The Performance of Consistent Checkpointing

Elmootazbellah Nabil Elnozahy
*David B. Johnson**
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas 77251-1892

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

Abstract

Consistent checkpointing provides transparent fault tolerance for long-running distributed applications. In this paper we describe performance measurements of an implementation of consistent checkpointing. Our measurements show that consistent checkpointing performs remarkably well. We executed eight compute-intensive distributed applications on a network of 16 diskless Sun-3/60 workstations, comparing the performance without checkpointing to the performance with consistent checkpoints taken at 2-minute intervals. For six of the eight applications, the running time increased by less than 1% as a result of the checkpointing. The highest overhead measured for any of the applications was 5.8%. Incremental checkpointing and copy-on-write checkpointing were the most effective techniques in lowering the running time overhead. These techniques reduce the amount of data written to stable storage and allow the checkpoint to proceed concurrently with the execution of the processes. The overhead of synchronizing the individual process checkpoints to form a consistent global checkpoint was much smaller. We argue that these measurements show that consistent checkpointing is an efficient way to provide fault tolerance for long-running distributed applications.

1 Introduction

The parallel processing capacity of a network of workstations is seldom exploited in practice. This is due in part to the difficulty of building application programs that can tolerate the failures that are common in such environments. Consistent checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications without requiring additional programmer effort [26, 30]. With consistent checkpointing, the state of each process is saved separately on stable storage as a *process checkpoint*, and the checkpointing of individual processes is synchronized such that the collection of checkpoints represents a *consistent state* of the whole system [6]. After a failure, failed processes are restarted on any available machine and their address space is restored from their latest checkpoint on stable storage. Surviving processes may have to rollback to their latest checkpoint on stable storage in order to remain consistent with recovering processes [15].

Much of the previous work in consistent checkpointing has focused on minimizing the number of processes that must participate in taking a consistent checkpoint or in rolling back [1, 11, 15, 17]. Another issue that has received considerable attention is how to reduce the number of messages required to synchronize the consistent checkpoint [2, 5, 8, 16, 19, 24, 28, 29]. In this paper, we focus instead on the overhead of consistent checkpointing on the failure-free running time of distributed application programs. We report measurements of an implementation of consistent checkpointing and analyze the various components of the overhead resulting from consistent checkpointing.

The overhead of checkpointing during failure-free computation includes (1) the cost of saving the checkpoints on stable storage, (2) the cost of interference between the checkpointing and the execution of processes, and (3) the cost of the communication between processes required to ensure that the individual process checkpoints record a consistent system state. Stable storage for checkpoints is provided by a highly available network file server. The checkpoints cannot be saved on a local disk or in local nonvolatile memory since that would make them inaccessible during an extended outage of the local machine. The cost of saving the checkpoints to stable storage therefore includes both the cost of network transmission to the file server and the cost of accessing the stable storage device on the file server.

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-9116343, by the Texas Advanced Technology Program under Grant No. 003604014, and by an IBM Graduate Fellowship.

*Author's current address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Our implementation of consistent checkpointing runs on sixteen diskless Sun-3/60 workstations connected by a 10 megabit per second Ethernet. Our measurements show that consistent checkpointing can be implemented very efficiently, adding very little overhead to the failure-free execution time of distributed application programs. With a 2-minute checkpointing interval, the running time increased by less than 1% for six of the eight distributed application programs that we studied. The highest overhead measured was 5.8%. The most important factors affecting the performance were the interference between a process's checkpointing and its concurrent execution, and the amount of data saved with each checkpoint on stable storage. The synchronization of the individual process checkpoints to form a consistent global checkpoint added little overhead.

Section 2 of this paper describes our implementation of consistent checkpointing. In Section 3, we briefly describe the eight application programs used in our study. We report and analyze our performance measurements of this implementation in Section 4. In Section 5, we compare our research with related work, and in Section 6, we present our conclusions.

2 Implementation

The system is assumed to consist of a collection of fail-stop [23] processes. A process consists of a single address space, residing on a single machine, and all threads executing in that address space. On each machine, a *checkpoint server* controls the checkpointing of the local processes, and participates in the consistent checkpointing protocol.

