Measured Performance of Consistent Checkpointing

Elmootazbellah Nabil Elnozahy
David B. Johnson
Willy Zwaenepoel

Rice COMP TR91-172
December 1991

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Measured Performance of Consistent Checkpointing

Elmootazbellah Nabil Elnozahy
David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

(713) 527-4834

mootaz@rice.edu, dbj@rice.edu, willy@rice.edu

DRAFT

Abstract

Consistent checkpointing protocols can be used to provide transparent fault tolerance for long-running distributed application programs. Many designs for such protocols have been published in the literature, but little has been reported about their performance in practice. In this paper, we give a preliminary report on the performance measurements and implementation of consistent checkpointing on a distributed system running on 16 SUN 3/60 diskless workstations. These measurements include the effect of consistent checkpointing on the running time of six compute-bound distributed application programs.

For five of the applications that we studied, consistent checkpointing performs very well, adding less than 2.5 % overhead to the running time of the program, for checkpointing intervals as small as 30 seconds. For the sixth program, which rapidly modifies large amounts of memory during its execution, the overhead of consistent checkpoint is higher, reaching a maximum of 20 %. The primary factor affecting the performance of consistent checkpointing is the performance of stable storage on which the checkpoints are recorded. The number of messages required by the consistent checkpointing protocol is not a significant factor in the overall performance in our environment.

This work was supported in part by the National Science Foundation Grants CDA-8619893 and CCR-9116343, and by IBM Corporation under Research Agreement No. 20170041, and by an IBM Graduate Fellowship.

however, include the kernel or any of the system servers running on that workstation, since we do not attempt to make these components reliable. An RU can be restarted on any available machine. Currently, each workstation can contain only one RU.

2.1 Checkpointing a Single RU

The checkpoint of a single RU includes a copy of its address space and the state maintained by both the kernel and the system servers for that RU. Instead of writing the full address space during each checkpoint, we use *incremental* checkpointing to reduce the amount of data written on stable storage, thereby lowering the overhead. Incremental checkpointing involves writing only the pages of the address space that have been modified since the last checkpoint. We use the memory management hardware to support incremental checkpointing as follows: At the beginning of every checkpoint, the *dirty* bit of each page in the RU's address space is examined. If the page has been modified, its *dirty* bit is cleared and the page is included in the set of pages to be written on stable storage.

The set of pages that constitute the incremental checkpoint must not be modified by the application processes while the checkpoint is being recorded on stable storage. Otherwise, the resulting checkpoint may not represent the state that the RU had at any single point in time. The solution in which the processes in the RU are blocked until the checkpoint is completely written on stable storage [8, 15] is expensive. Instead, we have considered two alternative methods that allow the processes to continue execution in parallel with checkpointing.

The first method uses *copy-on-write* by relying on the memory protection provided by the memory management hardware [4]. In this method, the pages that constitute the incremental checkpoint are write-protected. A checkpointing server writes these pages on stable storage while the processes continues execution. Each page is unprotected after being written on stable storage. If a process attempts to modify one of these pages before it is written on stable storage, a memory protection fault is generated. In this case, the kernel will remove the protection on the page after copying it into a newly allocated physical memory page, and the process is allowed to continue execution. The newly allocated physical memory page is not accessible to the processes running in the RU, and is used by the checkpointing server to write the original content of the page to be modified on stable storage. The server deallocates the page after it is written on stable storage. This scheme is similar to that used by Li et al. [10] in their concurrent checkpointing technique for small physical memories. Unlike our implementation, however, they did not use the memory management hardware to implement incremental checkpointing.

