REAL-TIME, CONCURRENT CHECKPOINT FOR PARALLEL PROGRAMS

Kai Li
Jeffrey F. Naughton
James S. Plank

CS-TR-239-89

December 1989

Real-Time, Concurrent Checkpoint for Parallel Programs*

Kai Li Jeffrey F. Naughton James S. Plank

December 21, 1989

Abstract

We have developed and implemented a checkpointing and restart algorithm for parallel programs running on commercial uniprocessors and shared-memory multiprocessors. The algorithm runs concurrently with the target program, interrupts the target program for small, fixed amounts of time and is transparent to the checkpointed program and its compiler. The algorithm achieves its efficiency through a novel use of address translation hardware that allows the most time-consuming operations of the checkpoint to be overlapped with the running of the program being checkpointed.

Introduction

This paper presents a checkpointing and restart algorithm for parallel programs running on commercial uniprocessors and multiprocessors. The algorithm runs concurrently with the target program, interrupts the target program for small, fixed amounts of time (under 0.1 seconds in our implementation) and requires no changes to the target's code or its compiler.

One use of a checkpointing algorithm is to allow long-running programs to be resumed after a crash without having to restart at the beginning of the computation. Of course, a programmer can always write his or her own checkpointing routines, but this will entail

- Writing code to synchronize the multiple threads of the target program before taking a checkpoint,
- Defining an external format for the checkpoint data to be written to disk,
- Writing code to dump the checkpoint data to disk, and

*This research was supported in part by the National Science Foundation under grant CCR-8814265 and by the Digital Equipment External Research Program and Systems Research Center. Kai Li and James Plank's address: Computer Science Department, Princeton University. Jeffrey Naughton's address: Computer Science Department, University of Wisconsin.

- Writing code to read in the checkpoint data from disk and resume the program after a machine crash.

Our algorithm eliminates the need for this, since checkpointing and resuming is provided transparently to the programmer. Our algorithm uses the virtual-memory hardware of traditional processors to implement a medium-grained synchronization that is coarse enough to be efficient, yet fine enough to keep latency low.

A real-time concurrent checkpointing algorithm can be used in object-oriented systems, main-memory based database systems, and persistent programming environments. Without an efficient, real-time checkpointing algorithm these systems cannot provide persistence without degrading system performance or causing the system to "freeze" periodically while a checkpoint is taken.

Another application for a real-time, concurrent checkpoint algorithm is parallel program debugging. Multiple checkpoints of an execution of the target program can be used to provide "playback" for debugging; an efficient, real-time checkpoint algorithm can provide this playback with minimal impact on the target program execution.

To test the efficiency of our algorithm, we implemented two versions of our algorithm on the DEC Firefly multiprocessor [TS87], and profiled the performance of the checkpointing algorithm on five benchmark programs. One version of our algorithm is intended for machines with large real memories; the other for machines with smaller real memories. Additionally, we implemented two simple algorithms, one not concurrent, the other not real-time, against which to compare our real-time concurrent algorithm.

Related Work

In the past, simple checkpointing and recovery have been proposed and used for programs and transactions to recover from system crashes [Ran75,AL81,Lam81].

More complicated checkpointing schemes have been proposed in main-memory database and transaction-processing systems. Checkpointing in a main-memory database system is used for recovering data after a crash, rather than resuming a computation after a crash. Thus, instead of trying to record an actual snapshot, a checkpoint for a database need only record a database state that could have been reached through some serializable schedule of the transactions which have committed at the point of system interruption. To achieve this, knowledge of the data structures such as records or objects can be used.

DeWitt et al. [DKO*84] describe a concurrent algorithm that uses timestamps for memory pages to identify which pages should be checkpointed, and to ensure a consistent checkpoint. Pu [Pu85] describes a two-color algorithm that uses two colors to coordinate the checkpointer and transaction execution. Hagmann [Hag86] describes a "fuzzy checkpoint" algorithm that

requires a log in addition to the checkpoint for correct system restart. Li and Naughton [LN88] give a multiprocessor checkpointing algorithm that maintains a copy of the database on a checkpoint processor which alternates between installing logged updates in batch mode and checkpointing its copy of the database. Hence, it requires no synchronization between running transactions and the checkpointer. More details about some of these algorithms, and a performance comparison, appear in [SGM87].

Recently, checkpointing has been used in in parallel and distributed debugging to support reversible execution of a single process by saving interim execution states [FB89,PL89,Wit89]. In Feldman and Brown's debugger Igor [FB89], checkpointing is done incrementally by saving those pages which have been changed since the last checkpoint. The debugger uses virtual memory management to detect changed pages and and reconstructs an execution state from a number of snapshots. Although the checkpointing method is real-time, it is sequential and runs on a uniprocessor. They did not pursue concurrent methods.

