Short Notes

Low-Latency, Concurrent Checkpointing for Parallel Programs

Kai Li, Jeffrey F. Naughton, and James S. Plank

***Abstract*—This short note presents the results of an implementation of several algorithms for checkpointing and restarting parallel programs on shared-memory multiprocessors. The algorithms are compared according to the metrics of overall checkpointing time, overhead imposed by the checkpointer on the target program, and amount of time during which the checkpointer interrupts the target program. The best algorithm measured achieves its efficiency through a variation of copy-on-write, which allows the most time-consuming operations of the checkpoint to be overlapped with the running of the program being checkpointed.**

***Index Terms*—Checkpointing, fault tolerance, copy-on-write, multiprocessing, backward error recovery**

I. Introduction

This short note presents four algorithms for checkpointing and restarting parallel programs running on shared-memory multiprocessors. To test the efficiency of these algorithms, we implemented them on the DEC Firefly multiprocessor [29], and profiled their performance on five benchmark programs. The algorithms range from very simple to more complex, using techniques such as copy-on-write [9], [30] and buffering [26] to realize good performance on the multiprocessor. In the best of these algorithms, most of the checkpoint is taken in parallel with the target program's execution, and when it does interrupt the target, the interrupts are for small, fixed periods of time (under 0.1 s in our implementation).

The main use of checkpointing is to provide the mechanism for performing *backward error recovery,* a general means of fault tolerance defined in [1]. The strength of backward error recovery is its ability to provide fault tolerance in the presence of unanticipated faults—faults that were not envisioned in the design of the system. No other means of fault tolerance has this property.

Checkpointing can also be used as a means of process migration or coarse-grained job swapping. This is the intended use of the Condor checkpointer [19]. In fact, one can view backward error recovery as merely process migration to the same machine at a different point in time.

All of the checkpointing algorithms presented take a "full checkpoint"; they checkpoint the entire state of the target program. The alternative would be to take "incremental" checkpoints, which save only that portion of the state that has changed since the last checkpoint. We have concentrated on full checkpoints to test the worst-case behavior of these algorithms. The work involved in taking an incremental checkpoint is a subset of the work involved in taking a full checkpoint; therefore, if the algorithms exhibit good efficiency, high concurrency, and low latency in taking a full checkpoint, taking an incremental checkpoint can only exhibit better performance in such measures.

Manuscript received July 7, 1992; revised July 9, 1993. This work was supported in part by the National Science Foundation under Grants CCR-8814265 and IRI-8909795, and in part by the Digital Equipment External Research Program and Systems Research Center.

K. Li is with the Department of Computer Science, Princeton University, Princeton, NJ 08544 USA; e-mail: li@princeton.edu.

J. Naughton is with the Department of Computer Science, University of Wisconsin, Madison, WI 53706 USA.

J. Plank is with the Department of Computer Science, University of Tennessee, Knoxville, TN 37966 USA.

IEEE Log Number 9401208.

II. The Checkpointing Algorithms

The goal of a checkpointing algorithm is to establish a *recovery point* in the execution of the program, and save enough information to reconstruct the state of the program at this recovery point in the event of a failure. In the case of a uniprocessor, a checkpoint can be taken by freezing the processor and saving the state of the central processing unit (CPU) (i.e., the registers) and the state of memory to disk. Note that in saving the state of memory, one need not save the executable code itself, as this can be reconstructed from the executable file. To restore this recovery point, one merely reads the state of memory from disk and then restores the state of the CPU. Thus, checkpointing is exactly like generating a core file (which is, of course, a type of checkpoint). This is the approach taken by Condor [19].

To generate a checkpoint for a shared-memory multiprocessor, one can do the analogous things: Freeze all the processors, then save the states of all of the CPU's and the state of memory to disk. We implemented this simple algorithm and called it *sequential*, because it performs none of its work in parallel with the program that it is checkpointing.

We set three goals for a good checkpointing algorithm on a multiprocessor.

1) It must be reasonably efficient.
2) It must impose little overhead on the target program. In other words, it should attempt to be maximally concurrent.
3) What overhead it does impose must be of low latency; that is, it should interrupt the target program for only small periods of time.