The second method uses *pre-copying* [7] to implement checkpointing[1]. In *pre-copying*, if the number of pages in the incremental checkpoint is small, they are copied in the local memory and written from there on stable storage. Otherwise, the pages are written in a *pre-copying* pass on stable storage while the application processes continue execution. After the *pre-copying* pass completes, the pages in the incremental checkpoint are re-examined and those that were modified during the *pre-copying* pass are marked for rewriting on stable storage. Again, a test is made to determine whether a local copy of the modified pages can be made or if another *pre-copying* pass is necessary. This iterative procedure continues until either a local copy is made or after a specified number of *pre-copying* passes is exhausted. In the latter case, the application processes are blocked while the modified pages are written on stable storage. This technique works well provided that the number of pages modified during a *pre-copying* pass is small since, in this case, the processes are not blocked except for the short time while the copy to local memory takes place.

We have implemented checkpointing by the two methods to compare between their performances. Our measurements (see Section 4) show that checkpointing using *copy-on-write* performs better than the *pre-copying* technique. Therefore, we have chosen *copy-on-write* as a basis for our consistent checkpointing implementation.

2.2 Consistent Checkpointing

We have based our implementation on a simple three-phase consistent checkpointing protocol that is a variation on the two-phase protocol by Koo and Toueg [9]. In this implementation, each RU maintains one *permanent* checkpoint, belonging to the most recent system-wide consistent checkpoint. A *checkpoint server* executes on each machine and controls the checkpointing of the local RU. One distinguished server acts as a *coordinator* and sends requests to the other servers to synchronize the consistent checkpoint. When the *coordinator* runs the protocol, each RU takes a *tentative* checkpoint which replaces the permanent one only if the protocol terminates successfully [9]. The protocol proceeds as follows:

1. The *coordinator inhibits* the local application processes from sending messages, protects the modified pages in the RU's address space with *copy-on-write*, and begins writing those pages to the tentative checkpoint file. The *coordinator* then sends a broadcast message to the other checkpoint servers, informing them of the start of a new checkpoint. When a checkpoint server receives the message, it inhibits the local RU from sending application messages. It

[1] *Pre-copying* was originally used by Theimer et al. to implement process migration [17]. What we present here is an adaptation of their method to support checkpointing.

also protects the modified pages in the local *RU*'s address space with *copy-on-write*, and begins writing those pages to the checkpoint file. It then acknowledges the receipt of the *coordinator*'s broadcast.

2. After the *coordinator* has received an acknowledgement from each checkpoint server, it allows the local processes to resume sending application messages and sends a broadcast to the other servers, directing them to remove the inhibition on sending application messages. When a server receives this message, it immediately acknowledges its receipt.

3. After each server completes writing the checkpoint of the local *RU*, it sends a message to the *coordinator* informing it of this fact. When the *coordinator* has collected this message from all servers, it replies to each instructing them to replace the previous permanent checkpoints with the tentative ones. The previous permanent checkpoints are discarded.

Inhibiting application processes from sending application messages in the period between the *coordinator*'s first and second broadcasts ensures that the *copy-on-write* protection in all *RU*s is established in a consistent manner. Since no process can send a message after its address space has been protected, the set of checkpoints of the different *RU*s cannot show a message that was sent *after* the checkpoint of the sender has been taken but received *before* the checkpoint of the receiver was taken. The recorded checkpoints must therefore form a consistent checkpoint.

This implementation deviates from the protocol of Koo and Toueg in three aspects:

- We added a third communication round so that application processes are inhibited from sending messages *only* for the period of time required for the *coordinator* to perform two broadcasts to the servers (including the acknowledgement of the first broadcast). In the two-phase protocol of Koo and Toueg, the processes are inhibited from sending messages until *all* checkpoints are recorded on stable storage. We believe that this would be a severe penalty in environments like ours, where recording checkpoints on stable storage is relatively slow.

- We have not implemented the dependency tracking of Koo and Toueg that is used to minimize the number of *RU*s that have to participate in taking a consistent checkpoint. This decision was motivated by the desire to simplify our first attempt at the implementation. We plan to add this feature eventually.

- We have only one fixed *RU* that can start the protocol. This decision was also motivated by our desire for simplicity, since we chose not to deal with the problems caused by concurrent invocations of the protocol.

