Compiler-Generated Staggered Checkpointing

Alison N. Norman
The University of Texas at Austin
ans@cs.utexas.edu

Sung-Eun Choi
Los Alamos National Laboratory[*]
sungeun@lanl.gov

Calvin Lin
The University of Texas at Austin
lin@cs.utexas.edu

ABSTRACT
To minimize work lost due to system failures, large parallel applications perform periodic checkpoints. These checkpoints are typically inserted manually by application programmers, resulting in synchronous checkpoints, or checkpoints that occur at the same program point in all processes. While this solution is tenable for current systems, it will become problematic for future supercomputers that have many tens of thousands of nodes, because contention for both the network and file system will grow. This paper shows that *staggered checkpoints*—globally consistent checkpoints in which processes perform checkpoints at different points in the code—can significantly reduce network and file system contention. We describe a compiler-based approach for inserting staggered checkpoints, and we show, using trace-driven simulation, that staggered checkpointing is 23 times faster that synchronous checkpointing.

1. INTRODUCTION
Supercomputing clusters are becoming increasingly popular platforms for scientific research. Because the peak power of these clusters scales easily, their sizes are growing at unprecedented rates. For example, Lawrence Livermore National Laboratory's Thunder, currently the largest cluster computer on the Top 500 list, consists of 1024 4-processor nodes—4096 processors in all—and achieves 20 Teraflop peak performance. Livermore intends to grow this machine to achieve over 40 Teraflops *sustained* performance. Unfortunately, as the number of processors, disks, peripherals, and network elements grows, the *mean time before failure* (MTBF) shrinks. Thus, fault detection and recovery are first-class concerns for large supercomputing clusters.

The problem of machine failures can be addressed by periodically checkpointing the application; upon failure, the program is restarted using data from the last successful checkpoint. Today, these checkpoints are manually placed by the application programmer who identifies convenient spots in the code where the checkpoint state (*i.e.*, the volume of data to be saved) is relatively small. For simplicity, all checkpoints occur at the same program location in all processes. These synchronous checkpoints temporarily halt progress of the program but are considered acceptable if their overhead is small, typically 1% of total execution time.

Unfortunately, the increasing size of supercomputing clusters makes these synchronous checkpoints infeasible. As the number of processes increases, checkpoint sizes grow, increasing contention for the network and global file system and making synchronous checkpointing less viable. Because network and file system technology is not evolving as quickly as processor and memory technology, the problem will only get worse over time.

Checkpointing to local disk appears to provide a scalable solution. However, local disks are typically the most failure-prone component of a cluster, and if a disk fails, the program cannot be restarted without first replacing the disk. Thus, in this paper we assume that checkpoints use a global file system.

This paper explores the notion of *staggered checkpointing* [11], a checkpointing approach that reduces contention for the network and file system. Staggered checkpointing allows individual processes to perform their checkpoints at different points in the program code, while still producing a *consistent state*, that is, a state that could have existed during the execution of the program [5]. Note that staggered checkpointing does not dictate *what* to checkpoint (*e.g.*, live data or core dump), only *where* to checkpoint. This paper explains how staggered checkpointing can be performed automatically by compiler analysis and *without* message logging at runtime. We describe a prototype compiler that we use as a proof-of-concept. This prototype makes several simplifying assumptions and does not use a scalable algorithm to identify recovery lines. We then use trace-driven simulation to show the performance advantages of staggered checkpointing over synchronous checkpointing—for 64K processes, synchronous checkpointing is 23 times slower than staggered checkpointing. Finally, we use sample applications to show that there is a large number of possible checkpoint locations, which can be combined in many different ways to produce checkpoints on each process but staggered across the system. With so many possibilities, it will be difficult to manually choose among them and guarantee correctness.

[*]Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. LA-UR-04-5101.

2. BACKGROUND

This section defines terms and concepts that will be needed to understand our algorithm.

A *recovery line* is a set of checkpoints, one per process. A *valid recovery line* represents a state that could have existed in the execution of the program [5], so it disallows a checkpoint of a message receive on one process if the corresponding send of the message is not also checkpointed. Recovery line *a* in Figure 1 is invalid because it saves a message as received on process 1, but process 0 does not save the corresponding send. Thus, the state represented by *a* could not exist in an actual execution of the system. Recovery line *b* is valid because it could have existed in a possible execution. If, upon failure, the system rolls back to a valid recovery line, it is guaranteed to resume in a consistent state.