Staknis proposed a new memory design called sheaved memory [Sta89] 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 would probably be used only in special purpose machines.

The idea of using virtual-memory access protection hardware to achieve synchronization for the concurrent checkpointing were motivated by both shared virtual memory [LH86] and real-time, concurrent garbage collection [AEL88].

Sequential Checkpoint and Recovery

The simplest checkpointing algorithm is a sequential method that "stops-the-world" upon taking a snapshot. We use this method as a baseline algorithm by which to compare our real-time, concurrent algorithms. This sequential method first stops all threads of the target program. Next, it dumps the entire writable state of the target program to disk. This state includes the globals, the heap, the stacks for the threads of the target program, and enough system-specific thread information so that the threads can be restarted. The amount of state that must be saved for each thread will vary from operating system to operating system, but in general, it corresponds to the amount of state that must be saved for doing a process migration.

The recovery algorithm performs the opposite. It first reads in the globals and the heap. Next, it creates the appropriate number of threads and restores their stacks and other states. Finally, it starts the threads.

The snapshot of the program state must be taken in such a way that a computation starting

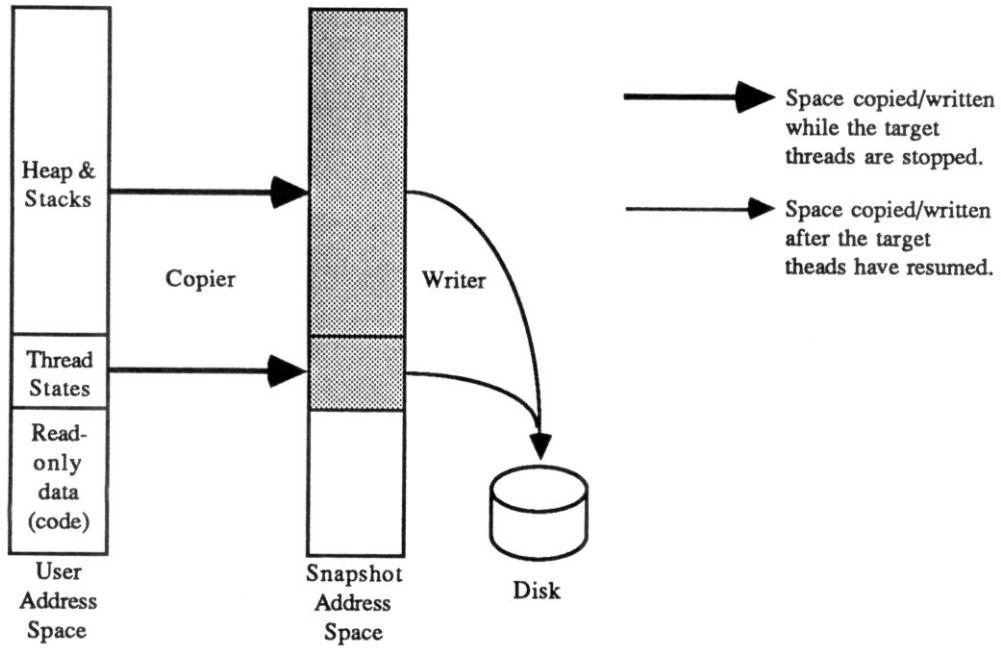


Figure 1: Concurrent Checkpoint with Large Memory.

from the snapshot will result in a correct execution of the target program. This will be the case if once a thread is stopped, it is not restarted until the states for all other threads in the computation have been saved.

If this condition is violated, an inconsistent checkpoint could result. For example, consider the following scenario. A thread T_1 is stopped, its state information saved, then restarted, at which point it interacts with another thread T_2 (perhaps by reading or writing a shared variable); then T_2 is stopped, and its state saved. Now in the checkpoint, the state for T_1 has been saved before the interaction of T_1 and T_2, while the state for T_2 has been saved after the interaction. By requiring that all threads be stopped and their states recorded before any thread is restarted, we guarantee that for any interaction between two threads in the program, either the result of the interaction appears in the saved state of both threads, or it appears in neither.

Concurrent Checkpoint with Large Memory

With a large physical memory, disk writing can be done concurrently with the execution of the target program. To do so, first all threads of the target program are stopped. Next, the writable state information for the program is copied to a separate address space, which we call the "snapshot address space."

