SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(9), 1103–1120 (SEPTEMBER 1997)

Continuous Checkpointing: Joining the Checkpointing with Virtual Memory Paging

SHANG-TE HSU* AND RUEI-CHUAN CHANG†

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Rd., Hsinchu, Taiwan, R.O.C.
(email: gis79535@os.nctu.edu.tw, rc@iis.sinica.edu.tw)

SUMMARY

Checkpointing is a basic mechanism for backward error-recovery in fault-tolerant systems. A checkpointed process stops execution and saves its states to files periodically. To reduce the file sizes, only data modified between two consecutive checkpoint times is saved. However, existing approaches do not consider operating system paging activities; which, if ignored may double the number of disk accesses required to checkpoint non-resident dirty pages. In this paper, we propose *continuous checkpointing*, which combines the checkpoint facility with virtual memory paging operations. Thus, checkpointing is continuous during the lifetime of a process without extra overhead. Checkpoint size is no longer proportional to application size, but rather is bounded by resident dirty pages. Experimental results show that disk accesses can be reduced by about 80% when checkpointing large applications. ©1997 by JohnWiley & Sons, Ltd.

KEY WORDS: fault tolerance; incremental checkpoint; memory paging

INTRODUCTION

When checkpointing, systems periodically save applications' memory contents, machine states and other information.[1,2] If a failure occurs, the interrupted process can be restarted from the last saved states, and only a few minutes worth of computation are lost. Although some operating systems[3,4] and packages[5,6] already provide checkpoint/restart facilities, they still lack efficient methods for checkpointing the memory contents of running processes. Many scientific applications often process large amounts of data. Checkpointing all the memory contents of these applications not only wastes CPU time and I/O bandwidth, but also increases elapsed times and system overheads.

Several studies[6,7,8,9,10,11,12] have proposed ways in which to reduce checkpointing overheads. A promising approach is incremental checkpointing.[9,13] This method saves only the data modified between two checkpoint times, but does not take the operating system's paging activities into account. In modern computing systems, physical memories are shared by running processes. Operating systems must keep enough pages to satisfy the memory requirements of each process. Applications with large working sets unavoidably require memory paging. Ignoring memory paging may increase the number of disk accesses required to checkpoint

* PhD student.
† Also with the Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan, R.O.C.

CCC 0038–0644/97/091103–18 $17·50
Received 22 July 1996
Revised 17 March 1997
©1997 by John Wiley & Sons, Ltd.

non-resident pages – to bring the missing dirty pages to physical memory and then save them to checkpoint files.

In this paper, we propose *continuous checkpointing*, an approach that combines incremental checkpointing with virtual memory paging operations. Instead of reading dirty pages from paging files and then writing to a checkpoint file, all the dirty pages are checkpointed to the paging file. By taking advantage of ordinary memory paging operations, checkpointing is no longer a task confined to particular times, but becomes a continuous activity during the course of a process. Once a dirty page is paged out, it is tentatively saved to a paging file on disk. When checkpointing time arrives, the operating system flushes the rest of the dirty pages from physical memory to paging files and converts all tentatively saved pages to the permanently checkpointed pages. These checkpointed pages are then marked 'read-only'. If the operating system overwrites new data to a checkpointed page, the original checkpointed page is not modified; instead, a new disk slot is allocated for the newly arrived data. Because operating systems only save resident dirty pages at checkpointing time, the number of disk accesses required is smaller than with conventional incremental checkpoint methods. In commercial operating systems, the checkpoint mechanism in the KeyKOS[14,15] has been integrated with memory paging and achieves extremely high disk I/O performance.

Although tasks are continuously being performed before next checkpoint time arrives, it does not mean a checkpointed process can be restarted to any time point. An interrupted process can be only restarted to the time that commits checkpoint.

We used OSF/1 version 1.3 as our development environment, and tested two computation-intensive applications. OSF/1 is a UNIX operating system released by the Open Software Foundation. Version 1.3 of OSF/1 is a microkernel product based on the Mach 3.0 kernel developed by Carnegie Mellon University. Besides having 4.3BSD compatibility, OSF/1 also facilitates dynamic linking of execution programs with shared libraries. The microkernel approach also makes the system modular, flexible and portable. Although we used a microkernel-based operating system as our development environment, the *continuous checkpointing* technique can also be applied to conventional monolithic operating systems.

Our experimental results show that for applications with large amounts of processing data disk accesses can be reduced by about 80 per cent when solving the *All-Pair Shortest Paths* problem. When checkpointing applications multiply small matrices, all dirty pages reside in physical memory. Although the operating system must then flush the same number of dirty pages as conventional incremental checkpointing, our approach avoids page-fault handling costs in finding these pages.

MEMORY PAGING CONCEPT

In virtual memory systems, a process can use more memory space than is physically available by taking advantage of *memory paging*. In this mechanism, operating systems keep active memory pages and send pages that are currently not in use to the *pager*. The pager stores pageout memory pages in paging files, records their locations in a mapping table and gives every pageout page a unique identity number that the operating system can later read back. Depending on implementation, the pager can be part of the operating system or be another process running outside of the operating system. Figure 1 illustrates how the operating system handles memory paging.