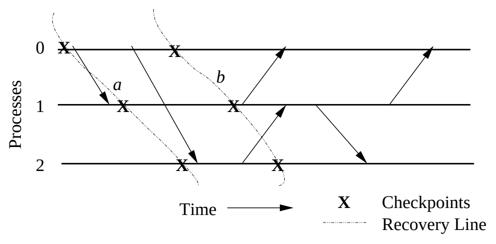


Figure 1: Examples of invalid (*a*) and valid (*b*) recovery lines.

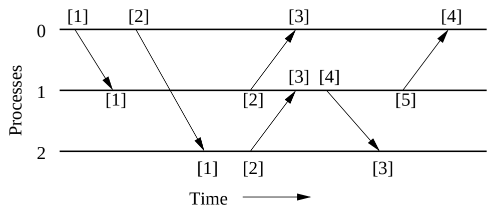


Figure 2: An example of Lamport's logical clocks.

To identify valid recovery lines, we need a method of ordering the messages in time. For this we use *vector clocks*, a method of tracking dependences across processes[1]. Vector clocks were derived from Lamport's logical clocks [9] in which each process maintains its own clock, incrementing it at each event (See Figure 2). Each vector clock is maintained locally by each process as follows:

$VC(e_i)[i] := VC[i] + 1$
if e_i is an internal or send event

$VC(e_i) := max\{VC(e_i), VC(e_j)\}$
$VC(e_i)[i] := VC[i] + 1$
if e_i is receive from process j
where the send was event e_j

where e_i is an event e on process i, and $VC(e_i)$ is its vector clock. $VC(e_i)[i]$ is the element for process i in that vector clock. A process increments its element in its vector clock for each event that occurs. When a process receives a message, it sets its vector clock elements to the maximum of the element in its vector clock and the corresponding element in the sending process' vector clock.

[1]Vector clocks were developed independently by many researchers.

For example, in Figure 3, process 0 begins by sending two messages. For each, it increments its element in its vector clock, so after sending the message to process 2, its vector clock is $[2, 0, 0]$. When process 2 receives the message from process 0, it updates its vector clock to reflect both that an event occurred (the receive) and that it is now dependent on process 0 executing at least two events. So, process 2's clock becomes $[2, 0, 1]$. When process 2 then sends a message to process 1, it increments its vector clock to $[2, 0, 2]$ to reflect another event (the send). Immediately before process 1 receives the message, its vector clock is $[1, 2, 0]$. Upon receiving the message, process 1 increments its element in its vector clock to reflect the receive, and then sets its element for process 2 to 2 to show that it depends on process 2 executing at least two events. Process 1 also sets its element for process 0 to the maximum value of its element for process 0 and that of process 2. Since process 1 had previously only depended on the first event of process 0 and process 2 depends on the first two, then process 1 updates its clock to reflect a dependence on the first two. Therefore, process 1's vector clock becomes $[2, 3, 2]$.

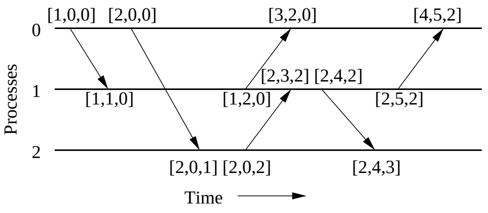


Figure 3: An example of vector clocks.

Given vector clocks, a valid recovery line can be determined using the following formula [10], which states that the recovery line is not valid if process j requires more events on process i than process i's clock reflects.

$$\forall i, j : 1 \leq i \leq n, 1 \leq j \leq n : VC(e_i)[i] \geq VC(e_j)[i]$$

3. OUR SOLUTION

This section describes our compiler-based approach to identifying staggered checkpoints. In this paper, we focus on the mechanisms needed to find valid recovery lines and *not* the policy needed to choose a good one.

Our algorithm has three main phases: the first identifies communicating processes, the second creates vector clocks for each process, and the third identifies all valid recovery lines. We assume that the number of nodes in the system is statically known and that there is only *deterministic communication*. Deterministic communication is communication that depends only on input values or the process's rank, or communication ID.