2.1 Checkpointing a Single Process

The checkpoint of a single process includes a copy of the process's address space and the state maintained by the kernel and the system servers for that process. Instead of writing the entire address space to stable storage during each checkpoint, we use *incremental* checkpointing to reduce the amount of data that must be written. Only the pages of the address space that have been modified since the previous checkpoint are written to stable storage. This set of pages is determined using the *dirty* bit maintained by the memory management hardware in each page table entry.

Furthermore, we allow the application to continue executing while its checkpoint is being written to stable storage. However, if the application process modifies any of its pages during the checkpoint, the resulting checkpoint may not represent the state that the process had at any single point in time. We have considered two alternative solutions to this problem.

The first solution uses *copy-on-write* memory protection, supported by the memory management hardware [9]. At the start of an incremental checkpoint, the pages to be written to stable storage are write-protected. After writing each page to stable storage, the checkpoint server removes the protection from the page. If a process attempts to modify one of these pages while it is still protected, a memory protection fault is generated. The kernel copies the page into a newly allocated page of memory, removes the protection on the original page, and allows the process to continue. The newly allocated page is not accessible to the process. It is used only by the checkpoint server to write the original contents of the page to stable storage and is then deallocated. If insufficient memory is available to allocate a new page for handling the copy-on-write fault, the process is blocked until memory can be allocated. This scheme is similar to that used by Li et al. [18] in their concurrent checkpointing technique for small physical memories. Unlike our implementation, however, they did not implement incremental checkpointing.

The second solution that we considered uses *pre-copying* [12, 27]. If the number of pages to be written to stable storage is below some threshold, the pages are copied at once to a separate area in memory and are then written from there to stable storage without interrupting the process's execution. Otherwise, a "pre-copying" pass is made over the process's address space, writing the modified pages from the process's address space to stable storage. The process continues to execute and can freely modify any of these or other pages during the pre-copying pass. Once these pages have been written to stable storage, the number of modified pages in the address space is reexamined. If it is still above the threshold, additional pre-copying passes are performed, up to a defined maximum number of passes. If the maximum number of passes has been exceeded, the process is suspended while the remaining modified pages are written directly from its address space to stable storage.

The pre-copying method avoids the expense and complication of handling copy-on-write faults, but may need to write some pages to stable storage more than once, if they are modified again during a pre-copying pass. In addition, pre-copying may need to suspend the process in order to complete the checkpoint, if additional pages of the address space are being modified too quickly by the process during pre-copying passes.

We have implemented checkpointing using each of these two methods and compared their performance. Our measurements show that the overhead introduced by copy-on-write checkpointing is always less than or equal to that introduced by pre-copying checkpointing. For example, for one application, the time required to write a checkpoint with pre-copying was 40% higher than with copy-on-write. Therefore, we chose copy-on-write for our implementation of consistent checkpointing. All measurements reported in the remainder of this paper were performed with the copy-on-write implementation.

2.2 Consistent Checkpointing

One distinguished checkpoint server acts as a *coordinator* and sends messages to the other servers to synchronize the consistent checkpoint. Each process maintains one *permanent* checkpoint, belonging to the most recent consistent checkpoint. During each run of the protocol, each process takes a *tentative* checkpoint, which replaces the permanent one only if the protocol terminates successfully [15]. Each consistent checkpoint is identified by a monotonically increasing Consistent Checkpoint Number *(CCN)*. Every application message is tagged with the *CCN* of its sender, enabling the protocol to run in the presence of message re-ordering or loss [5, 16]. We use this checkpointing protocol both for its simplicity and because we have found that it performs well in our environment.

The protocol proceeds as follows:

1. The *coordinator* starts a new consistent checkpoint by incrementing *CCN* and sending *marker* messages [6] that contain *CCN* to each process in the system.

2. Upon receiving a *marker* message, a process takes a tentative checkpoint by saving the process's kernel and server state and writing the modified pages of the address space to the checkpoint file, as explained in Section 2.1. The tentative checkpoint is written concurrently with the process's execution.