We believe that reasonable values for these goals are as follows. The overall checkpoint time should be no more than twice the optimal checkpoint time. The checkpointing should add no more than 20% to the running time of the program during the time that it is checkpointing. The latency of interrupts should be kept below 0.1 s. We chose this value because any more might be perceived as noticeable to the user watching the program's execution.

The sequential checkpointing algorithm is clearly optimal in terms of overall checkpoint time, because it is limited solely by the duration of the disk writes. However, it has zero concurrency and is 100% latency. The second algorithm that we implemented attempts to improve the concurrency of checkpointing. We call it *main memory* checkpointing, because it freezes the processors, saves the checkpoint into a separate address space in main memory, and then restarts the processors. After starting the processors, the algorithm forks a new thread in the new address space that writes the checkpoint to disk. In this algorithm, the concurrency of checkpointing should be improved over the sequential algorithm, because the execution of the target program is overlapped with the writing of the checkpoint to disk.

To improve the latency of the main memory algorithm, we implemented a third algorithm, which uses *copy-on-write* [9], [30] to make the main memory checkpoint. Copy-on-write is a technique that uses a processor's virtual memory page protection hardware to

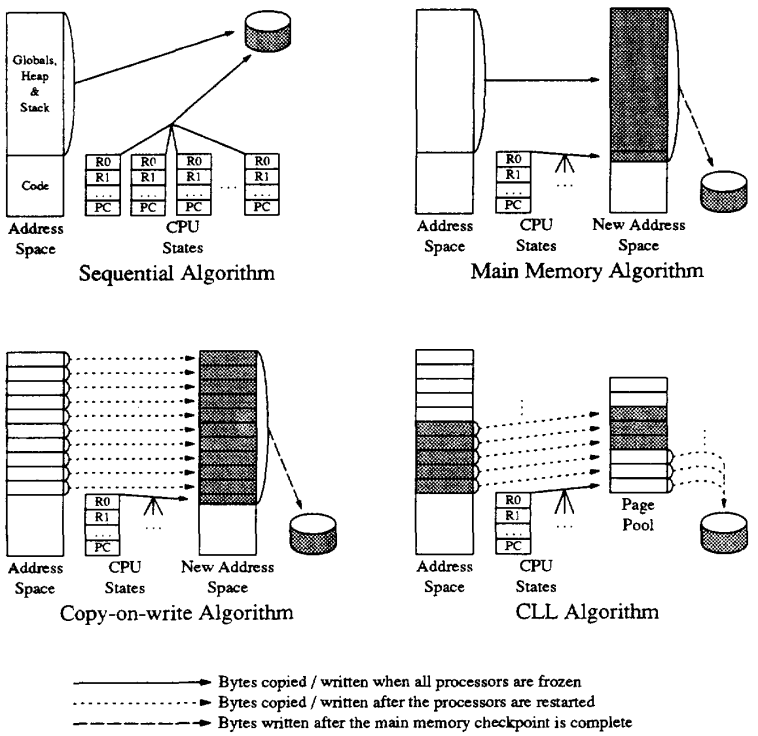


Fig. 1. The four checkpointing algorithms.

make a memory-to-memory copy with low latency. The copy-on-write checkpointing algorithm works as follows: First, it freezes the processors and copies their CPU states to the separate address space. Also, it sets the page protection bits of all pages in main memory to be "read-only." Next it unfreezes the processors and starts a separate *copier* thread that copies pages to the new address space and resets the pages' protection to "read-write." If a thread of the target program generates a page access violation, then it must write that page to the new address space before setting its protection to "read-write" and restarting. When the copier thread is done copying, the main memory checkpoint is complete, and the copier thread writes the checkpoint to disk.

Finally, we implemented a fourth algorithm which we call *concurrent, low-latency* (*CLL*) checkpointing. It improves upon the copy-on-write algorithm by adding buffering, a standard operating systems technique usually used to hide disk latency during file system writes. What the buffering does in this case is allow the checkpoint to be written to disk at the same time as it is being copied from memory. Specifically, the CLL algorithm allocates a fixed pool of pages (the buffer) in the second address space that the copier thread and page fault handlers fill, and that a new thread called the *writer* thread empties by writing the pages to disk. The writer and copier threads are both started immediately after the processors are unfrozen. All four algorithms are shown graphically in Fig. 1.