Communicating Processes

In order to find valid recovery lines, the analysis must first identify communication events and use them to compute vector clocks. To do this, it must first identify all pairs of communicating processes—or *neighbors*—and match sends with the corresponding receives. Some communication calls will have different neighbors based on execution context—or may not be executed at all; the analysis must perform a control-dependence analysis to detect these differences.

```
p = sqrt(no_nodes)
cell_coord[0][0] = node % p
cell_coord[1][0] = node /p
j = cell_coord[0][0] − 1
i = cell_coord[1][0] − 1
from_process = (i −1 + p) % p + p * j
MPI_irecv( x, x, x, from_process, ...)
```

```
from_process = (node / (sqrt(no_nodes)) −1 −1 +
                    sqrt(no_nodes)) % sqrt(no_nodes) +
                    sqrt(no_nodes) * n % sqrt(no_nodes) −1
```

Figure 4: An code example from the NAS parallel benchmark BT and its corresponding result from symbolic expression analysis.

Our compiler computes this information by performing *symbolic expression analysis* on the communication call arguments representing the neighbor. Symbolic expression analysis is a form of backwards constant propagation where variables are expressed in terms of other variables. We assume that these expressions consist of arithmetic operations on constants, the size of the system, and the rank of the process where the call is occurring—the *communicator* (See Figure 4). Our compiler also performs symbolic expressions analysis to track control dependences.

To match the sends to their respective receives and the non-blocking calls to their respective waits, our compiler instantiates each rank in the system and evaluates the relevant symbolic expressions. To do this, the compiler must know the number of processes in the system. It matches the calls based on location in the program and their tag or request values (whichever is pertinent). Control dependence is also taken into account. Previous research has shown how to match these calls [7, 8], but our current implementation uses a simpler algorithm that handles fewer cases.

Vector Clocks

Once the communication has been identified, the analysis can use vector clocks to find valid recovery lines. To create vector clocks, the analysis computes the neighbor for each process at every communication call. Vector clocks can then be created using the rules described in Section 2.

From the communication information gleaned in the previous phase, our compiler creates a vector clock for each node. For each communication call, our compiler iterates through each process that executes that communication call and updates that process's vector clock. We assume that each non-blocking receive occurs at its associated wait; all other events occur at their calls. To assist in the identification of valid recovery lines, with each event our compiler associates the vector clocks for all processes as they occur at that event. Since our compiler must instantiate each process, this algorithm is not ideal, but it does scale linearly with the number of processes in the system and with the number of communication calls in the program.

Finding Valid Recovery Lines

Recall that a recovery line is a set of checkpoint locations, one per process, and a *valid* recovery line is a set of checkpoint locations that represents a state that could have existed in the execution of the program. Two checkpoint locations on different processes are part of a valid recovery line if their vector clocks are *congruous*, where congruous means that process j does not require more events on process i than process i's clock reflects. A valid recovery line is one in which all checkpoint locations are congruous with respect to each other.

The number of valid recovery lines in a program can grow exponentially as a function of the number of processes as well as the number of possible checkpoint locations in a program. Thus, we limit our search to only *dependence-generating* locations; a dependence-generating location is one where a process is waiting for communication (*e.g.*, MPI_Recv or MPI_Wait). These are precisely the events that generate dependences between processes. Thus, for each unique valid recovery line that our analysis discovers, it is possible to adjust the various checkpoint locations relative to the locations that do not generate dependences, but we do not consider such fine-tuning in this paper. It is also worth noting that the number of valid recovery lines in a program is directly related to the communication. For example, a recovery line cannot cross any collective communication, such as a barrier, because all processes are synchronized at the point of the collective communication. Thus, we define *phases* that restrict the scope of recovery lines, further limiting the search space. A phase is the set of communication events between any two synchronization points. The phase includes the beginning synchronization point.

To find a valid recovery line, our compiler first builds a graph that describes the relationship of all possible checkpoint locations to each other. For each of P processes, a node is created for every possible checkpoint location within that process. We will refer to the collection of all of process p's possible checkpoint locations as PCL_p, beginning with PCL_0 through PCL_{P-1}. Hence, a recovery line can be defined by picking one node from each PCL.

Next, our compiler adds edges to the graph. An edge exists between two nodes in the graph if and only if the two nodes can be part of a valid recovery line as defined above. No edges exist between nodes in the same PCL since two checkpoint locations on the same process cannot be part of a valid recovery line. Edges are added in quadratic time, or in our case $O(N \times P \times N \times P)$, where N is the maximum number of possible checkpoint locations for any process. Note that for SPMD programs N will most likely be the same for all processes, and in our benchmark programs described in the next section, we found N to usually be less than 50.