After the copying to the snapshot address space is completed, the threads of the target

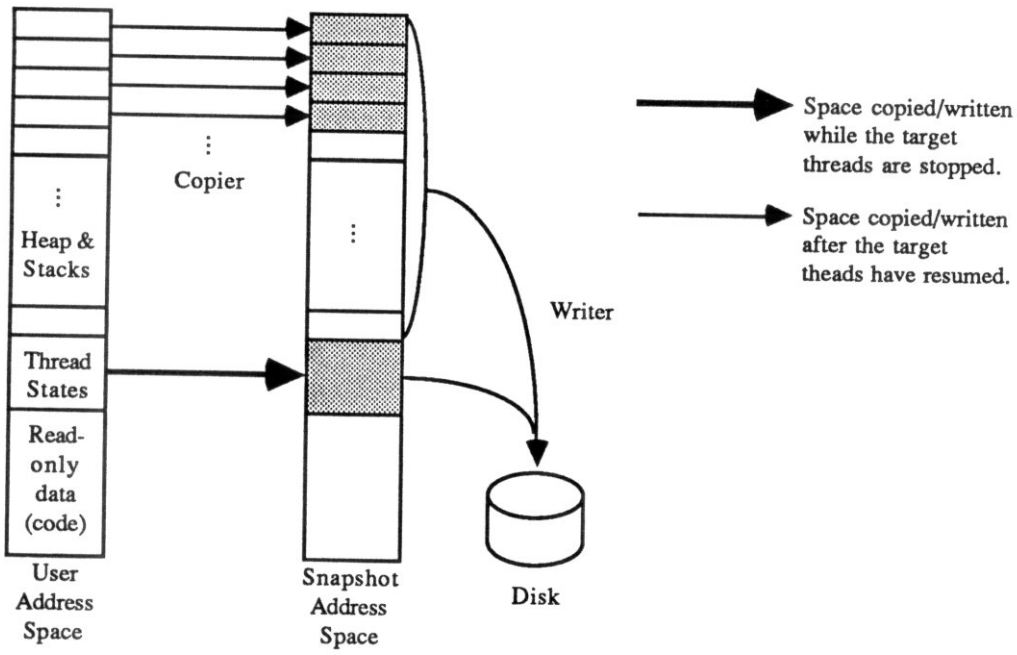


Figure 2: Real-time, Concurrent Checkpoint with Large Memory.

program are restarted; in parallel, a writer thread is started up, dumping the snapshot address space to disk. The writer and the checkpoint thread must synchronize so that the checkpointer does not begin the next checkpoint until the writer has written the previous checkpoint to disk.

Since the information written to disk under this algorithm is the same as that written to disk in the sequential method, the recovery method can be the same in both cases.

If the writable address space for the target program is large, the time for the memory to memory copy of the address space will also be large. This in turn means that a program being checkpointed under this scheme will be stopped for a long time while this copying is done. In many applications, this is unacceptable, which motivates the next algorithm.

Real-Time Concurrent Checkpoint with Large Memory

If a checkpointing algorithm is to be real-time, we must be able to guarantee that the threads of the computation are stopped for at most some small constant period of time, such as 0.1 second. To achieve this goal, we let the threads of the target program execute in parallel with the copying of the target program address space.

The difficulty here is to guarantee that the target program doesn't write to an address before that address has been checkpointed. Using standard synchronization techniques to coordinate the checkpointer and the target program would impose too great a performance penalty

on the target; instead, we use traditional virtual-memory access protection hardware for this coordination. This use of memory protection hardware follows that in [AEL88], where it was used to synchronize the collector and the mutator in a real-time concurrent garbage collection algorithm.

First, after stopping all threads, we set the protection bits for each page of the entire target program address space to "read-only." Next, we copy the minimal state information necessary for resuming the threads to the snapshot address space. This state information does not include the heap, globals, or stacks for the program, so it will be small.

At this point we resume all threads of the target program, and in parallel start a system thread called the "copier" thread. The copier thread sequentially scans the program address space, copying pages to the snapshot address space as it goes.

When the copier thread finishes copying a page, it changes its access rights from "read-only" to "read-write." When the threads of the target program are restarted, if we are lucky they will not access any pages until they have been copied and reset to "read-write." In general, however, the threads of the program will access some set of pages before they have been copied. When this happens, an illegal memory access fault will occur. At this point the copier thread immediately copies the page (even if it is not next in order in the sequential scan), sets the access for the page to "read-write," and restarts the faulting thread.

We call page copies due to memory access faults "forced copies," and copies due to the sequential scan of the address space "sequential copies."

Because of locality of reference, if a program tries to write a given page, it will often access neighboring pages next. We can try to take advantage of this behavior by writing groups of pages on forced copies. We will refer to such groups of pages as "segments." One goal of our implementation was to identify appropriate segment sizes.

After the entire program address space has been copied to the snapshot address space, a writer thread is started, dumping the snapshot address space to disk. When the snapshot address space is completely copied to disk, the checkpointer can begin another checkpoint.