   A process also starts a tentative checkpoint if it receives an *application* message whose appended *CCN* is greater than the local *CCN*. Since this message was transmitted *after* its sender had started participating in the consistent checkpoint, the receiver must checkpoint its state before receiving this message in order to maintain the consistency of the global checkpoint.

3. After the tentative checkpoint has been completely written to stable storage, the process sends a *success* message to the coordinator.

4. The coordinator collects the responses from all processes, and if all tentative checkpoints have been successful, it sends a *commit* message [10] to each process; otherwise, it sends an *abort* message. When a process receives a *commit* message from the coordinator, it makes the tentative checkpoint permanent and discards the previous permanent checkpoint. When a process receives an *abort* message, it discards its tentative checkpoint.

2.3 Stable Storage

Each process checkpoint is stored as a file on a shared network file server. The file server structures the disk as a sequential log in order to optimize write operations [21]. Files that store different checkpoints of the same process physically share data blocks, in order to efficiently store the incremental changes to the checkpoint file. When a process records a tentative checkpoint, it writes the pages of its address space that have been modified since its last checkpoint to a new file. The remaining data blocks, which represent the portions of the address space not modified since the previous checkpoint, are automatically shared with the older checkpoint files of that process. Each file logically contains a complete image of the process's address space. When a checkpoint file is deleted, only the data blocks that are not shared with other checkpoint files are discarded.

In order to protect against a failure of the primary server, the checkpoint files are also saved on a backup file server. During the period of low load between two consecutive consistent checkpoints, the primary file server updates the backup's state.

3 The Application Programs

We chose the following eight long-running, compute-intensive applications, representing a wide range of memory usage and communication patterns:

- `fft` computes the Fast Fourier Transform of 16384 data points. The problem is distributed by assigning each process an equal range of data points on which to compute the transform.

- `gauss` performs Gaussian elimination with partial pivoting on a 1024×1024 matrix. The problem is distributed by assigning each process a subset of the matrix columns on which to operate. At each iteration of the reduction, the process which holds the pivot element sends the pivot column to all other processes.

- `grid` performs an iterative computation on a grid of 2048×2048 points. In each iteration, the value of each point is computed as a function of its value in the last iteration and the values of its neighbors. This application occurs in the kernel of many fluid-flow modeling algorithms. The problem is distributed by assigning each process 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.

- `matmult` multiplies two square matrices of size 1024×1024. The problem is distributed by assigning each process a portion of the result matrix to compute. No communication is required other than reporting the final solution.

- `nqueens` counts the number of solutions to the n-queens problem for 16 queens. The problem is distributed by assigning each process an equal portion of the possible positions of the first two queens. No communication is required other than reporting the total number of solutions found at completion.

- `prime` performs a probabilistic test of primality for a 64-digit integer, using the Pollard-Rho method. A master process distributes work from a task queue to each slave process. Each slave process communicates only with the master, and the master announces the number's factors that have been discovered at completion.

- `sparse` solves a sparse system of linear equations in 48000 unknowns, using a variation on the iterative Gauss-Seidel method. The system is sparse in that less than 0.25% of each row in the matrix is nonzero. The problem is distributed by assigning each process an equal subset of the unknown variables. After each iteration, each process sends the new values of its assigned unknown variables to all other processes.

- `tsp` solves the traveling salesman problem for a dense map of 18 cities, using a branch and bound algorithm. A main process maintains the current best solution and a task queue containing subsets of the search space. The main process assigns tasks from the queue to the slave processes. When a slave process finds a new minimum, it reports the path and its length to the main process. The main process updates the current best global solution, if necessary, and returns its length to the slave process.