2.3 Stable Storage

A checkpoint of an *RU* is stored as a file on a shared network file server. The file server allows files that store the tentative and permanent checkpoints of the same *RU* to share data blocks. This facility is used to support storing the incremental changes to the checkpoint file with low overhead. When an *RU* records a tentative checkpoint, it writes to a new file only the pages of its address space that have been modified since the last checkpoint. The remaining data blocks that represent the portions of the address space that have not been modified since the last checkpoint are automatically shared between the tentative and permanent checkpoint files. Each file thus contains a complete continuous image of the *RU*'s address space. When the tentative checkpoint is to be made permanent, the data blocks that are private to the old checkpoint file are discarded.

Our current implementation uses a single network file server to simulate stable storage. This subjects the system to a single point of failure [2]. In the final version of the implementation, we plan to replicate the checkpoint files on several network file servers to address this problem.

3 Application Programs

The set of distributed application programs used in this study was chosen to represent a wide range of compute-bound programs. These programs differ in memory requirements and usage, as well as in communication patterns. They were not modified in any way to make use of consistent checkpointing.

nqueens: This program counts the number of solutions to the *n-queens problem*. The search space is evenly distributed among multiple processes, and no communication is required among the processes except for reporting the final solution.

tsp: This program uses a branch-and-bound algorithm to find the solution to the *traveling salesman problem* for a given map of n cities. A main process maintains the current best solution to the problem and a task queue that contains subsets of the search space. The main process assigns tasks from the queue to a number of slave processes. Each slave process solves its assigned task and reports the solution back to the main process, which updates the current best solution and returns it back to the slave process along with a new task.

[2] When we started the implementation, we simply did not have the resources to replicate the checkpoint files on two different network file servers.

`fft`: This program computes the *Fast Fourier Transform (FFT)* for a given input of n data points. The problem is evenly distributed among multiple processes, and no communication is required except for reporting the final solution.

`sor`: This program carries out *Successive Over-Relaxation (SOR)* in a given $n \times n$ matrix. The problem is distributed among multiple processes by giving each a section of the matrix on which to compute. After each iteration, each process exchanges the new values on the edges of its section with the corresponding neighbor processes.

`gauss`: This program performs *Gaussian elimination with partial pivoting* on a given $n \times n$ matrix. The problem is distributed among multiple processes by giving each a subset of the matrix columns on which to operate. At each step of the reduction, the process which holds the pivot element sends the pivot column to all other processes.

`matmult`: This program multiplies a given pair of matrices of size $n \times n$. The problem is distributed among multiple processes by giving each a section of the result matrix to compute, and no communication is required except for reporting the final solution.

4 Performance of Checkpointing a Single RU

Consistent checkpointing protocol cannot perform well without efficient checkpointing for each RU. In this section, we examine the performance of checkpointing a single RU in isolation to obtain more accurate measurements than would be possible when running as part of a large distributed application. We also compare the performance of checkpointing by *copy-on-write* against that by *pre-copying*.

4.1 Component Costs

The total elapsed time required to write a new checkpoint is dominated by the time required to write the modified pages of the address space to the checkpoint file. In our implementation, this time is approximately 2.78 seconds per megabyte of modified address space, or about 21.7 milliseconds per 8-kilobyte memory page. The time required to write the address space to the checkpoint is approximately 4 % more than that required to write the same amount of data to a file on the network file server. In addition to the cost of writing the modified address space, there is a small fixed cost of about 70 milliseconds per checkpoint. Included in this fixed cost are 17 milliseconds required to open and close the checkpoint file, and 2.1 milliseconds required to extract the relevant state of the kernel and the system servers.