In this graph, a recovery line, *i.e.*, a set of nodes, one from each PCL, is valid if and only if every node has an edge to every other node, *i.e.*, the nodes form a clique. All valid recovery lines are found by finding all such possible cliques. Our algorithm for finding all such cliques is exhaustive. As future work, we plan to implement heuristics for finding *good* recovery lines during the search. First, we must learn more about the characteristics of good recovery line.

Implementation

This algorithm has been implemented using the Broadway [6] source-to-source ANSI C compiler. Broadway performs context-sensitive inter-procedural pointer analysis, which provides a powerful base for our analysis.

4. RESULTS

This section includes our experimental methodology and results. It briefly describes our trace-driven simulator and then describes the results of our synthetic benchmarks. The synthetic benchmarks compare the effects of staggered checkpointing to those of synchronous checkpointing. This sections concludes by analyzing the effects of applying our compiler to application benchmarks.

4.1 Methodology

To simulate thousands of processes, we use a locally-produced trace-driven simulator that models computation events, communication events, and checkpointing events for each individual process. The simulator optimistically models network contention by allowing all processes requiring the network to share it evenly. Even when the network is saturated, our simulator allows the network to deliver its maximum bandwidth. Thus, our results will underestimate the deleterious effects of contention when the network is beyond saturation. The simulator models the file system similarly.

To drive our simulator, we use our compiler to generate trace files from benchmarks. The trace generator employs static analysis and profiling to gather accurate control flow information for each process in the modeled system. Using this information, we create a trace file containing events for each process.

All results assume the following characteristics, which model an existing cluster at Los Alamos National Laboratory: a 1GB/s network that can accept 70MB/s of data from each process; all checkpoint data is written to a global file system that can write 7MB/s.

4.2 Contention Effects

To demonstrate the effects of contention, we use two synthetic benchmarks that consist of a large number of sequential instructions and two checkpoint locations per process. In the first benchmark, dubbed `Synchronous`, every process checkpoints synchronously, once exactly half way through the computation and once at the end of computation. Both sets of checkpoints are followed immediately by a barrier, as is done in the manually-placed checkpoints used today. In the second benchmark, `Staggered`, the processes checkpoint in groups of four at intervals spread evenly throughout the sequential instructions.

Staggered checkpoints improve performance—especially as the cluster size, data checkpointed, and instructions executed increase. Performance improves because, as the cluster size and checkpoint size grow, there is more contention present during synchronous checkpointing and thus more room for improvement when checkpoints are staggered. As the number of instructions executed increases, there is more work with which to stagger the checkpoints, further reducing contention and improving performance. Figure 5 shows run times for the `Synchronous` and `Staggered` benchmarks. In this figure, the number of instructions executed and the amount of data checkpointed increase proportionally with the number of processes. Also, notice that while the graphs show the same trends, the y-axes actually vary by an order of magnitude.

Whereas Figure 5 shows results for a fixed per process problem size, we are also interested in considering the effects of staggered checkpointing when the problem size is fixed even as cluster size grows. Figure 6 shows the improvement in the average time spent checkpointing per process of `Staggered` over `Synchronous`. In this figure, the number of instructions the system executes and the amount of data it checkpoints remain constant as system size increases; therefore, as the number of processes in the system increases, the number of instructions executed and the amount of data checkpointed per process decreases. Please note the different y-axes. This figure demonstrates that staggered checkpointing becomes more helpful as the number of processes and the amount of data being checkpointed increases, in other words, as contention for the network and file system increases. In a system of 64K processes, processes using staggered checkpointing checkpoint 23 times faster than those using synchronous checkpointing.

Table 1 displays our results for a large number of cluster sizes and checkpoint sizes. The amount of data checkpointed and instructions executed is fixed across a row, meaning that the amount of data checkpointed and instructions executed per process is decreasing as the cluster size increases. We increase the amount of data checkpointed by the system until we have achieved 4 GB/process for the 64K process system—the desired amount of memory for future clusters. This table shows a large range of situations for which staggered checkpointing is beneficial. This table also generalizes the results of Figure 6, showing that staggered checkpointing becomes more helpful as the number of processes and the amount of data being checkpointed increases.