4 Performance

4.1 Overview

Our implementation of consistent checkpointing runs on an Ethernet network of 16 diskless Sun-3/60 workstations. Each workstation is equipped with a 20-MHz Motorola MC68020 processor and 4 megabytes of memory, of which 740 kilobytes are consumed by the operating system. These machines run a version of the V-System distributed operating system [7] to which we have added our checkpointing mechanisms. Our experimental environment also includes two shared Sun-3/140 network file servers, each using a 16-MHz MC68020 processor and a Fujitsu Eagle disk, on which the checkpoints are written. The checkpoint data of a single process can be written to the file server over the network at a rate of about 550 kilobytes per second. All measurement results presented in this paper are averages over a number of trials. Standard deviations for all measurements were under 1% of the average.

All measurements of the eight application programs were made with the execution distributed across 16 machines, with one process per machine. The running times range from about 48 minutes for `gauss` to about 3 hours for `fft`, and the total amount of memory used across the 16 machines ranges from 656 kilobytes for `nqueens` to 47 megabytes for `sparse`. Table 1 summarizes the running time and the memory requirements of each application.

Program Name	Running Time (minutes)	Per Process Memory (Kbytes)		
		Code	Data	Total
`fft`	186	21	555	576
`gauss`	48	20	576	596
`grid`	59	21	2163	2184
`matmult`	137	20	2348	2368
`nqueens`	77	18	22	40
`prime`	53	38	74	112
`sparse`	65	22	2954	2976
`tsp`	73	21	27	48

Table 1 Application running time and memory requirements.

4.2 Checkpointing Overhead

4.2.1 Measurements

Table 2 presents a comparison between the running times of the application programs when run without checkpointing and when run with consistent checkpointing with a 2-minute checkpointing interval. We believe this choice of checkpoint interval is conservative. In practice, we expect longer checkpoint intervals to be used. In that sense, our measurements overestimate the cost of consistent checkpointing, since longer checkpoint intervals reduce failure-free overhead.

Some additional performance statistics are provided in Table 3. The *data written* column represents the average amount of data written to stable storage per consistent checkpoint (summed over all 16 processes). The *elapsed time* column shows the time from the initiation of the checkpoint to the receipt by the coordinator of the last acknowledgement of its *commit* message. This time corresponds roughly to the period during which a process may incur copy-on-write faults due to checkpointing. The *copy-on-write faults* column gives the average number of such faults that occur per checkpoint in each process. The checkpoint's elapsed time is also the time during which a process may become blocked, waiting for a new page to become available to service a copy-on-write fault. The

Program Name	Without Checkp. (sec.)	With Checkp. (sec.)	Difference	
			(sec.)	%
`fft`	11157	11184	27	0.2
`gauss`	2875	2885	10	0.3
`grid`	3552	3618	66	1.8
`matmult`	8203	8219	16	0.2
`nqueens`	4600	4600	0	0.0
`primes`	3181	3193	12	0.4
`sparse`	3893	4119	226	5.8
`tsp`	4362	4362	0	0.0

Table 2 Running times with and without checkpointing.

Program Name	Total	Coord.	Per Process	
	Data Written (Mbytes)	Elapsed Time (sec.)	Copy-on Write Faults	Blocked Time (sec.)
fft	0.4	2.0	4	0.0
gauss	7.1	14.1	50	0.0
grid	35.0	60.2	122	0.1
matmult	0.9	3.3	3	0.0
nqueens	0.3	1.5	2	0.0
prime	0.7	2.8	4	0.0
sparse	13.5	25.7	44	5.5
tsp	0.2	0.2	2	0.2

Table 3 Additional performance statistics (per checkpoint).

blocked time column indicates the average amount of time that each process was actually blocked during each checkpoint.

4.2.2 Analysis

For all applications but `grid` and `sparse`, the effect of checkpointing on the application program performance is negligible. The overhead for `grid` is somewhat larger because that program modifies every point in the 2048×2048 grid during each iteration. As a result, most of the address space of each `grid` process is modified between any two consecutive checkpoints, and must be written to stable storage for each checkpoint. The `sparse` program has the most overhead, 226 seconds or 5.8% of the running time. Blocking is responsible for 176 of the 226 seconds of overhead: The program takes 32 checkpoints during its execution, and the average blocked time per checkpoint is 5.5 seconds (see Tables 1 and 3). The `sparse` program consumes about 95% of the available memory on each machine. The remaining pages of memory are quickly exhausted in servicing the copy-on-write faults during each checkpoint, causing the execution to block for extended periods.