The time required to perform recovery of an application program after a failure is highly dependent on the particular program being recovered. This time varies most with the time needed to re-execute from the checkpointed state, but this re-execution time is bounded by the interval at which new checkpoints are recorded. As with checkpointing, the time to restore a process from its checkpoint is dominated by the time required to read the process address space data into memory. In our implementation, the measured time for restoring a checkpoint is approximately 1.5 seconds per megabyte of user address space being restored, plus a small fixed cost of about 50 milliseconds. For comparison, the time required to read the address space from the checkpoint is approximately the same as that required for a user process to read the same amount of data from a file on the network file server.

4.2 Performance of the Application Programs

To measure the effect of checkpointing on the performance of application programs, we ran each of the six application programs described in Section 3, using only a single process rather than executing as a distributed application. Table 1 shows the memory requirements for each program when run using only a single process. The memory size is the total amount of memory used by the program. The code size and the data size are the amounts of memory used for code and data, respectively. The size of the checkpoint file is equal to the total size of the memory used by the process in addition to about 2 Kbytes of kernel and server states. Table 1 also shows the running time for each process without checkpointing. The running time was measured in hours, minutes and seconds.

We measured the running time of each program with four different checkpointing intervals: 30 seconds, 1 minute, 5 minutes, and 10 minutes. Table 2 shows the percentage increase in run-

Table 1

Properties of application programs measured (sequential version).

Name	Problem Size (n)	Memory Size (Kbytes)	Code Size (Kbytes)	Data Size (Kbytes)	Running Time (hh:mm:ss)
`nqueens`	15	56	30	26	2:38:04
`tsp`	17	112	29	83	3:49:10
`fft`	4096	152	23	129	2:56:46
`sor`	512	2080	18	2062	3:22:42
`gauss`	600	2856	24	2832	3:01:21
`matmult`	850	7536	18	7518	2:21:24

ning time for each application due to checkpointing overhead. For `nqueens`, `tsp`, and `fft`, the running times remained essentially the same with the increase in checkpointing frequency from no checkpointing to 30-second checkpoint [3]. For `gauss` and `sor`, which rapidly modify large amounts of memory, the running time increase was noticeable, reaching 16 % for `sor` with the increase of checkpointing frequency from no checkpointing to 30-second checkpoint. However, for larger checkpoint intervals, the effect of checkpointing overhead was negligible. For `matmult`, the running time remained essentially the same even though this program uses the largest amount of memory among the six applications studied.

4.3 The Behavior of Checkpointing

Table 3 shows the average performance of recording a process checkpoint during the execution of these application programs. For each application program and each checkpointing interval tested, the average elapsed time required to complete each checkpoint is reported, as is the average time that the application program's execution was blocked during each checkpoint. The elapsed time represents the time during which the modified pages of the process's address space were being written to the checkpoint file, but most of this writing occurred in parallel with the process's continued execution due to the *copy-on-write* mechanism used in our implementation. The process is blocked only if no free memory page is available when a *copy-on-write* memory protection fault occurs during the checkpoint. Table 3 also shows the average number of pages written to the file during each checkpoint. This is the number of pages of the process's address space that had been

Table 2

Percentage increase in running time for application programs due to checkpointing overhead.

Program	Checkpointing Interval			
	30 sec.	1 min.	5 min.	10 min.
`nqueens`	0.07	0.04	0.00	0.02
`tsp`	0.02	−0.02	−0.04	−0.04
`fft`	0.13	0.07	0.03	0.02
`sor`	15.63	7.79	1.60	0.77
`gauss`	4.72	3.14	0.69	0.38
`matmult`	0.02	−0.03	0.00	0.09

[3] Some measurements imply that checkpointing actually improves performance. This is only an artifact of the difference in running times being much less than the standard deviation in these measurements, which was 0.5%.

Table 3

Average performance of a recording a process checkpoint (secs.)