Since processes in `Staggered` spend significantly less time checkpointing than those in `Synchronous` for clusters, a 1024-process cluster checkpointing 64 TB of data using staggered checkpointing can take 1,007 checkpoints during the run for a total checkpoint overhead of 1%, whereas if it were using synchronous checkpointing, only 43 checkpoints could be taken for the 1% checkpointing overhead. Figures 7 (a) and (b) compare the number of checkpoints a process may take for 1% checkpointing overhead for varying cluster sizes and varying checkpoint sizes. For the 1% checkpointing overhead that application programmers are willing to tolerate, staggered checkpointing allows a greater checkpoint frequency than synchronous checkpointing, which is important as MTBF shrinks.

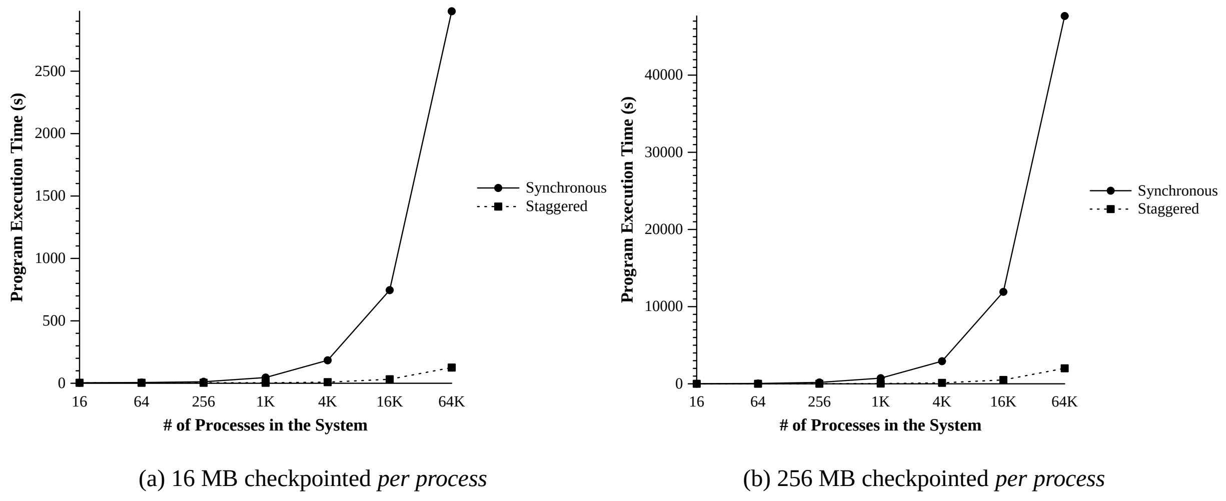
Another option is to use staggered checkpointing to reduce the checkpointing overhead. Figures 7 (c) and (d) show the number of checkpoints that may be taken for .5% checkpointing overhead. Reducing the checkpoint overhead will please the application programmers, and staggered checkpointing still allows many checkpointing options.

4.3 Benchmark Results

As mentioned in the previous section, the number of valid recovery lines is directly impacted by the program's communication. In this section we use three application benchmarks to illustrate the potential for staggered checkpointing.

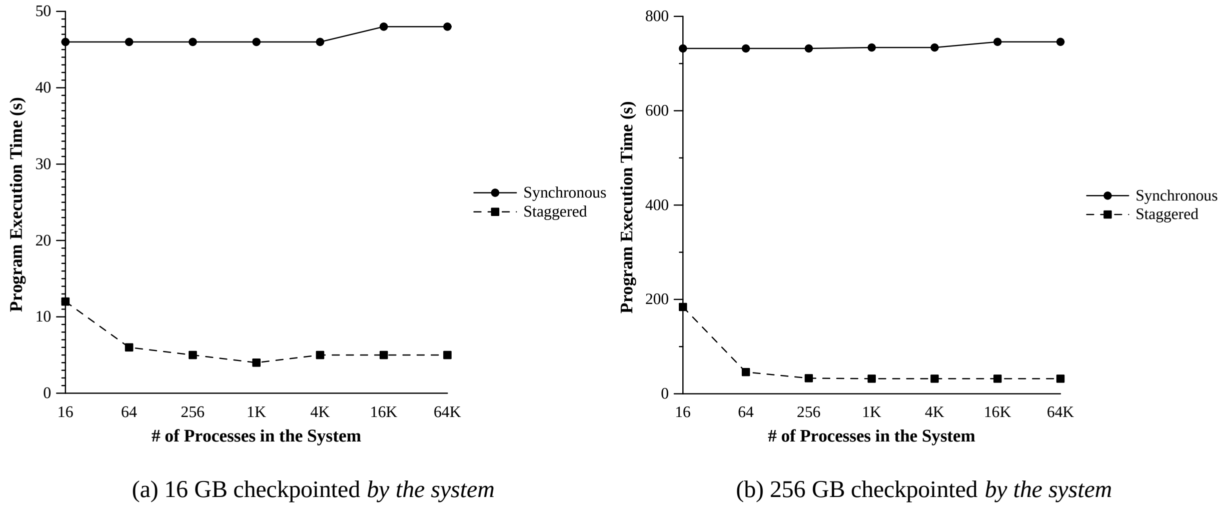
Table 2 describes our three moderately sized benchmark programs. `IS` and `BT` are Fortran codes from the NAS Parallel Benchmark Suite [3] that we converted to C. `BT` is run for 2 iterations. The third program, `ek-simple`, is a well-known CFD benchmark and is the most realistic of our benchmarks. It is also simplified—command-line arguments are assumed to be constants, and a function pointer is replaced with a static function call.