The improvements of the CLL algorithm over the copy-on-write algorithm should be twofold. First, the extra memory requirements of this scheme are fixed; they are the size of the page pool. The copy-on-write scheme needs extra space that is the same size as the target program's address space, and thus is more likely to cause thrashing. Second, the overall checkpoint time of the CLL algorithm should be less than that of the copy-on-write algorithm. This is because the CLL algorithm starts writing to disk as soon as the processors are restarted, instead of waiting until a complete main-memory checkpoint has been made.

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

III. Recovery

Recovering from a checkpoint is straightforward, and is the same for all four algorithms. The processors are frozen, and the contents of main memory are replaced with the contents saved in the checkpoint. The states of the CPU's are restored to their states at the recovery point, which are also saved in the checkpoint. When the processors are restarted, execution of the target program continues from the recovery point.

IV. Implementation

We have implemented all four checkpointing algorithms as well as recovery on a DEC Firefly multiprocessor. The Firefly is an experimental shared-memory multiprocessor developed at the DEC System Research Center [29]. A Firefly consists of four CVAX

processors, each with a floating point unit and a direct-mapped 64 kilobyte cache. The caches are coherent, so that all processors within a single Firefly see a consistent view of shared memory. The operating system for the Firefly is Taos [20], an Ultrix with threads and cheap thread synchronizations.

The Firefly upon which we tested our algorithms has 16 megabytes of physical memory, and a separate input–output (I–O) processor that shares memory with the other four processors, but also has a separate bus connecting the Firefly to I–O devices and the outside world. In our system, the I–O processor is connected to an RD54 disk drive.

The implementation is written in Modula-2+ [24], an extension of Modula-2. The user interface to the checkpointing routines is a single call to *setup_checkpoint()* at the beginning of the user's program. This call is used either to restore the program's execution to a saved recovery point or to specify how long the checkpointer should wait before interrupting the program to take a checkpoint. Freezing the processors is provided by Taos through a system call. In the absence of such a system call, one could freeze the processors either through interprocessor interrupts, or by protecting all the pages in memory to be "no access," and gaining control of the processors in the page fault handlers.

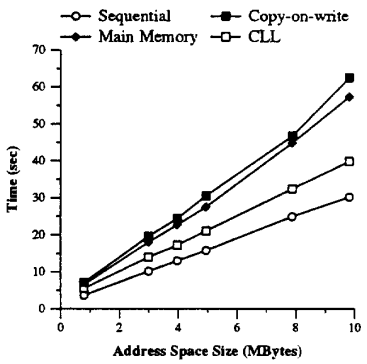
In taking a checkpoint only the user's address space is saved. The states of the kernel and the file system are not saved. The ramifications of this decision are that constructs that rely on kernel and external state, such as remote procedure calls and open file pointers, are not guaranteed to be recoverable. Thus, the programs that we tested were bereft of such constructs. We view these restrictions as tolerable for two reasons. First, the goal of our implementation is to examine the performance of checkpointing algorithms in regard to the metrics of speed, concurrency, and latency. The goal is not to write a production-level checkpointing system for the Firefly. Second, recoverable kernels have been studied and implemented [6], [21], as have checkpointers for uniprocessors that either provide recovery for read-only and sequential read-write files [19], or rewrite the UNIX™ file system to be completely recoverable [28]. We cannot justify taking the time to duplicate this work on Taos when the result is so tangential to our experiments.

One of the variables in our implementation is the page size. Although the actual page size on the Firefly's memory management unit is 512 bytes, we emulate different page sizes by varying the number of bytes that are copied to the second address space by the copier thread and by the page fault handlers. Larger pages will increase the time required to handle a fault, but they will also decrease the number of faults, and because of locality of reference, they may also decrease the rapidity at which faults occur.

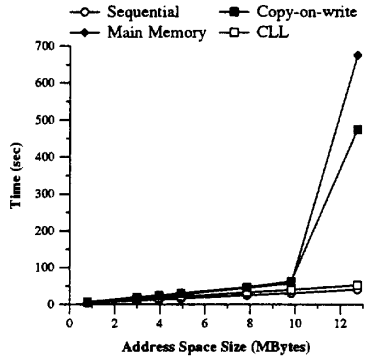
V. Experiments

For our initial experiments, we tested all four checkpointing algorithms on a parallel implementation of merge sort. We also checkpointed four other parallel programs: traveling salesman, matrix multiplication, pattern matching, and bubble sort. Since the results from experiments with these programs were so similar to the results with merge sort, we do not present them here. The merge sort program sort 250 000 indexed records, where the record size can be changed to modify the size of the program's address space.