The increase in failure-free running time as a result of checkpointing is affected primarily by the amount of free memory available on each workstation and by the amount of data to be written to stable storage. The amount of free memory available determines the effectiveness of copy-on-write in preventing the application program from blocking during a checkpoint. The amount of data written on stable storage determines the elapsed time required to complete the checkpoint. The elapsed time influences the number of copy-on-write faults that may occur and determines the period during which a process may be blocked.

4.2.3 Summary

Consistent checkpointing adds little overhead to the running time of the application programs. On average, the overhead is about 1%, with the worst overhead measured being 5.8%. We argue that this is a modest price to pay for the ability to recover from an arbitrary number of failures.

4.3 Copy-on-Write Checkpointing

4.3.1 Measurements

We use copy-on-write to avoid blocking the processes while the checkpoint is written on stable storage. To measure the effectiveness of this solution, we modified our checkpointing implementation such that a process remained blocked for the duration of its process checkpoint. We then measured the performance of the eight distributed application programs using this implementation and compared the performance to our copy-on-write implementation. These results are presented in Table 4.

4.3.2 Analysis

The measurements show that blocking the application program while the checkpoint is being written to stable storage is expensive. The performance degradation is dependent on the amount of checkpoint data to be saved, due to the latency in writing the data to the file server. For example, applications with large memory sizes to be checkpointed such as `grid` and `sparse` show high overheads (85% and 20%, respectively) when blocking checkpointing is used, but incur only small overheads (1.8% and 5.8%, respectively) with copy-on-write checkpointing. Applications with very small memory sizes such as `nqueens` and `tsp` show no measurable overhead at all with copy-on-write.

4.3.3 Summary

Using copy-on-write eliminates most process blocking during checkpointing and thus greatly reduces the overhead of consistent checkpointing. For programs using larger

Program Name	% Increase in running time	
	Blocking Checkpointing	Copy-on-write Checkpointing
fft	0.2	0.2
gauss	13.7	0.3
grid	85.0	1.8
matmult	3.7	0.2
nqueens	1.8	0.0
prime	2.9	0.4
sparse	20.0	5.8
tsp	1.8	0.0

Table 4 Blocking checkpointing vs. copy-on-write checkpointing: % increase in running time.

memory sizes, copy-on-write should become even more important.

4.4 Incremental Checkpointing

4.4.1 Measurements

The goal of using incremental checkpointing is to reduce the amount data written on stable storage during each checkpoint. We compared incremental checkpointing against full checkpointing, where the entire address space of each process is written to stable storage during each checkpoint. Tables 5, 6 and 7 compare the amount of data written to stable storage, the percentage increase in running time, and the elapsed time for full and incremental checkpointing.

4.4.2 Analysis

The applications can be subdivided into three categories with respect to incremental checkpointing: applications with a large address space that is modified with high locality (`fft`, `matmult` and `sparse`), applications with a large address space that is modified almost entirely between any two checkpoints (`gauss` and `grid`), and applications with a small address space (`nqueens`, `prime`, and `tsp`). For the applications in the first category, incremental checkpointing is very successful. For the applications in the second category, incremental checkpointing is much less effective, because most of the address space is modified between any two consecutive checkpoints. Finally, the small address spaces of the applications in the third category make any reduction in overhead insignificant.

4.4.3 Summary

Incremental checkpointing reduces the overhead for many applications. Since it is easy to implement and never makes performance worse, its potential gain justifies its inclusion in any checkpointing implementation.

Program Name	Amount of data written (Mbytes)		
	Full Checkpoint	Incremental Checkpoint	% Reduction
`fft`	9.4	0.4	96
`gauss`	9.4	7.1	24
`grid`	35.1	35.0	0
`matmult`	37.9	0.9	98
`nqueens`	0.6	0.3	50
`prime`	1.8	0.7	61
`sparse`	47.7	13.5	72
`tsp`	0.8	0.2	75

Table 5 Full vs. incremental checkpointing: amount of data written (Mbytes).