Program	Checkpoint Interval	Elapsed Time	Blocked Time	Pages Written	Pages Faulted
`nqueens`	30 sec.	.17	.00	2	2
	1 min.	.15	.00	2	2
	5 min.	.15	.00	2	2
	10 min.	.19	.00	2	2
`tsp`	30 sec.	.17	.00	2	2
	1 min.	.13	.00	2	2
	5 min.	.18	.00	2	2
	10 min.	.15	.00	2	2
`fft`	30 sec.	.29	.00	4	3
	1 min.	.28	.00	4	3
	5 min.	.29	.00	5	3
	10 min.	.32	.00	5	3
`sor`	30 sec.	5.57	4.26	258	131
	1 min.	5.55	4.11	258	131
	5 min.	5.52	4.33	258	131
	10 min.	5.46	3.98	258	131
`gauss`	30 sec.	5.09	.19	229	20
	1 min.	5.58	.38	267	34
	5 min.	5.88	.47	274	34
	10 min.	5.82	.09	274	34
`matmult`	30 sec.	.35	.00	8	3
	1 min.	.44	.00	13	3
	5 min.	1.29	.00	48	3
	10 min.	2.14	.00	92	3

modified since the previous checkpoint. The final column of Table 3 gives the number of pages of the process's address space that were being written to the checkpoint file but were modified again while this write operation was still in progress, causing a *copy-on-write* protection fault.

For the `nqueens`, `tsp`, and `fft` programs, blocking the process's execution during the checkpoint was not necessary. As indicated by the number of pages written and faulted during the checkpoint, these three programs use very little data memory. For example, only one stack page and one global variable data page are used by `nqueens` and `tsp`. Since this is the minimum memory that a normal

running program can use, the checkpointing of these programs is essentially unaffected by the choice of checkpointing interval.

For `sor` and `gauss`, their heavier memory usage placed increased demands on the checkpointing mechanism. The `sor` program modifies its entire data memory during each iteration, and thus must write all of this memory to every checkpoint. This is demonstrated by the number of pages written on each checkpoint, which is equal to the program's data size. In fact, during execution intervals as small as the elapsed checkpointing time (about 5.5 seconds), `sor` was able to modify about half of its data space, as indicated by the number of page faults during each checkpoint. This high fault rate caused all available memory on our hardware to be exhausted with *copy-on-write* copies, and thus `sor` was forced to block for most of its elapsed checkpointing time. In this respect, `sor` represents a worst case scenario for our checkpointing method. The `gauss` program exhibits similar behavior, although it does not modify as much of its address space as quickly, and thus does not cause as many *copy-on-write* faults or as much blocked time as does the `sor` program.

Although `matmult` has the largest data size of the six programs studied, much of this data is actually treated as read-only by the program (the two input matrices), and it computes more slowly across its data space, modifying only one output matrix element at a time. Thus, incremental checkpointing makes `matmult` behave like the smaller programs in terms of its checkpointing performance, even though it has the largest memory requirement.

Our conclusion is that the overhead of checkpointing is mainly determined by the rate at which an application program modifies its address space during each checkpoint interval.

4.4 The Effect of Incremental Checkpointing

Table 4 shows the size of the incremental checkpoint as a percentage of the total amount of memory used by each application program. Incremental checkpointing shows its effectiveness in `matmult`. Compared to the size of a full checkpoint, the size of an incremental checkpoint is smaller by two orders of magnitude at a checkpointing interval of 30 seconds, and smaller by an order of magnitude at a checkpointing interval of 10 minutes. If we did not implement incremental checkpointing, we believe that the running time for `matmult` would have increased by at least the same ratio as `sor`, instead of being essentially unchanged as shown in Table 2. Incremental checkpointing was also effective for `nqueens`, `tsp` and `fft`, reducing the amount of data written on disk by a factor from 3 to 6. Incremental checkpointing, however, did little for `gauss` and almost nothing for `sor`. These applications present a worst case for incremental checkpointing, because the rapid modification of large portions of the address space tends to make an incremental checkpoint more or less a full checkpoint.