This algorithm is attractive if the memory used by the target program is not too large. However, if the program requires a lot of memory relative to the real memory available in the machine, the algorithm can be inefficient, since the copying from program address space to snapshot address space will now entail paging of the address spaces. This motivates the next algorithm.

Real-Time Concurrent Checkpoint with Small Memory

Instead of first copying the program address space to the snapshot address space, we can incrementally copy the program address space directly to disk. This avoids the requirement of

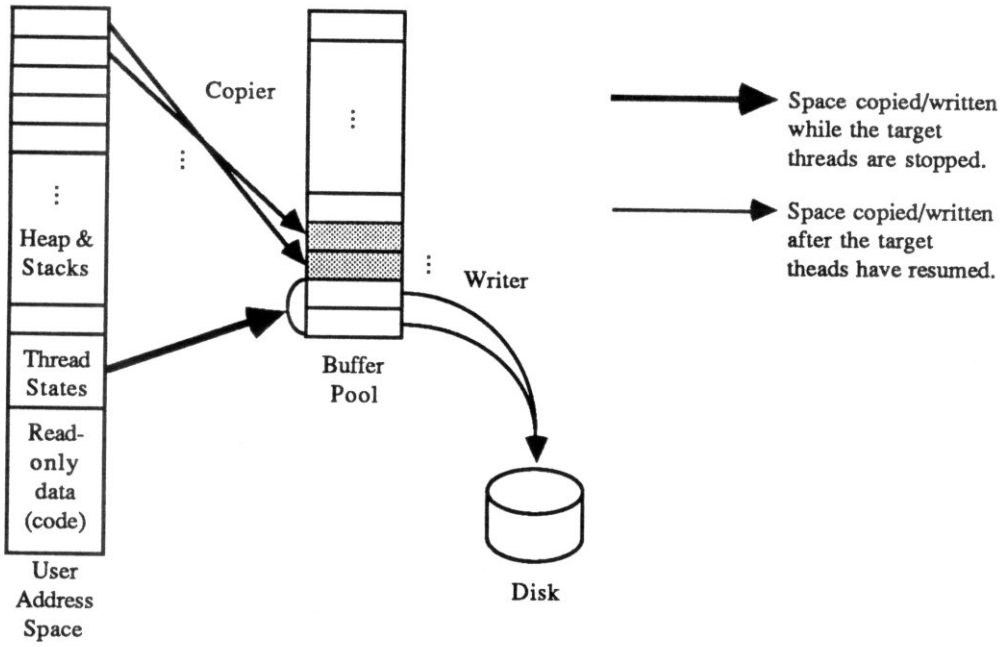


Figure 3: Real-time, Concurrent Checkpoint with Small Memory.

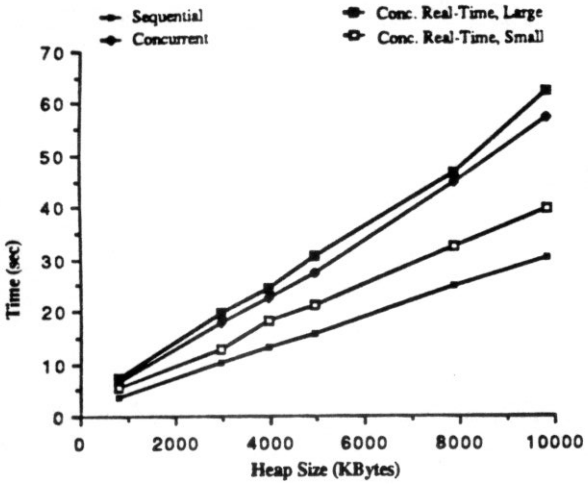
keeping two copies of the address space in memory.

A disadvantage of copying directly to disk arises in connection with forced copies. Recall that the faulting thread is stopped until the segment causing the fault is copied. With memory to memory copying, this time will be small, but with memory to disk, it can be unacceptably large.

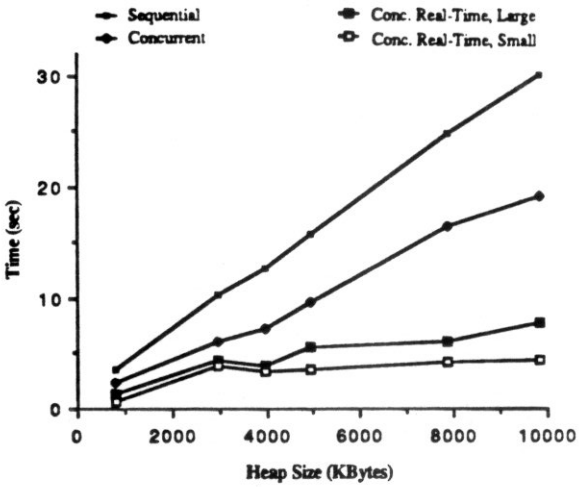
To reduce the time a thread is stopped during a forced copy, one can allocate a buffer pool of segment-sized sections of memory. Upon a fault, the copier thread copies the segment containing the faulting page to the buffer pool and the faulting thread is restarted. All the while, the writer thread takes buffers from the pool, dumps them to disk, and then frees them up for future use.