Upon initialization, the operating system creates a *free-page pool* consisting of all available page memory. No physical pages are allocated to new processes initially, and all page table validation bits are *cleared*. When a process begins execution, the validation bit in its page

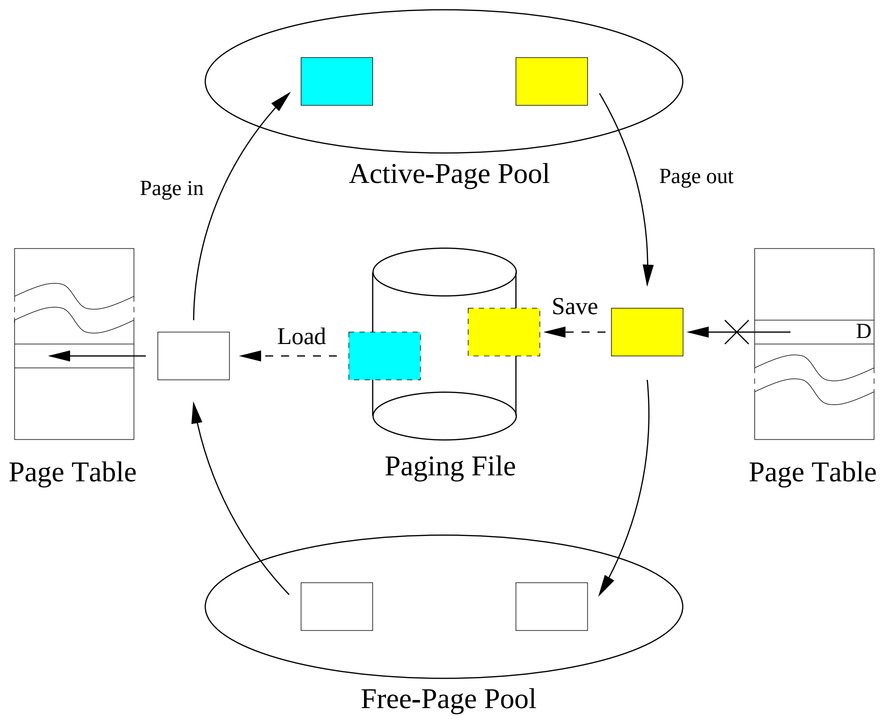

Figure 1. Memory paging

table initiates a *page fault* trap to the operating system. If the process has the right to access the faulting address, the operating system maps a free page from the free-page pool to the process's page table, loads the data, and places the allocated page into the *active-page pool*. If the process has no right to access the address, an error signal is returned to the faulting process. In order to keep free memory above a pre-determined threshold, operating systems choose pages to replace. If the page is clean and can be read back from the original execution file or backing storage, it is removed from the process's page table and flushed. If the page has been modified, it is sent to the pager for inclusion in the paging file first, then flushed from physical memory. This procedure is repeated until the number of free pages exceeds the pre-determined threshold.

Clearly, some processes' pages will be stored in the paging file before the system generates a checkpoint. Thus, including these pages in a checkpoint file requires additional disk accesses. If, for example, a process has 16 MB of modified data and only 8 MB of physical memory is available, then 8 MB of data will be stored in the paging file. Theoretically, this means that checkpointing the memory contents of this process will require I/O accessing of 24 MB of data: reading the 8 MB of paging file data, and writing a total of 16 MB of data to the checkpoint file, i.e. the dirty pages in the paging file must be passed along the I/O bus twice.

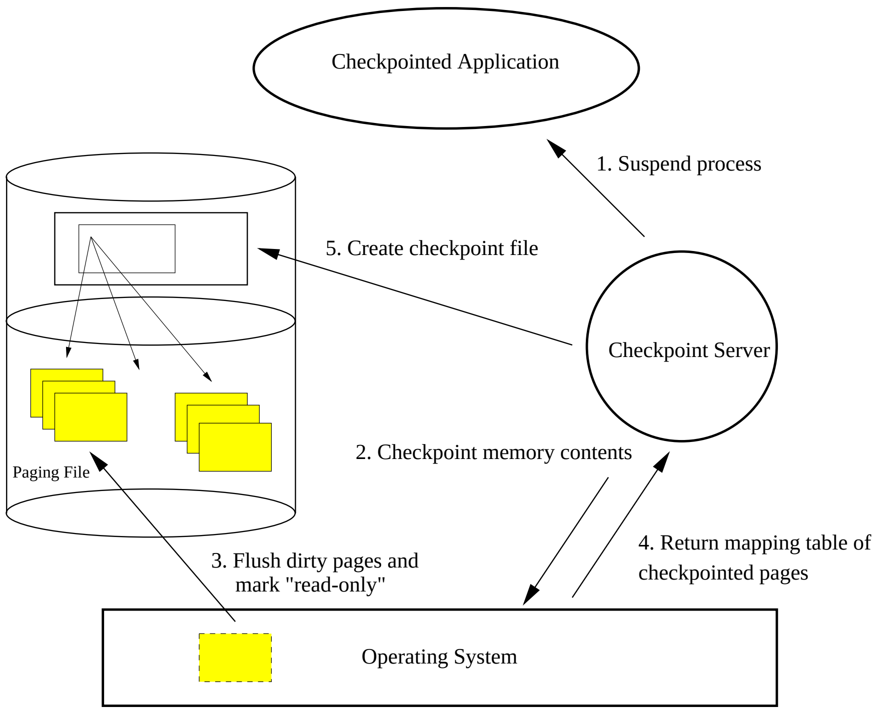
Checkpointed Application
1. Suspend process
5. Create checkpoint file
Checkpoint Server
Paging File
2. Checkpoint memory contents
4. Return mapping table of checkpointed pages
3. Flush dirty pages and mark "read-only"
Operating System

Figure 2. Continuous checkpointing architecture

CONTINUOUS CHECKPOINTING

Continuous checkpointing extends the virtual memory system to checkpoint every dirty page when the page is paged out. Instead of reading the dirty pages from the paging file and then saving to a checkpoint file, continuous checkpointing uses the paging file to store the checkpointed memory contents. Figure 2 shows the architecture of continuous checkpointing. A user-level checkpoint server provides multiple checkpointing policies and timing triggers. A checkpoint consists of two parts; one is the checkpointed memory contents in the paging file, the other is a checkpoint file containing machine states, process-related information and a mapping table that records the locations of checkpointed memory pages in the paging file.

Checkpoint phases

Dirty pages generated during a checkpoint interval are saved in the *continuous* phase and the *frozen* phase. In the *continuous* phase, every dirty page is saved when the page leaves physical memory. This phase is built on the pageout operation in the virtual memory system. The pageout daemon selects pages for eviction from the physical memory without regard to whether they belong to checkpointed processes. However, since normal pageout operations are used, the checkpoint facilities do not perform additional disk accesses during this phase.