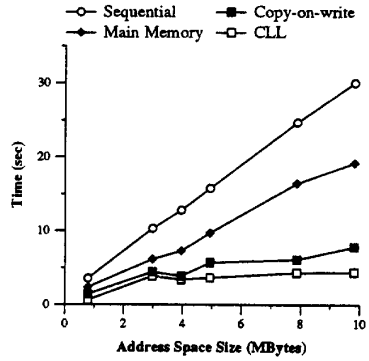
In all experiments, the four processors of the Firefly are partitioned so that the target program uses three and the checkpointer uses one. This is to measure the maximal concurrency of our checkpointing methods. The checkpointer waits for the target to run for 10 s, and then it takes one complete snapshot. For the results presented here, the page size is 8 kilobytes, and the page pool size is 1 megabyte. All times represent wall-clock time when the target program and checkpointer have exclusive use of the system. All times are averages of five or more runs.



Graph 1. Checkpoint time.



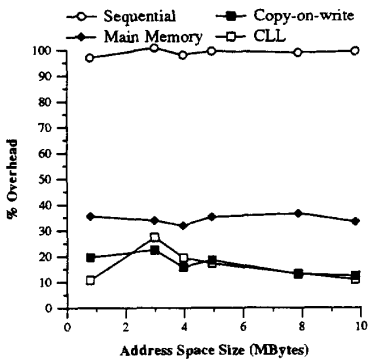
Graph 2. Checkpoint time for large address space.



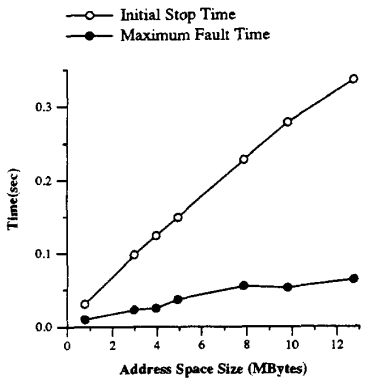
Graph 3. Checkpoint overhead.

Graphs 1 thorough 4 display the overhead imposed by the four checkpointing algorithms, as a function of the size of merge sort's address space. The total checkpoint time displayed in Graph 1 measures the elapsed time from the start of the checkpoint to its conclusion. Graph 2 extends Graph 1 to include the checkpoint time when the address space approaches the size of physical memory. The checkpoint overhead in Graph 3 is the amount of time by which the checkpoint increases the target's running time. Graph 4 displays the overhead as a percentage of the checkpoint's running time. This is equal to checkpoint overhead divided by the checkpoint time. Concurrency can be calculated as follows:

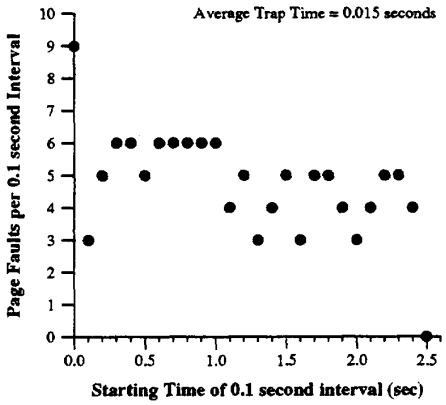
$$\text{concurrency} = 100\% - \text{overhead}.$$



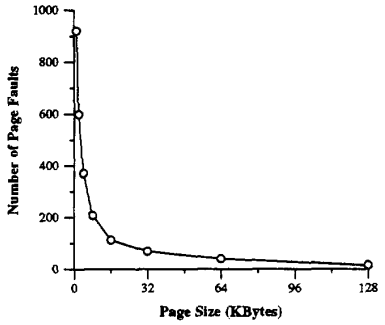
Graph 4. Checkpoint overhead percentage.



Graph 5. Latency data.



Graph 6. Frequency of page faults.



Graph 7. Number of page faults vs. page size.