Implementation

We have implemented our algorithms on a DEC Firefly multiprocessor. The Firefly is an experimental shared-memory multiprocessor developed at the DEC System Research Center [TS87]. A Firefly consists of four CVAX processors, each with a floating point unit and a direct-mapped 64 KByte 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 [MS87], an Ultrix with threads and cheap thread synchronizations.



Graph 1: Checkpoint Time.

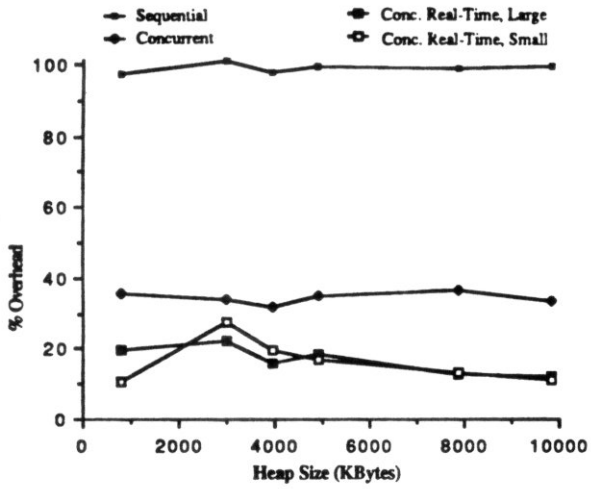


Graph 2: Checkpoint Overhead.

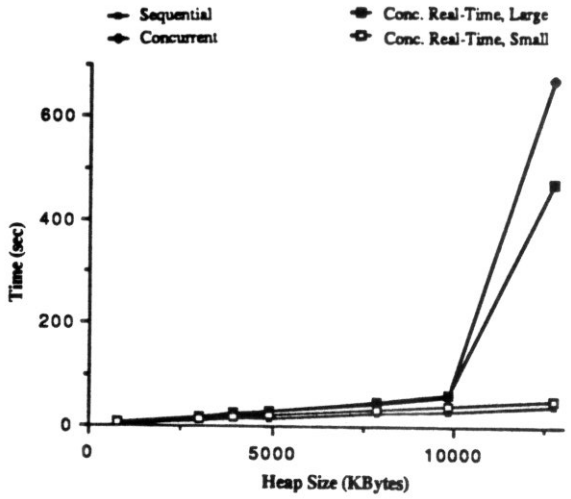
Checkpointing and recovery can be viewed as a special case of process migration – migration to another time and place within a single machine. Therefore, were the operating system to provide general purpose tools for process migration, our task would be somewhat simplified. However, like most other operating systems, Taos was not designed with process migration in mind, so we were forced to impose some restrictions on the kinds of targets that we could checkpoint.

The first of these is that the target program should contain no open file pointers; second, it should not be involved in any remote procedure calls at the moment of checkpointing. Both of these restrictions free the checkpointer from worrying about external events which would otherwise require some sort of log. Third, the Firefly retains much internal information, such as page tables and scheduling queues in a special address space called the "nub." As the nub cannot be explicitly reconstructed, our final restriction is that the target program be bereft of any nub-dependent information.

The implementation was written in Modula-2+ [RLW85], an extension of Modula-2. The checkpoint and recovery modules are both linked with the target program, and executed in the target's address space. This way, the code segment of the program need not be saved by the checkpointing thread, as it is automatically loaded into memory upon recovery. Moreover, Taos requires that every executable automatically start up seven or eight threads in addition to the target, for utilities such as garbage collection, trap handling and remote procedure calls. These threads need not be checkpointed either, as they too are automatically created by Taos upon recovery.



Graph #3: Checkpoint Overhead Percentage.



Graph 4: Checkpoint Time for Large Heaps.