In the *frozen* phase, when checkpointing time arrives, the operating system flushes dirty pages and marks them all ‘read-only’. After all dirty pages have been checkpointed, the checkpoint server generates a checkpoint file that stores the last process states, contents of machine registers and a mapping table that records the locations of all checkpointed dirty pages in the paging file.

To incrementally save modified process data, the checkpoint facility needs a way to identify the process’s dirty pages. Existing algorithms[6,10,11] use the virtual memory protection mechanism to find such dirty pages. A checkpointed process first has the `WRITE` permission to its memory space disabled. When the process wants to write to protected memory, it must generate a page fault. The operating system or checkpoint server then records this page as a dirty page. When checkpoint time arrives, any faulted pages are saved to a checkpoint file. Because each page fault induces overhead, conventional incremental checkpointing methods pay higher costs to find dirty pages.

In continuous checkpointing, the operating system scans the dirty bit in the page table entry to find checkpoint data. If a dirty bit is set, it indicates that this page needs to be flushed before checkpointing. Dirty bits in the page table provide enough information to identify which memory pages must be saved. Thus, the operating system does not need to set an extra protection bit on any memory page and pays no extra cost in fault handling. Therefore, continuous checkpointing incurs lower costs in finding dirty pages.

Paging space management

The conventional paging file is a temporary backing store for non-checkpointed processes. In checkpoint facilities, this paging file acts as permanent storage for checkpointed processes. The pager uses the shadow-paging technique[16] to manage the backing store. For each checkpointed process, the pager creates two mapping tables. One is the `working set` that records the non-checkpointed dirty pages on the disk, the other is the `shadow set` that records the dirty pages last checkpointed. Checkpointing simply marks all the disk slots pointed to by the `working set` as ‘read-only’, and then saves this mapping table to the checkpoint file.

The shadow-paging algorithm uses the copy-on-write algorithm for disk space allocation. Checkpointed pages indexed by the `working set` are marked ‘read-only’ first. If a new write request occurs, the pager pushes its entry from the `working set` to the `shadow set` and allocates a new disk slot for the request.

Figure 3 shows what happens when the operating system wants to overwrite a checkpointed page. A read-only bit in the `working set` map entry was set at the preceding checkpoint time. The operating system can read these read-only pages back when the checkpointed process accesses them again. If these pages are not modified after checkpointing, the disk space occupied by these pages can be used for paging and belongs to part of a checkpoint. However, if a physical memory page is modified and selected as the next pageout page, the pager first checks its read-only bit in the `working set`. If this bit is not set, the pager saves the pageout data to the disk space pointed to by the `working set` map entry. If this bit is set, the pager then pushes the map entry in the `working set` to the `shadow set` and then allocates a disk slot for the new data. If the `shadow set` already has a read-only entry, the disk space pointed to by the `shadow set` is released. The `shadow set` acts as a buffer for previous read-only entries. The pager uses the `shadow set` to keep the last unreleased checkpointed pages and minimize disk space utilization. A checkpointed process uses at most twice its image size in the paging file.

We maintain a single virtual-to-physical memory translation and provide two sets of disk slots in the paging file. This approach can easily be applied to other operating systems, and

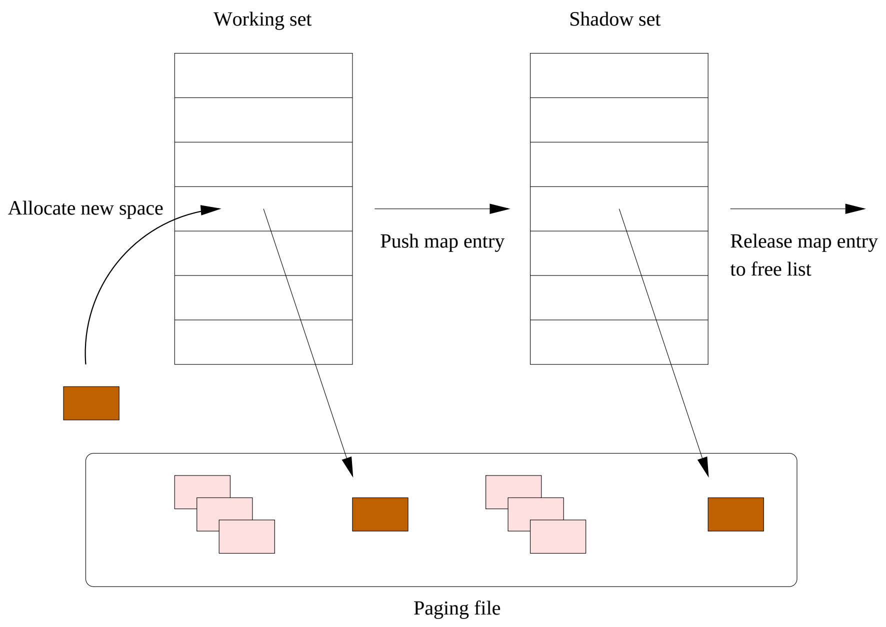

Figure 3. Pager allocates a new disk slot when new data is written to a checkpointed page. The map entry in the `working set` *is pushed to the* `shadow set`*. The disk slot pointed to by the* `shadow set` *is released to the free list*

most functions in virtual memory systems need not be changed. Our approach is different from Bowen and Pradhan's,[7,8] in which two mapping sets in the virtual-to-physical memory translation are used for every process. Details of their approach are described in the 'Related work' section.

Recovery from a failure

When a process is restarted after a failure, the checkpointed dirty pages in the paging file are reserved before the recovery has been completed. However, in most operating system, the paging file is temporary storage; its data is cleaned when the system is rebooted. Retaining checkpointed data is thus an important issue. A simple solution is to copy useful data to another file before the system cleans the paging file. Because the dirty-page mapping table is stored in the checkpoint file, the pager does not keep information on how to identify which disk slots are free for use, and it is difficult to reconstruct failed processes without identifying which disk slots are free. The recovery procedures must first check file system consistency, and then locate useful data from the mapping table in the checkpoint file.