Program Name	% Increase in running time	
	Full Checkpoint	Incremental Checkpoint
`fft`	0.2	0.2
`gauss`	0.5	0.3
`grid`	2.0	1.8
`matmult`	1.8	0.2
`nqueens`	0.0	0.0
`prime`	0.9	0.4
`sparse`	17.0	5.8
`tsp`	0.0	0.0

Table 6 Full vs. incremental checkpointing: percentage increase in running time.

Program Name	Elapsed time (sec.)		
	Full Checkpoint	Incremental Checkpoint	% Reduction
`fft`	17.6	2.0	89
`gauss`	17.8	14.1	21
`grid`	60.2	60.2	0
`matmult`	66.1	3.3	95
`nqueens`	2.6	1.5	42
`prime`	4.0	2.8	30
`sparse`	86.9	25.7	70
`tsp`	2.2	0.2	91

Table 7 Full vs. incremental checkpointing: elapsed time (sec.).

4.5 Checkpoint Synchronization

4.5.1 Measurements

In order to create a consistent checkpoint, the processes in the system must synchronize their checkpointing such that the most recent checkpoint of each process records a consistent state of the system. In contrast, in *optimistic checkpointing* [2], each process takes checkpoints independently. The system attempts to construct a consistent system state from the available process checkpoints. Optimistic checkpointing avoids the overhead of checkpoint synchronization, but may lead to extensive rollbacks and the domino effect [2, 20, 22]. It also requires garbage collection of process checkpoints no longer needed.

To measure the effect of the synchronization on checkpointing overhead, we modified our implementation to use optimistic checkpointing. We measured the performance of the application programs using this modified implementation, such that each process takes the same number of checkpoints as in the experiment described in Section 4.2. Table 8 shows the percentage increase in running time for the application programs using both forms of checkpointing.

4.5.2 Analysis

For all applications, with the exception of `sparse`, the increases in running time as a result of either consistent checkpointing or optimistic checkpointing were within 1% of each other. For `sparse`, the overhead of optimistic checkpointing was 3.0% vs. 5.8% for consistent checkpointing. Optimistic checkpointing performed better for `sparse` because each process was able to write its checkpoint to the file server with little interference from other processes. In consistent checkpointing, all processes attempted to write their checkpoints at essentially the same time, increasing the load on the file server and slowing its response. Optimistic checkpointing performed worse on `gauss` than consistent checkpointing. This apparent anomaly is due to the global communication-intensive nature of the `gauss` program. The execution of a process slows down somewhat while it is being checkpointed, which may cause some delay in transmitting application messages. Each iteration of `gauss` requires global communication among the processes of the application to distribute the next pivot column. As a result, slowing down the execution of a single process tends to slow the entire application program waiting for messages from that process. With optimistic checkpointing, the checkpoints of separate processes are taken at different times, causing additional slowdown of the entire application. With consistent checkpointing, instead, all processes take a checkpoint at essentially the same time, causing only a single slowdown of the application.

4.5.3 Summary

The difference between the overhead introduced by optimistic checkpointing and that introduced by consistent checkpointing is small. Given the potential for extensive rollback and the domino effect with optimistic checkpointing, consistent checkpointing appears the method of choice for our environment.

Program Name	% Increase in running time	
	Optimistic Checkpointing	Consistent Checkpointing
`fft`	0.2	0.2
`gauss`	1.0	0.3
`grid`	1.6	1.8
`matmult`	0.1	0.2
`nqueens`	0.0	0.0
`prime`	0.2	0.4
`sparse`	3.0	5.8
`tsp`	0.0	0.0

Table 8 Optimistic vs. consistent checkpointing: % increase in running time

5 Related Work

Previous work in checkpointing has concentrated on issues such as reducing the number of messages required to synchronize a checkpoint [2, 5, 8, 16, 19, 24, 28, 29], limiting the number of hosts that have to participate in taking the checkpoint or in rolling back [1, 11, 15, 17], or using message logging to eliminate the need for synchronizing the checkpoints and to accelerate input-output interactions with the outside world [4, 13, 25]. There are very few empirical studies of consistent checkpointing and its performance.