It comes as no surprise that in all four algorithms, checkpoint time is proportional to the size of the address space. Also as expected, the sequential algorithm records the fastest checkpointing time, followed by the CLL algorithm. The other two algorithms take longer to record a checkpoint, because they wait for a complete copy of the checkpoint to exist in main memory before writing the checkpoint to disk. Of these two algorithms, the copy-on-write algorithm takes the longest, because of the extra work it spends processing page faults, and because it copies a page at a time.

As shown in Graph 2, when the address space approaches the size of physical memory, the main memory and copy-on-write algorithms exhibit severe thrashing, because their memory needs far exceed the size for physical memory. The other two algorithms keep their memory needs below the size of physical memory, and therefore do not suffer such rash penalties. It is worth noting that for all but the smallest address space tested, the pool of pages in the CLL algorithm became completely filled. Therefore, some worst-case data is included in the graphs.

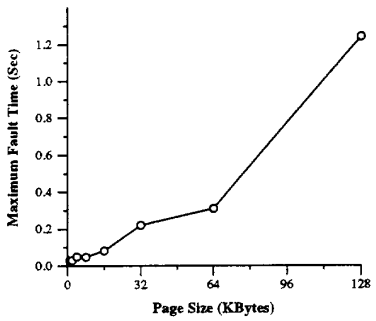
Graphs 3 and 4 show that the two algorithms based on copy-on-write display the smallest overhead and therefore the greatest concurrency. This is because these algorithms freeze the processors for the smallest amount of time. Taken as a whole, Graphs 1 through 4 show that the CLL algorithm is clearly the best of the four with regard to the combination of checkpoint time and concurrency. The results that follows pertain only to this algorithm.

Graphs 5 and 6 show latency data for the CLL algorithm. The overhead of checkpointing is divided into two parts: the time that all the threads are stopped initially to protect the address space and save the threads' states, and the time that the target threads are trapped, waiting to process page faults. The first curve in Graph 5 represents the initial stop time as a function of address space size, and the second represents the maximum time that any thread waits as a result of a page fault.

For address spaces up to 3 megabytes, the initial stop time is kept below 0.1 s. Moreover, for all address space sizes, the maximum page fault time is well below our low-latency goal of 0.1 s. Graph 6 displays the frequency of page faults over time for a run with a 4-megabyte address space. In this graph, the checkpoint is broken into 0.1 s intervals, and the number of page faults in each interval is plotted. The purpose of the graph is to show that work is indeed being accomplished by the target threads during the initial phases of the checkpoint.

Note that after an initial burst of nine page faults, the trapping frequency steadies at six faults per 0.1 s for the first second. Then it slows to about four traps per 0.1 s, until there are no more page faults. The average time to process a page fault is 0.015 s. Thus, during the first second of the checkpoint, the threads spend about 0.09 s/0.1 s interval processing page faults. Since there are three target threads, this means that the threads spend only one-third of their time processing page faults in the first second; the rest of the time is devoted to completing the merge. Thus, even at the beginning of the checkpoint, when one expects the highest frequency of page faults, the target still performs an adequate amount of work.

Graphs 7 and 8 display the results of altering the page size, again for a merge sort example with a 4-megabyte address space. As would be expected, the total number of page faults is proportional to the inverse of the page size, whereas the maximum time to process a trap increases almost linearly with the page size. Therefore, the ideal page size is one that significantly decreases the number of page faults while not significantly increasing the maximum page fault time.



Graph 8. Maximum fault time vs. page size.

These data show that a page size of 16 kilobytes is ideal. The number of page faults is kept relatively small (around 120 faults), and the maximum page fault time is still below 0.1 s. Note that this large page size has one other advantage: If the hardware were built with an actual page size of 16 kilobytes, then upon protecting the address space, it would have to change only $\frac{1}{32}$ the number of page table entries that it currently has to change. This should reduce the initial stop time (the first curve in Graph 5) by the same factor, which would bring it to well under 0.1 s for all address space sizes.

We omit the data for the results of checkpointing for the other four benchmark programs (traveling salesman, matrix multiplication, pattern matching, and bubble sort), except to say that their performance in all cases was the same or better than merge sort examples with similar address space sizes.

VI. Related Work

The bulk of work on and implementations of backward error recovery and fault tolerance in parallel and distributed systems has been in database and transaction-processing systems [5], [10], [17], [22], [25]. These schemes benefit from the fact that database computations can be viewed as consisting of atomic transactions. Since we concentrate on general-purpose parallel programs, no such computational model can be assumed.