To solve this problem, we propose a two-stage boot procedure at system startup. Each stage uses a different paging file, but only the paging file in the second stage, the *normal paging file*, contains checkpointed data. The paging file in the first stage, the *temporary paging file*, is only used to check file system consistency and copy checkpointed data. After all dirty pages have been copied, the system uses the *normal paging file* as the backing store for all processes, and then restarts the checkpointed processes.

To restart a process, the operating system first maps the process memory space into its initial state and then re-maps the memory contents with the checkpointed dirty pages. After all memories have been mapped, the operating system re-creates the process data structures and restores them to their checkpointed states. Finally, the operating system sets the machine registers to their last checkpoint states and restarts the failed process.

An important issue is how to migrate a continuously checkpointed process to another machine when the original machine cannot be restarted. This is dependent on implementations. In this paper, we present a command that reads the checkpointed data from the paging file and generates a restart file. The system can transfer restart files to another machine during an idle cycle. If the original system cannot be rebooted, another machine can take over the checkpointed processes. If users want to get restart files after processes have been killed, they can also use this command to get them. In our experiments, we used this command to verify checkpoint correctness.

IMPLEMENTATION

We implemented continuous checkpointing in OSF/1 version 1.3, which is a Mach-based microkernel operating system. We give an overview of the Mach kernel in the first subsection, and then describe implementation of the continuous checkpoint facility in the second subsection.

Mach kernel

OSF/1 1.3 is a Mach-based,[17] microkernel operating system. The Microkernel provides the basic mechanism and resource management. Other services are provided by user-level servers. At least two servers must be present in the system, a UNIX server that provides conventional UNIX services, such as file systems, tty and network communication, and a default pager that manages the virtual memory backing store.

There are five abstractions in Mach: task, thread, port, message and memory object. A task is an execution environment that contains a virtual address space, ports and threads. A thread is a system scheduling unit. A task may contain multiple threads for parallel execution. A conventional UNIX process is equivalent to a task with a single thread.

A port is a unidirectional communication channel. A task that holds *send rights* to a port can send messages to this port. The *task control port* is a special port that can suspend/resume tasks and obtain the execution states of these tasks.

Mach allows user applications to provide the semantics associated with portions of virtual address space. A memory object abstraction represents the non-resident state of a memory region. The task that responds to serve request messages and manages the backing store is called an *external pager*. A particular external pager called the *default pager* is the final arbiter for memory object requests that cannot be served by the user-provided external pager. Some memory regions, such as task stacks, created by the microkernel are also managed by the default pager. Complete descriptions of Mach and OSF/1 can be found elsewhere.[17,18,19,20,21,22]

Implementation issues

Our implementation consists of a modified microkernel, UNIX server, default pager and a separate checkpoint server (checkpointer), which makes it easier to debug the checkpointer.

Since the checkpointer must periodically stop processes, it is necessary to have the process-control capability of the UNIX server. To provide this capability, the UNIX server sends task-control rights to the checkpointer before spawning new processes.

The checkpointer is a multi-threaded server. When a process wants checkpoint service, the checkpointer creates a thread to checkpoint it and orders the microkernel to handle the continuous checkpointing. Because the checkpointer holds task-port rights to the checkpointed process, it is not necessary to create a service thread within the checkpointed process to suspend/resume the checkpointed process and obtain information from the microkernel.

After the checkpointer gets task-port rights to the process, the process is set to run. Users can specify use of either *incremental* or *snapshot* policies to checkpoint their processes periodically. When checkpoint time arrives, the checkpointer first stops the checkpointed process and issues a system call to synchronize memory contents.

When the microkernel receives the memory-synchronization request and the process being synchronized is a continuously checkpointed process, the microkernel flushes the resident dirty pages to the default pager and obtains a page map containing the locations of checkpointed pages. Resident memory pages are not unmapped; their dirty bits in the hardware page table are just cleared. The checkpointed application will not cause any page faults due to checkpointing procedures.

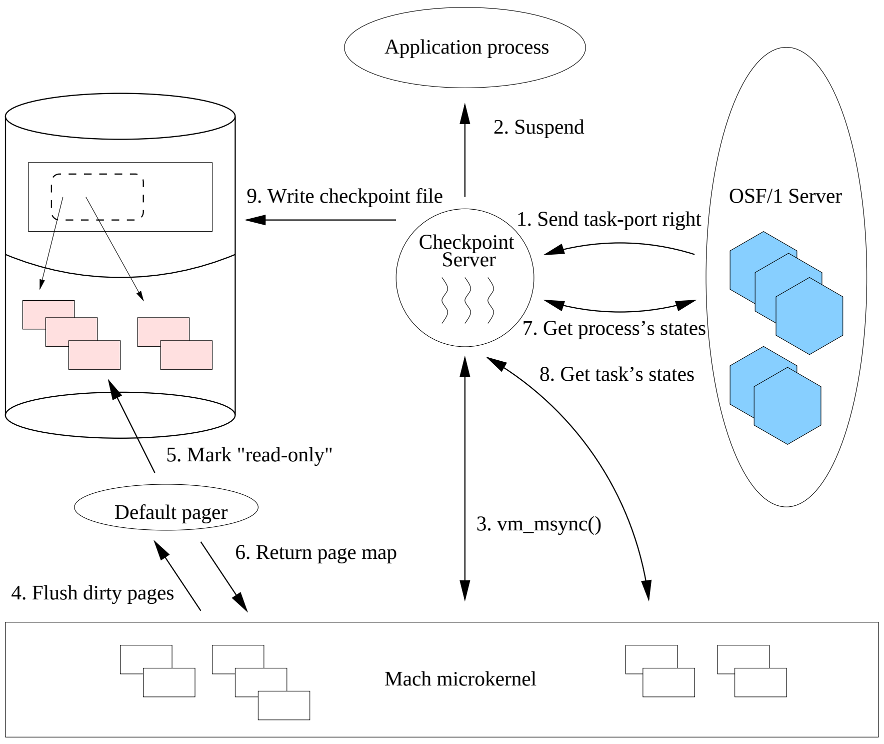

Figure 4. The steps of continuous checkpointing in OSF/1

The default pager applies the shadow-paging technique to every checkpointed process, however, only the checkpointed processes contain shadow maps. The default pager scans the page map and sets the ‘read-only’ bit of each valid entry when it receives a request from the microkernel. After all map entries have been set to ‘read-only’, the map is returned to the microkernel for use in recovery procedures. Figure 4 shows the detailed checkpointing steps for OSF/1.