Bhargava et al. [3] reported on the performance of checkpointing. They concluded that, in their environment, the messages used for synchronizing a checkpoint were an important source of overhead. Their conclusion is different from ours, because of the small size of the programs used in their study (4 to 48 kilobytes). For such small sizes, the overhead of writing data to stable storage is indeed negligible, making the communication overhead an important factor. For larger applications, the overhead of writing data to stable storage dominates.

Kaashoek et al. [14] implemented consistent checkpointing to add fault tolerance to Orca, a distributed shared object-oriented language. Their implementation takes advantage of the ordered broadcasts already present in the Orca runtime system to order marker messages with respect to application messages. Processes are blocked while their checkpoint is being written to stable storage. A limited form of incremental checkpointing is used: the application code is written to the checkpoint only once, but all data is written out on each checkpoint, whether modified or not. As can be seen from Section 4.3, for applications with a large amount of memory to be checkpointed, the cost of blocking checkpointing can be quite high. Furthermore, the results in Section 4.4 indicate that the amount of data written to stable storage can be reduced significantly by writing only modified pages to the checkpoint.

Li et al. [18] described several checkpointing methods for programs executing on shared memory multiprocessors. Their results showed that nonblocking copy-on-write checkpointing reduces the overhead for checkpointing programs running on shared memory multiprocessors. They did not implement incremental checkpointing, which we found to be an important optimization. They also did not address the problem of consistent checkpointing in distributed systems. We have shown that the cost of synchronizing process checkpoints to form a consistent system state is quite small.

6 Conclusions

We have presented performance measurements taken on an implementation of consistent checkpointing on an Ethernet network of 16 Sun 3/60 workstations. The results demonstrate that consistent checkpointing is an efficient approach for providing fault-tolerance for long-running distributed applications. With a checkpoint interval as short as 2 minutes, consistent checkpointing on average increased the running time of the applications by about 1%. The worst overhead measured was 5.8%. Detailed analysis of the measurements further demonstrates the benefits of nonblocking copy-on-write checkpointing and incremental checkpointing. Using copy-on-write allows the process to continue execution in parallel with taking the checkpoint. It avoids a high penalty for checkpointing for processes with large checkpoints, a penalty that reached as high as 85% for one of our applications. Using incremental checkpointing reduces the load on the stable storage server and the impact of the checkpointing on the execution of the program. Without incremental checkpointing, the worst overhead measured for any application increased from 5.8% to 17%. Synchronizing the checkpoints to form a consistent checkpoint increased the running time of the applications studied by very little, 3% at most, compared to optimistic checkpointing. In return, consistent checkpointing limits rollback to the last consistent checkpoint, avoids the domino effect, and does not require garbage collection of obsolete checkpoints.

Acknowledgements

We would like to thank John Carter, Alan Cox, Pete Keleher, and Kai Li for their comments on earlier drafts of this paper. We also wish to thank the referees for their suggestions.

References

[1] M. Ahamad and L. 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] B. Bhargava and S-R. Lian. Independent checkpointing and concurrent rollback recovery for distributed systems — an optimistic approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, October 1988.

[3] B. Bhargava, S-R. Lian, and P-J. Leu. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proceedings of the International Conference on Data Engineering*, pages 182–189, March 1990.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[5] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the 4th Symposium on Reliable Distributed Systems*, pages 207–215, October 1984.

[6] 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.

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

[8] F. Cristian and F. Jahanian. 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.

[9] 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.

[10] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

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

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

[13] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[14] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.

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

[16] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[17] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the International Conference on Data Engineering*, February 1988.

[18] 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.

[19] 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.

[20] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[21] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[22] D.L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[23] 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.

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

[25] R.E. Strom and S.A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[26] 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.

[27] 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.

[28] 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.

[29] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25:295–303, July 1987.

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