Collective communications are more frequent in `IS` and `BT` and are also scattered through the main computation, thus creating many small- or moderately-sized phases for our recovery line algorithm. `ek-simple` has fewer collective communication calls, but more importantly, the collective communication calls do not result in moderately-sized phases—one of the phases is much larger than the



(a) 16 MB checkpointed *per process* (b) 256 MB checkpointed *per process*

Figure 5: Comparison of Execution Times of `Staggered` and `Synchronous`. Here the amount of work per process remains constant across each data point.



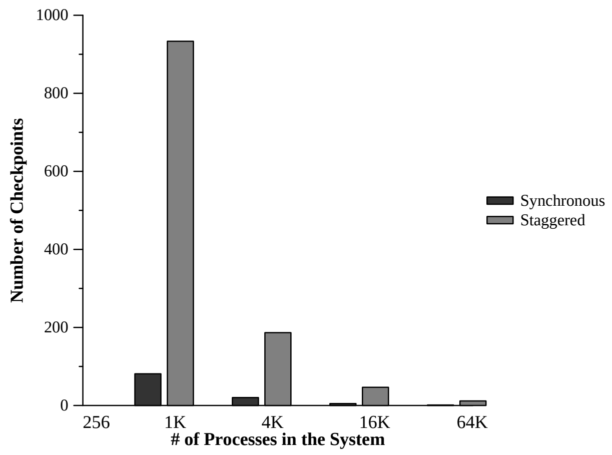
(a) 16 GB checkpointed *by the system* (b) 256 GB checkpointed *by the system*

Figure 6: Comparison of Execution Times of `Staggered` and `Synchronous`. The amount of work remains constant for all system sizes.

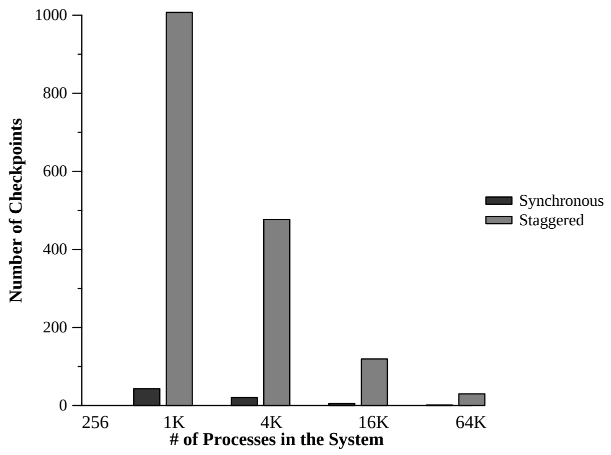
others. This large phase leads to very large numbers of valid recovery lines, as we discuss below.

Table 3 shows the number of statically unique valid recovery lines for each of the benchmark programs for 4, 9, and 16 processes. A statically unique recovery line occurs only once in the application code—it does not account for calling context. `IS`, which has almost all collective communication, has a small, constant number of valid recovery lines for all three cases. `BT`, which has more point-to-point communication and few collective communication calls, sees a doubling in the number of recovery lines with each problem size. `ek-simple`, which we believe is much more representative of real applications, illustrates the real potential for staggered checkpoints. The number of valid recovery lines blows up very quickly, by three orders of magnitude from 4 to 9 processes; the 16 process case has 641,568,404 unique recovery lines (these are not statically unique recovery lines—calling context is considered). Increases in collective communication diminish the benefits of our compiler; checkpointing can still be staggered among local events. The more point-to-point communication a program has, the more opportunity there is for staggering checkpoints between communication events and, thus, the more useful our compiler.