The Mach kernel interface, `vm_msync()`, has been expanded, and two new options added to checkpoint dirty pages. The `vm_msync()` is designed to synchronize the memory region with its backing stores. When the new `VM_SYNC_FREEZE` option is set, the microkernel sends a freeze request to the default pager after all dirty pages have been flushed out.

The `VM_SYNC_PRECIOUS_OFF` option is used to turn the `PRECIOUS` option in the default pager off. The default pager uses the `PRECIOUS` option to reduce paging file utilization. Memory pages are either stored in the paging file or located in physical memory. When a memory page is read by the microkernel, it is deleted from the paging file. Although the `PRECIOUS` option reduces paging file utilization, it causes a great many disk accesses when applications generate large amounts of data that will not later be modified. The microkernel cannot determine whether these pages have been checkpointed or not. It must flush them and mark them ‘read-only’ again. Turning the `PRECIOUS` option off bypasses these pages.

When the microkernel sends a memory-object-freeze message, the default pager sets the ‘read-only’ bit in the pageout page map and returns the page map to the microkernel. The microkernel collects all the maps and then sends them to the checkpoint server.

OSF/1 also dynamically links user programs with shared libraries. The execution file contains unresolved external references prior to being loaded. Conventional restart mechanisms that use original execution files to reconstruct process images may not work in this environment. If shared libraries are upgraded by the system administrator or linkage paths are changed by the user, a restarted process will differ from the original execution image. To solve this problem, we take a complete snapshot of process images at the first checkpointing of any process. This approach separates the relationship between the applications and the shared libraries. The restart procedure only needs the incremental checkpoint file and the snapshot file to reconstruct the original process, even if the shared libraries have been changed.

PERFORMANCE EVALUATION

We ran our experiments on an Intel Pentium 90 MHz processor with 16 MB of RAM and a one GB SCSI disk. After the system was booted, the available physical memories consisted of 12.32 MB (3154 pages). The paging file used for applications was 256 MB. All the test applications made complete snapshots at the first checkpoint to generate basic recovery points. The checkpoint file size and disk accesses required for the first snapshot were not counted in the evaluations.

We first wanted to explore the extent of disk access when the application’s working set was larger than the available physical memory. The first application tested solves the *All-Pairs Shortest Paths* problem with 1408 nodes. The memory requirement for writable data, including stacks and bss for the shared libraries, was 35.18 MB (9008 pages). Total running time was 830 minutes. We used 10 minutes as our checkpoint interval. This interval is reasonable for a long-running application. Figure 5 shows the number of disk accesses in terms of pages accessed for different checkpoint policies. Because of the program’s total run time, a complete graph could not exhibit the detailed variations. We arbitrarily selected the results from 12,000 seconds to 19,000 seconds for inclusion in Figure 5. Although Figure 5

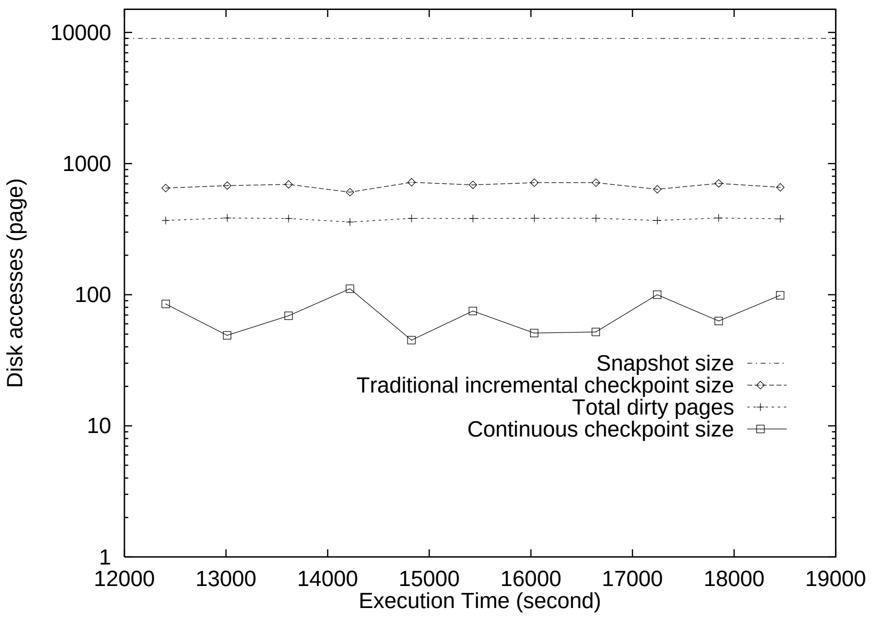

Figure 5. The total dirty pages and actual disk accesses for different checkpoint policies when solving All-pair Shortest Paths with run-times from 12,000 second to 19,000 second. The average number of disk accesses for conventional incremental checkpointing was 677 pages. When using continuous checkpointing, only 75 pages, on average, were accessed

shows only partial results, the data reported in the next paragraph were calculated for the whole program.

Dirty pages between checkpoints averaged 1.46 MB (376 pages). Because user-level incremental checkpointing must read dirty pages from the paging file and then save them to checkpoint files, the average number of disk accesses for conventional incremental checkpointing was 2.6 MB (677 pages). When the continuous checkpointing policy was used, only 300 Kbytes (75 pages) on average were actually written to disk. At least 80 percent of the dirty pages had already been saved to disk before the checkpoint times arrived.