Table 4

Percentage of the address space included in an incremental checkpoint

Program	Checkpointing Interval	
	30 sec.	10 min.
`nqueens`	28.6	28.6
`tsp`	14.3	14.3
`fft`	21.0	26.3
`sor`	99.2	99.2
`gauss`	64.1	76.8
`matmult`	0.8	9.8

Our conclusion is that incremental checkpointing is an effective optimization for many applications. Since it does not add overhead and is simple to implement, we believe that it should be an integral part of any checkpointing implementation, even though there are applications that will not benefit.

4.5 Pre-Copying Versus Copy-on-Write

We found that the effect of either technique on the application performance is dominated by the time in which the process is blocked while the checkpoint is taken. Under either technique, the blocking time was a small fraction of the checkpoint elapsed time for all applications but the `sor` program. Therefore, for these applications, the effect of *pre-copying* checkpointing on the application performance is similar to that reported for *copy-on-write*. However, for `sor`, the pre-copy of the address space before the process was blocked accomplished little, due to the rapid modification of the address space. When the process was blocked, almost its entire address space was rewritten to the file. For example, at a 1-minute checkpointing interval, the blocking time for *pre-copying* is about 40 % higher than the blocked time for *copy-on-write* and the elapsed time per checkpoint was just over twice that with *copy-on-write*.

Pre-copying also writes more pages to disk in a checkpoint than *copy-on-write* on the average. Because this tends to increase the load on the file server, and because the measurements do not show any advantage of *pre-copying* over *copy-on-write* we conclude that *copy-on-write* is always at least as good as *pre-copying*, and for large programs, may significantly outperform *pre-copying*.

5 Consistent Checkpointing

In this section, we present the measurements of the application programs when run using 16 identical SUN-3/60 workstations connected by a 10Mbit/sec Ethernet. For brevity, we report here the measurements using only 30-second checkpointing intervals.

5.1 The Performance of Application Programs

Each application program contained sixteen processes that ran each on a different workstation and communicated through the network to compute the solution. Table 5 shows the memory requirements of each of these sixteen processes. It also shows the running time for each of the six application programs when distributed. The total size shown for each application-program is the amount of memory required by *each* of the sixteen processes. The code and data size are the amounts of this memory used to store code and data, respectively. For most programs, the amount of memory used by each process in the distributed case is smaller than that used in the sequential case, since the data is not fully replicated at each process.

Table 6 gives the percentage increase in running time for the six distributed application programs at a checkpointing interval of 30 seconds. As with the sequential checkpointing performance, the overhead of consistent checkpointing for most programs did not cause a significant increase in the running time. Likewise, consistent checkpointing overhead was high for **sor**. However, the overhead is higher with consistent checkpointing than with the sequential case. The reason is that all processes of the application checkpoint at essentially the same time, increasing the load on the file server. We plan to investigate whether using dependency tracking as suggested by Koo and

Table 5

Memory requirement of each process in the application programs measured (distributed version)

Name	Problem Size (n)	Total Size (Kbytes)	Code Size (Kbytes)	Data Size (Kbytes)	Running Time (hh:mm:ss)
`nqueens`	15	48	18	30	0:12:30
`tsp`	17	48	21	27	0:58:24
`fft`	4096	104	21	83	0:11:38
`sor`	512	2088	20	2068	0:35:59
`gauss`	600	248	20	228	0:12:24
`matmult`	850	3224	18	3206	0:06:37

Table 6

Percentage increase in running time for application programs running on 16 machines.

Program	Checkpointing Interval
	30 sec.
`nqueens`	0.6
`tsp`	1.0
`fft`	2.0
`sor`	19.6
`gauss`	2.4
`matmult`	0.7

Toueg [9] can solve this problem by minimizing the number of processes that have to participate in taking the checkpoint.