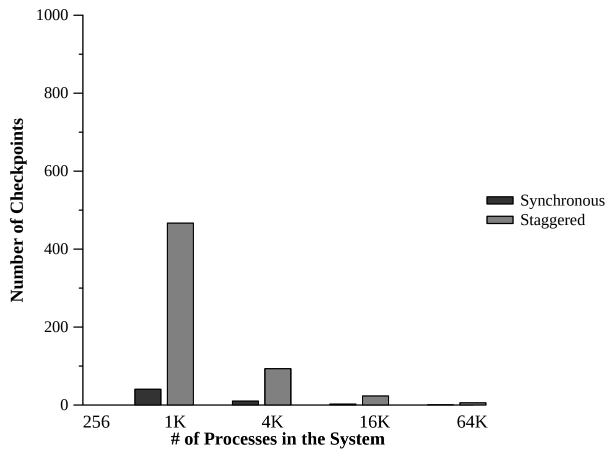
This large number of valid recovery lines reflects the potential that applications have for using staggered checkpointing. The number of valid recovery lines is a function of phase size, in other words, how “staggered” or “spread out” the checkpoint locations can be. As we showed in the previous section, a larger spread reduces the



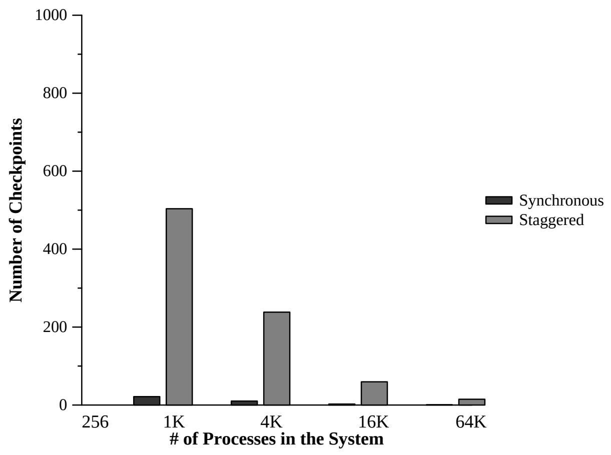
(a) 16 GB checkpointed by the system, 1% allowed checkpoint overhead



(b) 64 TB checkpointed by the system, 1% allowed checkpoint overhead



(c) 16 GB checkpointed by the system, .5% allowed checkpoint overhead



(d) 64 TB checkpointed by the system, .5% allowed checkpoint overhead

Figure 7: Staggered checkpointing has lower overhead than synchronous checkpointing. Figures (a) and (b) show, for the synthetic benchmark, the number of checkpoints that can be performed while allowing the checkpoint overhead to be 1%. Figures (c) and (d) show, for the synthetic benchmark, the number of checkpoints that can be performed while allowing the checkpoint overhead to be .5%.

checkpoint overhead by relieving contention on the network and file system. A good recovery line is a trade-off between this spread and the volume of data checkpointed by each process. This large number represents both an opportunity and a burden. It is important to have flexibility in choosing a good recovery line, but because the number of options grows exponentially, we must find efficient techniques for identifying the good ones. We leave this effort as future work.

In summary, a more globally synchronized application with collective communications throughout will benefit less from staggered checkpointing than one that performs more localized synchronization via point-to-point communication calls. When an application does perform mostly point-to-point communication the potential for staggered checkpointing is enormous. Our future work includes an investigation into identifying characteristics of good recovery lines as well as heuristics for finding them.

5. RELATED WORK

With very few exceptions, the growing body of work on compiler-inserted checkpointing has so far ignored the effects of contention in the network and the global file system. Some approaches use input from the programmer [1] to identify locations that allow small checkpoint sizes. Other solutions use static analysis to restrict checkpoints to communication-free ranges within which checkpointing will lead to a consistent state [4]. Most recently, Bronevetsky et al. proposed an application-level checkpointing protocol [2] that uses message logging with early and late message processing. This pro-