The second test application we used was matrix multiplication of two 1600×1600 matrices. The total memory requirement was 29 MB (7424 pages), but only 10 MB (2560 pages) were actually necessary when calculating each row of results. Because other daemon processes coexisted during this test, the test program's working set was slightly larger than the available memory. The execution time was 135 minutes. The checkpoint interval was also set at 10 minutes. Figure 6 shows the total number of dirty pages and actual disk accesses at every checkpointing time (snapshots not included). Disk-access reductions ranged from 15 percent to 51.67 percent, with an average improvement in disk accessing of 35 percent.

Figure 7 presents the average number of disk accesses for multiplication of matrices of different sizes. When the matrix size was small, the entire working set could reside in physical memory, so, using continuous checkpointing was the same as user-level implementation of incremental checkpointing. However, continuous checkpointing did avoid the cost of page-

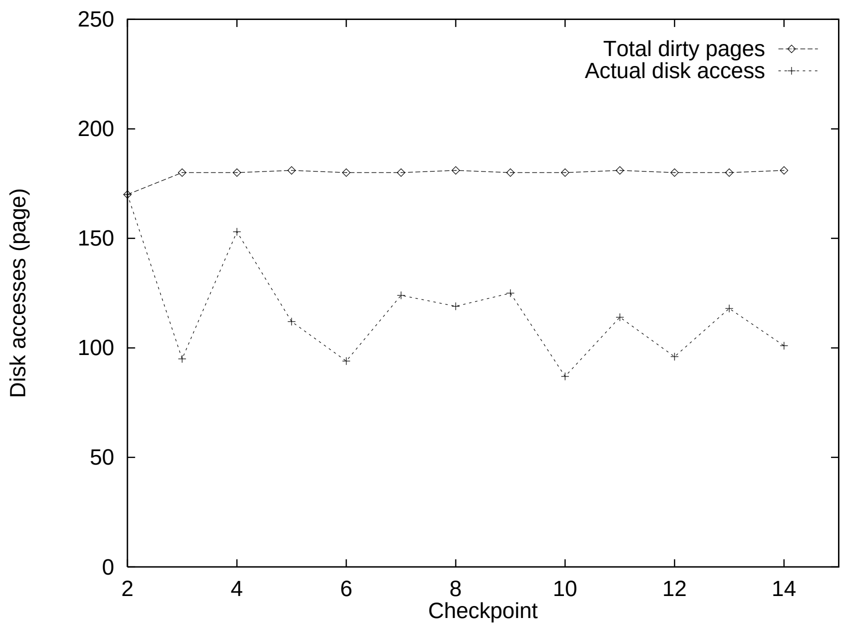

Figure 6. Total dirty pages and actual disk accesses in 1600 $\times$ 1600 *matrix multiplication*

fault handling. As the matrix size was increased, the total number of dirty pages decreased because some CPU time was spent in memory management. When the memory requirement was larger than the available physical memory, some of the dirty pages had been paged out before checkpointing times arrived. The actual number of disk accesses required for checkpointing was smaller than that required for incremental checkpointing. When the matrices were larger than 1792 $\times$ 1792, continuous checkpointing saved only two dirty pages to disk, and the process of checkpointing worked like a system pageout operation.

To compare the impact of continuous checkpointing with that of conventional checkpointing methods, we ported the libckpt library in Plank *et al*[6] to our platform. The libckpt library contains many conventional user-level checkpointing methods. We changed some system calls for memory allocation/protection and process creation to Mach system calls. When the incremental mechanism was active, the library applied the `mprotect` system call to detect which memory pages were dirty. When the incremental option was not set, then the checkpoint procedure took sequential snapshots of memory. When combined with the `fork` system call, the checkpoint operations could be performed in parallel with the process being checkpointed. Copy-on-write optimization is a built-in mechanism in Mach. When the parent process writes a page being checkpointed, Mach first copies this page to another place, and then gives write permission to the parent process.

The test program was a $B = A \times A$ matrix multiplication included in the libckpt library. The test program took checkpoints after every 40 rows of the product matrix had been calculated. Table I lists the results. The times reported in Table I are in minutes:seconds.

The average cost of handling single page fault was 1650 μs in our system. The ‘No

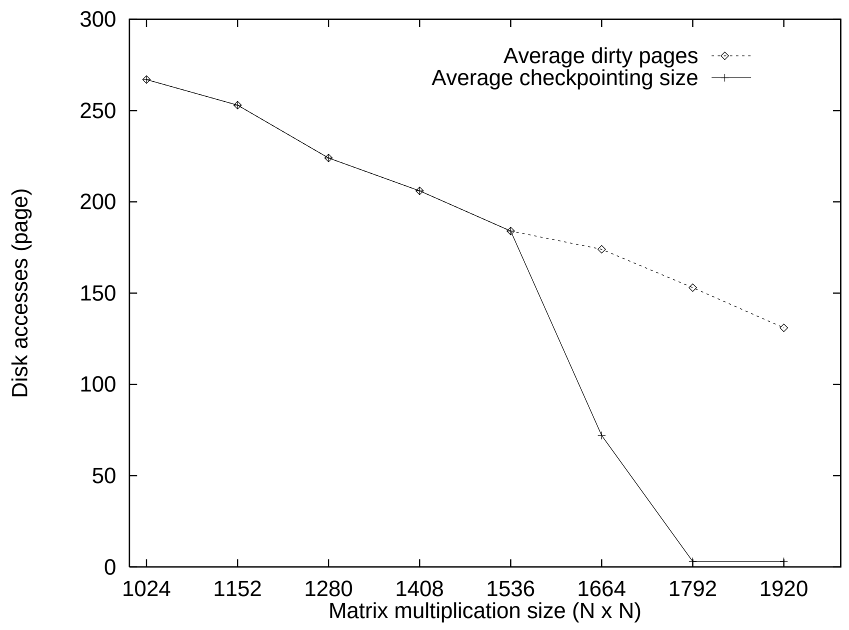

Figure 7. The average dirty pages and actual disk accesses at different size of matrix multiplication

checkpoint' column represents the elapsed time without checkpointing. When the matrix size was 1024 $\times$ 1024, all pages being checkpointed resided in physical memory. Thus, the I/O accesses required for incremental checkpointing and continuous checkpointing were the same. The only differences between these two methods were in page-fault handling and context-switching times. The performance of both incremental and continuous checkpointing improved when combined with copy-on-write optimization. When incremental checkpointing was combined with copy-on-write, the elapsed time was almost the same as that of continuous checkpointing without copy-on-write, but continuous checkpointing with copy-on-write was still better than incremental checkpointing with copy-on-write.