5.2 The Behavior of Consistent Checkpointing

Table 7 shows the average performance per process for each checkpoint recorded during the execution of these six distributed application programs. This table shows the same performance measures as Table 3 reported for single-process checkpoint. The figures reported here are smaller than in Table 3 because the program data is distributed over 16 machines. Again, the overhead of checkpointing is determined by the rate at which each process modifies its address space during the interval between two consecutive checkpoints.

Table 7

Average performance per process for consistent checkpointing (secs.)

Program	Checkpoint Interval	Elapsed Time	Blocked Time	Pages Written	Pages Faulted
`nqueens`	30	.45	.17	2	1
`tsp`	30	.45	.28	2	1
`fft`	30	.50	.36	4	2
`sor`	30	1.40	.59	24	19
`gauss`	30	1.50	.54	18	16
`matmult`	30	.67	.48	5	2

5.3 The Effect of Incremental Checkpointing

The arguments made in our study of the checkpointing of the sequential programs in favor of using incremental checkpointing apply for the distributed case as well. For example, incremental checkpointing reduces the amount of data written during each checkpoint to only 1 % of the amount of memory used by the process.

5.4 The Effect of the Number of Messages

Our implementation of consistent checkpointing requires three rounds of communication between the *coordinator* and the rest of the processes, irrespective of the distributed application that is being checkpointed. This, in theory, is more expensive than many other consistent checkpointing protocols in the literature. However, the results reported in Tables 6 show that the overhead for `matmult` is 0.7 %. This insignificant increase in running time includes both the effect of writing the checkpoint on stable storage and the effect of the communication. Since the communication rounds are fixed for all applications, the increase in overhead in applications like `sor` cannot be attributed to the number of messages exchanged. We conclude that the overhead of consistent-checkpointing is mainly affected by the performance of stable storage, rather than the number of messages needed to coordinate the checkpoint.

6 Conclusion

In this paper, we have presented a preliminary report about the measured performance and implementation of consistent checkpointing. For application programs that use only small amounts of memory or slowly modify new portions of their memory, consistent checkpointing performs well, adding essentially no overhead to the running time of the program, for checkpointing intervals as small as 30 seconds. For programs that quickly modify all or most of a large address space, the overhead of consistent checkpoint can be as high as 20 % for the applications that we tested at a checkpointing interval of 30 seconds. The primary factor affecting the performance of consistent checkpointing is the performance of stable storage on which the checkpoints are recorded. The effect of the coordination messages required by the consistent checkpointing protocol on the overall performance is insignificant.

We are continuing to collect performance measurements using additional types of programs such as distributed event simulation. In addition, we are also measuring the effects of replicating the checkpoint data on two or more file servers, in order to increase the resiliency of the checkpointing

implementation. We also plan to study the overall performance of application programs in the presence of failures and recoveries.

References

[1] Mustaque Ahamad and Luke Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 1–11, October 1989.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[4] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[5] S. Israel and D. Morris. A non-intrusive checkpointing protocol. In *The Phoenix Conference on Communications and Computers*, pages 413–421, 1989.

[6] F. Jahanian and F. Cristian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, Bologna, Italy, September 1991.

[7] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[8] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent fault-tolerance in parallel orca programs. Technical Report IR-258, Vrije Universiteit, Amesterdam, October 1991.

[9] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[10] K. Li, J.F. Naughton, and J.S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[11] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 1–10, October 1991.

[12] B.P. Miller and J.D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 141–150, June 1988.

[13] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[14] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, May 1986.

[15] Y. Tamir and C.H. Séquin. Error recovery in multicomputers using global checkpoints. In *1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[16] K.E. Taylor. The role of inhibition in consistent cut protocols. In *Proceedings of the 3rd Workshop on Distributed Algorithms*, 1989.

[17] M. Theimer, K. Lantz, and D.R. Cheriton. Preemptable remote execution facilities in the V-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[18] Z. Tong, R.Y. Kain, and W.T. Tsai. A lower overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

[19] K.-L. Wu and W.K. Fuchs. Recoverable distributed shared memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.