	16	64	256	1024	4096	16384	65536
64 MB	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
256 MB	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
1 GB	33.33%	33.33%	33.33%	33.33%	33.33%	33.33%	33.33%
4 GB	50.00%	66.67%	66.67%	66.67%	66.67%	66.67%	66.67%
16 GB	73.91%	86.96%	88.32%	89.54%	88.79%	89.26%	89.26%
64 GB	75.00%	93.48%	94.63%	94.77%	95.02%	95.13%	95.13%
256 GB	74.86%	93.72%	95.42%	95.57%	95.61%	95.65%	95.65%
1 TB	74.98%	93.71%	95.58%	95.68%	95.68%	95.75%	95.75%
4 TB	75.00%	93.75%	95.60%	95.70%	95.71%	95.78%	95.77%
16 TB	75.00%	93.75%	95.61%	95.71%	95.72%	95.79%	95.78%
64 TB	75.00%	93.75%	95.61%	95.71%	95.72%	95.79%	95.78%

Table 1: Average checkpoint improvement per process: `Staggered` over `Synchronous` The y-axis is the amount of data checkpointed by the system; the x-axis is the number of processes in the system.

Benchmark	Problem Size	Lines of Code	Collective Communication	Point-to-Point Communication
IS	20	1083	14	2
BT	64	4147	8	24
ek-simple	32	3873	6	52

Table 2: Application Benchmark Characteristics. BT and IS are from the NAS Parallel Benchmarks and ek-simple is a well known CFD benchmark.

	4	9	16
IS	5	5	5
BT	30	73	191
ek-simple	98	57,206	$> 2^{24}$

Table 3: Number of Unique Valid Recovery Lines. IS and BT have smaller phases which limit the number of recovery lines. Ek-simple does not perform collective operations in the main loop and thus sees an enormous blow-up in the number of recovery lines as the number of processes increases.

tocol suffers high overhead due to, for example, message logging, even when no failures occur. Their work assumes checkpointing to local disk; which we believe is not a viable solution. Our work assumes the use of a global file system.

Two protocols have considered contention [11, 12]. Both allow processes to stagger their checkpoints. However, neither use a compiler analysis to place checkpoints and guarantee consistency, instead both must use either message logging or some form of additional synchronous checkpoints to guarantee a consistent state.

Many proposed protocols rely entirely on the runtime environment to determine checkpoint placement. Each of these has well-documented shortcomings, but for the purposes of our study, it suffices to realize that none considers contention. These solutions include:

- *coordinated checkpointing*, where each process checkpoints simultaneously with every other process [5],
- *communication-induced checkpointing*, where the communication history is piggybacked on each message, and each process checkpoints independently based on that information [5], and
- *uncoordinated checkpointing*, where each process checkpoints independently, but the end result may be an inconsistent global state [5].

6. CONCLUSIONS

We have shown that, for large supercomputing clusters that have thousands of nodes, contention will become significant. From this we conclude the current technique of manually-inserted synchronous checkpoints will not be effective in the future. We have introduced the notion of compiler-generated staggered checkpointing as an approach to reducing this contention. Our trace-driven simulator has shown that, for 64K processes, staggered checkpointing reduces checkpointing latency by a factor of 23 versus synchronous checkpointing. This reduction in latency will allow the checkpointing frequency to increase with no corresponding increase in overhead—an important conclusion as MTBF shrinks.

We conclude that staggered checkpointing has many benefits. A compiler can identify many valid recovery lines for programs—the next step is to choose good ones and place them in the code. We have outlined a promising solution, but there is still much work to be done. We plan to relax our simplifying assumptions and develop scalable analyses and heuristics to identify good recovery lines.

7. ACKNOWLEDGMENTS

We thank Los Alamos National Laboratory and Sandia National Laboratory for their support in this work. This work was also supported by NSF grant ACI-0313263, the Texas Advanced Research Program, and the DOE Office of Science. We also thank Adam Brown, Jeff Napper, and Lorenzo Alvisi for many enlightening technical conversations, and we thank Samuel Z. Guyer and Teck Bok Tok for infrastructure help.

8. REFERENCES

[1] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, 1994.

[2] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Principles and Practice of Parallel Programming*, June 2003.

[3] NASA Ames Research Center. NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB.

[4] Sung-Eun Choi and Steven J. Deitz. Compiler support for automatic checkpointing. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications*, June 2002.

[5] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[6] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In R.F. Boisvert and P.T. P. Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, 2000.

[7] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. pages 248–259. ACM Press, 1992.

[8] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, Laboratory for Computer Science, University of Bologna, Italy, January 1993.

[11] James S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, June 1993.

[12] Nitin H. Vaidya. On staggered checkpointing. In *Symposium on Parallel and Distributed Processing*, 1996.