When the matrix size was 1664 $\times$ 1664, the total memory requirement was larger than the available physical memory. Both incremental and continuous checkpointing did not perform well when combined with copy-on-write optimization because the system was in heavy-load state. A `fork` or copy-on-write operation will bring more memory to swap out. Thus the disk

Table I. Elapsed time (minutes:seconds) for matrix multiplication using different checkpointing policies

Matrix size	No checkpoint	Sequential checkpoint	Incremental checkpoint	Continuous checkpoint	Sequential + Fork	Incremental + Fork	Continuous + Fork
1024 $\times$ 1024	23:04	48:30	27:01	25:23	29:05	25:54	23:40
1664 $\times$ 1664	136:43	316:49	178:39	143:08	235:47	254:34	149:28

access figure is worse than the checkpoint without copy-on-write optimization.

The sequential checkpoint size in the row '1024×1024' was 2560 pages (matrix A, B and 2 MB stack). The cost to write a page was roughly 22.9 ms ($(48 \times 60 + 30) - (23 \times 60 + 4) = 1526$ seconds for 26 checkpoints; $\frac{1526}{26} \div 2560 \approx 0.0229$). Combining continuous checkpointing with the `fork` system call, the copy-on-write optimization adds 1.65 ms to the cost of processing a protected page. This is 7 percent extra overhead per page fault, which is nontrivial, but not nearly as significant as the extra cost of disk I/O saved with continuous checkpointing when pages are already in the backing store.

OPTIMAL CHECKPOINTING INTERVAL ANALYSIS

Many studies have been published on selecting an optimal checkpointing interval. We used the simple model in Young[23] to analyze the impact of continuous checkpointing on the optimal checkpointing interval. We omit the detailed derivation of the equation, and apply only the final result here. This analysis does not include the impact of the copy-on-write technique. If a `fork` system call is used, the figures change, but the analysis is more difficult since the overhead realized when copy-on-write is performed cannot be stated analytically. More complicated models and assumptions can be found in Jalote[2].

Let *F* be the expected time to establish a checkpoint, and λ be the failure rate of the system. The Mean Time Between Failures (MTBF) of the system is $\frac{1}{\lambda}$. The first-order approximation of the optimal checkpoint interval is $T_{opt} = \sqrt{\frac{2F}{\lambda}}$.

We assume the expected checkpoint cost is the time required to save the memory pages and system states to file. Thus the data that need to be saved is $M_{resident} + M_{swap} + S$, where $M_{resident}$ is the average number of resident memory pages, M_{swap} is the average number of swapped memory pages and S is the number of pages that contain system states. Because conventional checkpoint methods need to read the swapped pages back and then save them to files, the expected time to establish a checkpoint using conventional methods is $F = (M_{resident} + 2M_{swap} + S)C$, where C is the average cost of storing a single page to disk. The optimal checkpointing interval can be expressed as $T_{opt} = \sqrt{\frac{2(M_{resident}+2M_{swap}+S)C}{\lambda}}$. In continuous checkpointing, M_{swap} is zero. Thus, the optimal checkpoint interval is reduced to $T_{opt} = \sqrt{\frac{2(M_{resident}+S)C}{\lambda}}$. The system avoids $2M_{swap}$ page's worth of disk operations. If all the pages to be checkpointed reside in physical memory, M_{swap} is zero. In this case, F is determined by the resident pages. Continuous checkpointing is thus reduced to conventional incremental checkpoint. Practically, how much benefit continuous checkpointing provides depends on the ratio of M_{swap} and $M_{resident}$.

Suppose the MTBF for our platform is 29.39 days (from Long *et al*[24]), the cost of storing a single page is 22.9 ms and the checkpoint size is 75 pages (from our first test program). The optimal checkpoint interval can be calculated as follows:

$$T_{opt} = \sqrt{(29.39 \times 24 \times 60 \times 60) \times 2 \times 75 \times 0.0229} \approx 2953 \text{ seconds.}$$

RELATED WORK

In this section we review research related to checkpointing the memory contents of running processes. Some of these techniques have been used in commercial products with good results. We first introduce the process-pair approach often used in distributed systems, then discuss methods that can be used for checkpointing on local disks, and techniques that can be used to decrease processing times.

Process-pair

The process-pair technique is often implemented in distributed operating systems, for example, NonStop OS,[25] Ausos[26] and TARGON/32.[27] For each fault-tolerant application, the operating system creates two processes at a time, the primary process running on the local machine, and a backup process located in the remote machine. The backup process can be executed in parallel with the primary process or be kept in a suspended state. Operating systems guarantee the input/output messages of primary and backup processes are sent atomically. The memory contents of backup process can be synchronized with those of the primary process page-by-page. A page server manages the virtual memory backing store to ensure that the backup process has the same images as the primary process. Applications need not be re-compiled or re-linked with special libraries in these systems. A fault-tolerant system that uses the process-pair approach often has high availability characteristics. A fault-tolerant service can be easily accomplished using this method.

Because microkernel technology is becoming the trend in next-generation's operating systems, some research has been conducted on applying process-pair techniques to microkernel-based platforms. Babaoğlu,[28] and Chen and Ng[29] proposed fault-tolerant computing environments based on Mach.[17] They designed an `external pager` that finds checkpointed dirty pages and sends them to another machine. The external pager applies the virtual memory protection mechanism proposed by Li *et al.*[10] to asynchronously checkpoint dirty pages.

Since every primary process keeps a hot backup in another machine, recovery processing is accomplished almost immediately after system crashes. However, systems using process-pair techniques pay a higher cost in equipment than other methods. At least one backup machine is necessary whenever the primary process is running. The operating system also needs to modify most of its codes to satisfy the atomic message requirement.

