moment versus the approximate remaining time before the next dependence-generating event. To minimize potential network delay, we must place a large number of potential checkpoint locations. However, each check causes some overhead and so we must be careful to not oversaturate with potential checkpoint locations. Due to these necessary considerations, for our final changes we use our results to inform both modifications to the number and spacing of potential checkpoint locations and the heuristic used to decide whether each one is likely to be optimal.

Results:

We have only been able to complete a small number of simulations at this point. When working with the BT benchmark with small numbers of processes (4-16), both the original implementation baseline and new stagger method achieved similar results, but the new method's overall execution time consistently measures just under half a percent slower than that of the original method. This difference translates to a moderately significant increase in overhead for the checkpoint function. For larger process numbers, the difference is slightly higher but in the same realm: the new method increases execution time by approximately one percent. Please note that these results are preliminary, as they represent a very small sample size. Moreover the potential checkpoint location placement algorithm has not been trained to this data.

Conclusions:

In all spacing modifications we tested and heuristics we tried, we have yet to find any that improve performance in a statistically significant way over the original staggered model. In any experiment with a small number of processes, the network congestion appears too low for our model to make an impact, so our algorithm most often takes the first potential checkpoint of each segment. As

the number of processes increases, the amount of data being saved increases, such that the network could no longer handle the total throughput of the application without diminished performance, and any number of added delays had no appreciable effect. The most likely explanation for this result is that the original staggered model is sufficiently spaced such that throughput is already maximized during all sections utilizing checkpoints. As the network traffic grows, individual processes must wait longer for each checkpoint to be saved. These additional wait times, though small, may induce a small amount of additional stagger on the processes of each group while waiting, which propagate to future checkpoints. If any gains were made by avoiding the first set of small delays, they would seem to be lost by the additional checkpoint overhead later in execution.

Additionally we found very little difference in execution times between using a moderate number of potential checkpoint locations and a larger number, suggesting that the overhead is negligible when compared with the total execution time. Some of the areas of this solution are still unexplored. There may be areas where this method is useful, but we have not yet found any.

[1] A. N. Norman. *Compiler-Assisted Staggered Checkpointing.* PhD thesis, The University of Texas at Austin, 2010.
[2] A. N. Norman and C. Lin. *A Scalable Algorithm for Compiler-Assisted Staggered Checkpointing*, The University of Texas at Austin.