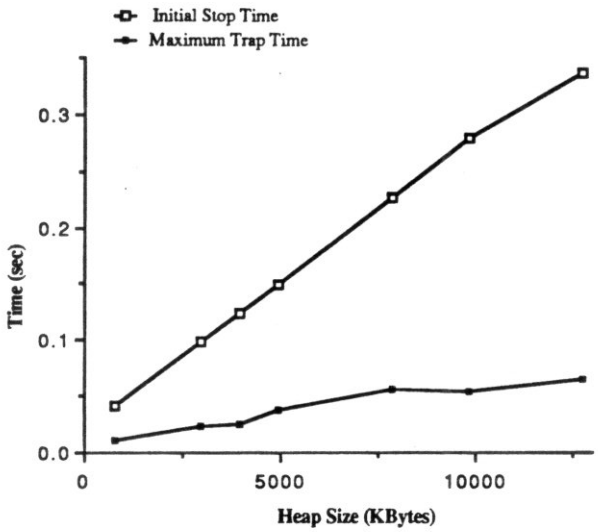
Experiments

For our initial experiments, we tested all four checkpointing algorithms on a parallel implementation of merge sort. (The results of other benchmark programs are included later). This program sorted 250,000 indexed records, where the record size could be changed to modify the heap size of the program. Merge sort was chosen to display initial results because of its relatively long running time (Between 45 and 60 seconds, depending on the record size), and because of its fairly random memory access patterns.

In all experiments, the four processors of the Firefly were partitioned so that the target program used three and the checkpointer used one; this was to measure the maximal concurrency of our checkpointing methods. The checkpointer would wait for the target to run for ten seconds, and then it would take one complete snapshot. For the real-time algorithms, a segment size of eight pages (1 page = 1KBytes) was used, and one MByte of memory was given to the buffer pool in the algorithm for small physical memories. All times represent wall-clock times rather than CPU time.

Graphs 1, 2, and 3 display the overhead imposed by the four checkpointing algorithms, as a function of the merge sort's heap size. The total checkpoint time, displayed in graph 1, measures the elapsed time from the start of the checkpoint to its conclusion. The checkpoint overhead in graph 2 is the amount of time by which the checkpoint increases the target's running time. Graph 3 displays the overhead as a percentage of the checkpoint's running time. This is equal to checkpoint time $\div$ checkpoint overhead.

The first observation we can draw from graph 1 is that in all four algorithms, checkpoint time is roughly proportional to heap size. This comes as no surprise, as the majority of the writable address space comes from the heap. Also coming as no surprise is that the sequential



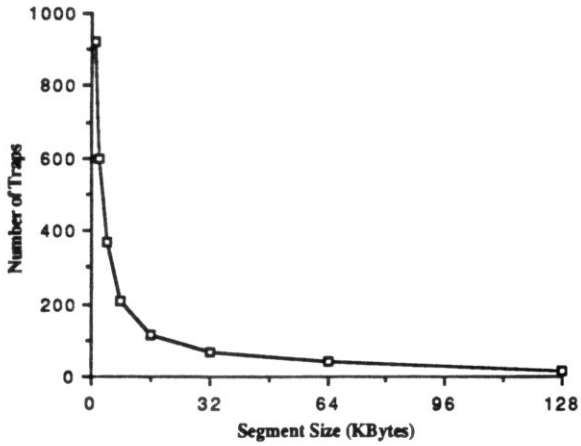
Graph 5: Latency Data.

baseline algorithm is the quickest; this is because it writes the snapshot directly to disk, and requires no memory-to-memory copying, and synchronization between threads.

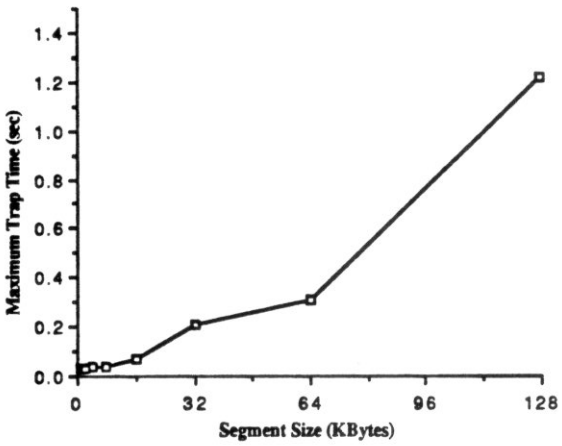
The two algorithms which copy to the snapshot address space take the longest time to complete their checkpointing. This is because of the extra overhead that Taos charges to manage different address spaces, and because the checkpointer must wait until all the data has been copied before it starts writing to disk. Of these two, the real-time algorithm takes the longest because of the extra work it spends processing page faults, and because it copies a page at a time. The concurrent, real-time algorithm with small physical memory is significantly faster than the other two because it overlaps the copying of data with the writing of it to disk.

Graphs 2 and 3 show that the two real-time algorithms significantly reduce the overhead imposed by the checkpointing, both in terms of the absolute value of the overhead (Graph 2), and as a percentage of the checkpointer's total work (Graph 3). Taken together, these three graphs suggest that the concurrent, real-time algorithm with small physical memory is the ideal one for checkpointing. It keeps the total checkpointing time low, so that more checkpoints can be taken in a given time interval, while keeping the overhead imposed on the target to a value around 10 percent.