General-purpose checkpointing has been studied and implemented on both uniprocessors and distributed systems. Proposals and overviews for uniprocessor checkpointing have been provided by [1], [15], [23]. In [28], a backward recovery implementation is described that focuses on a file system for UNIX that is fully recoverable. Reference [19] describes a portable checkpointing system called "Condor," which runs on any commercial uniprocessor and successfully checkpoints a majority of UNIX applications for the purpose of process migration. Neither implementation attempts to provide concurrency or low latency.

There has been much work on designing checkpointers for distributed systems [1], [3], [11]–[14] and for multicomputers [4], [18]. Here the focus of the work is on establishing a consistent recovery point, that is, either synchronizing the processors to define a global recovery point or postprocessing the processors' checkpoints to rebuild a plausible recovery point of the system. This is not a problem in shared-memory multiporcessor checkpointing, because the memory bus provides a simple place to enforce processor synchronization.

Staknis proposed a new memory design called sheaved memory [27] for supporting checkpointing in paged systems. In a sheaved memory, physical page frames can be bundled together, so that data written to one frame in the bundle is simultaneously written to all frames in the bundle. Removing a frame from its bundle would provide a snapshot of that memory page. Building such a memory would be quite costly, and it probably would be used only in special-purpose machines.

Of special note is a recent implementation by Elnozahy, Johnson, and Zwaenepoel [7]. They implement distributed checkpointing and recovery on a network of sixteen Sun 3/60's with a centralized file server. They implement the sequential and CLL algorithms, as well as both incremental and nonincremental checkpointing. They show that incremental checkpointing can reduce the amount of data being checkpointed by up to 97%. More relevantly to this short note, they corroborate our results by showing that the CLL significantly lowers overhead for incremental checkpointing. They do not show any results concerning the CLL algorithm for nonincremental checkpointing. The idea of using virtual-memory access protection hardware to achieve synchronization for the concurrent checkpointing was motivated by both shared virtual memory [16] and real-time, concurrent garbage collection [2].

VII. Conclusion

We have presented and implemented a low-latency concurrent algorithm for checkpointing parallel programs on stock shared-memory multiprocessors. The algorithm requires a constant amount of extra space, no change to the target parallel programs, and no special hardware assistance. Our experiment shows that this algorithm meets our performance goals on all five benchmarks: 80% to 97% of its checkpointing executes concurrently with the target programs, checkpoint time is always within 50% of optimal, and the latency is kept under 0.1 s. These goals were met by applying the techniques of copy-on-write [9], [30] and buffering [26] to the checkpointer.

Our algorithms are concerned solely with taking one snapshot with no prior history of the target's execution. For programs with large virtual address spaces, recording the changes between snapshots should be much more efficient than taking each snapshot separately. In the future, our scheme can be combined with [8] to use dirty page information and calculate snapshots incrementally. Such a method would not impose a large initial stop time. Moreover, the checkpointing time will be reduced because pages that have not been changed since the last snapshot will not be brought into physical memory and written out to disk. Results of distributed checkpointing from [7] support these assertions.

Acknowledgment

We would like to thank G. Swart for his help with modifying Taos, and M. Theimer for his helpful comments.

References

[1] T. Anderson and P. A. Lee, Fault Tolerence: Principles and Practice. Englewood Cliffs, NJ: Prentice-Hall International, 1981.
[2] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," In *ACM SIGPLAN'88 Conf. Programming Language Design Implementation,* 1988, pp. 11–20.
[3] K. Mani Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 3–75, Feb. 1985.
[4] F. Cristian and F. Jahanain, "A timestamp-based checkpointing protocol for long-lived distributed computations," in *Proc. 10th Symp. Reliable Distrib. Syst.*, 1991, pp. 12–20.
[5] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proc. ACM SIGMOD Int. Conf. Management Data,* 1984, pp. 1–8.
[6] F. Douglis and J. Ousterhout, "Process migration in the sprite operating system," in *Proc. 7th Int. Conf. Distrib. Computing Syst.*, 1987, pp. 18–25.
[7] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proc. 11th Symp. Reliable Distrib. Syst.*, 1992.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.