A more convincing argument for this algorithm arises when the memory usage of the target program approaches the size of physical memory. Graph 4 shows what happens when a heap of 13Mbytes is used, and physical memory is filled to eighty percent with no checkpointing: The two algorithms using the snapshot address space thrash the virtual memory, as twice as many pages must be brought into physical memory. As a result, the checkpoint times increase to wholly unacceptable levels, and the value of the algorithm for small physical memories becomes apparent.



Graph 6: Number of Traps vs. Segment Size.



Graph 7: Maximum Trap Time vs. Segment Size.

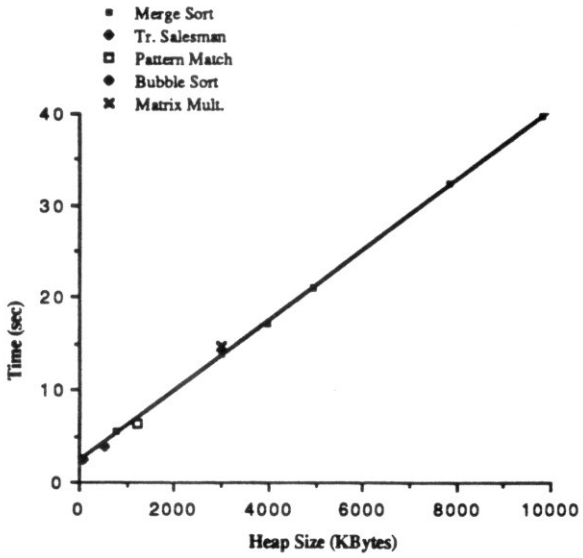
Graph 5 shows latency data to determine the real-time behavior of this algorithm. The overhead of checkpointing can be divided into two parts – the time that all the threads are stopped initially to set the protection bits and to save the threads' states, and the time that the target threads are trapped, waiting to process forced pages. The first curve in Graph 5 represents the initial stop time as a function of heap size, and the second represents the maximum time that any thread waits as a result of an access violation.

The results are acceptable. For heap sizes up to three mega-bytes, the initial stop time is kept below a tenth of a second. Moreover, for all tested heap sizes, the maximum trap time is well below our real-time goals. It's important to note that for the larger heap sizes (> 1.5 MB), the buffer pool gets filled up, so that part of the trap time is spent waiting for the writer thread to write a buffer to disk and free it for the forced copy. Thus, even some worst-case behavior fits into our real-time model.

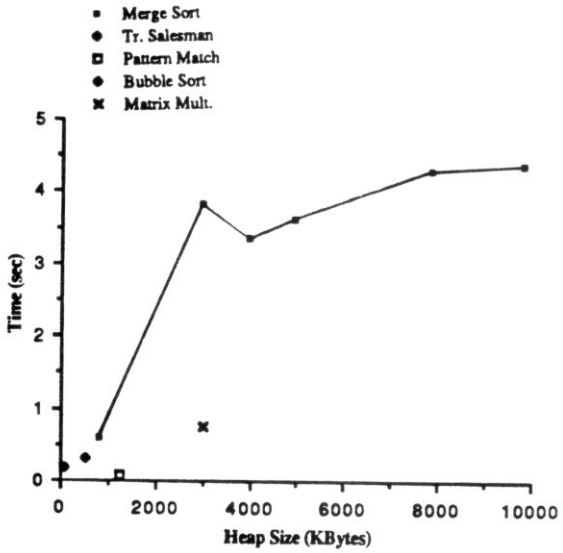
Graphs 6 and 7 display the results of altering the segment size of copying pages, for a merge sort example with a four MByte heap. As would be expected, the total number of forced copies is proportional to the inverse of the segment size, while the maximum time to process a trap increases almost linearly with the segment size. Therefore, the ideal segment size is one which will significantly decrease the number of forced copies, while not significantly increasing the maximum trap time. Our data show that a segment size of eight or sixteen pages to be ideal.

Finally, the checkpointer was tested with four other benchmark programs:

1. Traveling Salesman. This is a parallel program which solves the NP-complete Traveling Salesman problem for an instance with ten cities. Parallelism is achieved by each processor working on a subset of the possible paths. The program allocates a small heap of 64 pages, and takes 65.00 seconds to run.
2. Matrix Multiplication. This program multiplies two 200 X 200 integer matrices. The straightforward $O(n^3)$ algorithm is used, and parallelism is achieved by each processor



Graph 8: Checkpoint Time of Other Benchmarks.



Graph 9: Checkpoint Overhead of Other Benchmarks.

calculating a subset of the product's elements. The heap is 531 pages, and the running time is 42.85 seconds.

3. Pattern Matching. This parallel program takes a pattern array of 10 integers and finds the location of the 10 contiguous integers in a 2,400,000 element array which are the smallest edit distance from the pattern. Its heap size is 1,233 pages and it runs in 72.86 seconds.
4. Bubble Sort. This is a sequential program which uses the $O(n^2)$ bubble-sort algorithm to sort 3000 indexed records. Upon running, it allocates a heap of 3,000 pages, and takes 44.05 seconds.

Graphs 8 and 9 show the results of the concurrent, real-time algorithm for small physical memories on these benchmark programs, as compared with instances of merge sort with comparable heap sizes. Graph 8 displays that checkpoint time is essentially a function of heap size, no matter what kind of target it is checkpointing. Graph 9 shows that at least for the benchmarks tested, merge sort acts as a worst case target, because of its almost random patterns of memory access. The pattern-matching target, which has an extremely high locality of reference, produces almost no overhead at all, as does the bubble sort, which likewise tends to have more local memory access patterns.

Conclusion

We have presented a real-time, concurrent algorithm for checkpointing parallel programs on stock shared-memory multiprocessors. Through the use of virtual memory protection hardware,

the algorithm requires a constant amount of working buffers, no change to the target parallel programs, and no special hardware assistance. Our experiment shows that this algorithm performs well on all five benchmarks: 80 to 97% of its checkpointing executes concurrently with the target programs, while the latency is kept under 0.1 seconds. Our experiment also shows that the best segment size for such an algorithm is 8 or 16 KBytes for the DEC Firefly. However, for other machines, it may differ, as disk I/O speed, memory-to-memory transfer time, and the total number of processors will affect its value.

The weak point of our algorithm is that for targets with large heaps, the initial stop times are well over 0.1 seconds. Most of this time is spent changing the protection bits in page table entries of the heap. For example, in our experiment, since the CVAX MMU page size is 512 KBytes, the initial stop for a heap of 13 MBytes changes the protection bits of 26,000 page table entries. This takes about 0.35 seconds. If the MMU page size were the same as the optimal segment size (8KBytes), the initial time could be significantly reduced to about 0.02 seconds.

Our algorithms are solely concerned 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 will be much more efficient than taking each snapshot separately. In the future, our scheme can be combined with [FB89] 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 which haven't been changed since the last snapshot won't be brought into physical memory and written out to disk.

Another facet of our algorithm that we do not explore fully is the relationship between the copier thread and the trapped pages. First, the copier thread simply scans the pages of the heap in a linear order. Other orders, based on the pages which have faulted most recently might decrease the number of faults. Second, no priority is been placed on processing forced copies faster than normal copies from the copier thread. For example, one could envision two separate buffer pools, one for forced copies, and one for normal copies, where the writer thread tries to keep the forced copy pool empty most of the time. Such a scheme could significantly lower the maximum time to process traps.

Finally, to more generally implement our algorithm without having any constraints on the target programs, the operating system kernel data should be organized such that the states of all threads in an address space can be saved and restored easily. Such a requirement is no more than that of supporting process migrations.

References

[AEL88] A.W. Appel, J.R. Ellis, and K. Li. Real-time Concurrent Collection on Stock Multiprocessors. In *ACM SIGPLAN '88 Conference on Programming Language Design*

and Implementation, 1988. To appear.

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[DKO*84] D.J. DeWitt, R.H. Katz, F. Oiken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[FB89] S. Feldman and C. Brown. IGOR: A system for Program Debugging via Reversible Execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[Hag86] Robert B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

[Lam81] B.M. Lampson. Atomic Trasactions. In B.M. Lampson, M. Paul, and H.J. Siegert, editors, *Distributred Systems – Architecture and Implementation*, pages 246–264. Springer-Verlag, 1981.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[LN88] Kai Li and Jeffrey F. Naughton. Multiprocessor Main Memory Transaction Processing. In *Proceedings of the 1988 International Symposium on Database in Parallel and Distributed Systems*, December 1988.

[MS87] P.R. McJones and G.F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. Tech Report 21, DEC Systems Research Center, September 1987.

[PL89] D. Pan and M. Linton. Supporting Reverse Execution for Parallel Programs. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.

[Pu85] Calton Pu. On-the-fly, Incremental, Consistent Reading of Entire Databases. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 369–375, Stockholm, Sweden, August 1985.

[Ran75] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[RLW85] Paul Rovner, Roy Levin, and John Wick. On Extending Modula-2 For Building Large, Integrated Systems. Research report 3, DEC Systems Research Center, 1985.

[SGM87] Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, 1987.

[Sta89] Mark E. Staknis. Sheaved Memory: Architectural Support for State Saving and Restoration in Paged Systems. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–103, April 1989.

[TS87] C.P. Thacker and L.C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[Wit89] Larry D. Wittie. Debugging Distributed C Programs by Real Time Replay. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):57–67, January 1989.