Taking a snapshot

To checkpoint a process in a local machine, a brute force method is to take a snapshot of its memory. Because this method sequentially saves the memory contents to disk, Li[10] classified it as sequential checkpointing. Since this approach is simple, most user-level packages or libraries use it. For example, the Condor[5] library adopted this method to utilize idle workstations. Programs using the Condor library need not modify their source codes. However, relinking with the Condor Checkpointing Library is required. When checkpointing, the 'checkpoint' signal handler in the Condor checkpoint library first stops the original execution sequence and then saves the data area and stack area to file. At restart time, the restarting process (using code from the Condor library) reads the file containing the saved data and stack information into appropriate areas in its own address space. The latest release of Condor has been ported to Solaris, Irix, Linux and supports dynamically linked libraries as part of the checkpoint. Further information about Condor can be found on Condor's homepage, `http://www.cs.wisc.edu/condor/product.html.`

Incremental checkpointing

Another approach, incremental checkpointing, which saves only the pages modified between two checkpoint times, was first used in program debugging[13]. By means of the system call, `pagemod()`, user processes could checkpoint dirty pages without using page-fault handlers. Elnozahy et al.[9] used incremental checkpointing technique to implement their consistent checkpointing in Manetho. They showed that incremental checkpointing can efficiently reduce the checkpoint size. Plank et al.[6] used an `mprotect` system call to implement incremental checkpointing. However, providing incremental checkpointing at user-level entails a higher cost in page-fault handling.

Russinovich and Segall[12] designed an application-transparent checkpointing algorithm using the Mach operating system. We classify their algorithm as incremental checkpointing because they designed an external pager to checkpoint the memory contents. Using the external pager approach in Mach enables easy retrieval of information about modified data.

All the Mach-based approaches[12,28,29] involved designing a new external pager to checkpoint memory contents, which differs from our approach. We modify the microkernel's default external pager to support incremental checkpointing because:

(i) The default pager is a privileged task with the ability to obtain additional memory resources when the system in heavy-load states.
(ii) The default pager is the last resolver of paging. It also handles the memory pages created by the microkernel. Other external pagers cannot get these dirty pages.
(iii) The default pager directly uses the device interface provided by Mach. It does not rely on file system service from the UNIX server.

Virtual checkpointing

Bowen and Pradhan[7,8] proposed a technique for checkpointing memory contents in nonvolatile-memory supported environments. In their scheme, each virtual page has two mappings both in the physical memory and disk storage. One is the checkpoint data set and the other is the active data set. A global checkpoint counter (V) is used as the logical timestamp for the current checkpoint. Every virtual page has a local checkpoint counter (v) to indicate the last checkpoint timestamp. The operation at checkpoint time only increments the global checkpoint counter V. The actual checkpoint action is deferred until the first reference to a checkpointed page. In every memory reference, the local checkpoint counter v is compared with global checkpoint counter V. If the local checkpoint counter v is not equal to the global checkpoint counter V, an actual checkpoint procedure will be invoked — the active data set and checkpoint data set are switched and the global checkpoint counter V is copied to the local checkpoint counter v.

The virtual checkpoint approach needs nonvolatile-memory support. Most functions in virtual memory systems must be modified to implement the deferred checkpoint procedure. Rapidly comparing the local checkpoint counter v with the global checkpoint counter V at every memory reference, requires definition of additional bits in the Translation Lookaside Buffer (TLB) and page table entry. The continuous checkpointing we propose on the other hand, only needs two disk mapping per checkpointed process. The physical memory mapping is the same as usual and does not require nonvolatile-memory support. This simplifies the design of checkpointing facilities in most systems.

KeyKOS

KeyKOS[14,15] is an object-oriented microkernel operating system from Key Logic Inc. The checkpoint mechanism in KeyKOS is system-wide. This means that the system checkpoints all the processes' states at one time. The checkpoint facility in KeyKOS is integrated into the paging system. Every memory page is assigned a *home location* on disk. There are two swap areas on disk. One is the *checkpoint area* and the other is the *working area*. A small area on disk called the *checkpoint header* designates which swap area is the current checkpoint area.

When checkpointing, all the processes in the system are stopped. After the system writes all the changed pages to the working area, the roles of the swap areas are reversed, and the checkpoint header is updated to designate the new checkpoint area. When all the stopped processes are resumed, a background process performs a *migration* phase that copies the pages in the checkpoint area to their home locations.

Because the checkpoint header needs to differentiate between the checkpoint area and the working area, KeyKOS needs the additional migration phase to re-use the swap area. This is the major difference between the checkpointing mechanism in KeyKOS and our approach. If the last checkpointed pages are the same as the current in-use pages, they will be copied to their home location during the *migration* phase. A memory page in the backing store has at most three copies. The migration phase is a built-in low-priority user-level process that does not influence the execution of other applications.

Other approaches

Overlapping checkpointing and process execution may be accomplished by using the `fork` system call. After the original process forks a child, the parent process resumes the original execution flow and the child process saves its own memory space to the checkpoint file. This technique has been used by several authors[6,30,31]. Liu et al.[32] used a similar method to checkpoint multi-threaded applications. They created checkpoint threads in applications to obtain access rights to memory space.

Li et al.[10,11] proposed real-time and concurrent algorithms that overlap application execution and checkpointing. They used copy-on-write and buffering techniques to parallelize the memory saving and program execution. These techniques can improve the checkpoint snapshot time.

Plank et al.[6] implemented a user library `libckpt.a` that provides multiple checkpointing policies. They also proposed a user-directed checkpointing method that allows users to specify which memory can be excluded from checkpointing, and which the server should checkpoint. This method is not user-transparent. However, compiler-based techniques[33,34,35] can ease the task of user-directed checkpointing.

In the commercial sector, both UNICOS,[4] Cray Research's operating system, and ConvexOS,[3] CONVEX Computer Corporation's operating system, provide the system calls `chkpnt` and `restart` to automatically checkpoint and recover user processes. These facilities are transparent to the checkpointed processes. No modification is necessary for user application.

CONCLUSION

Fault tolerance will be more and more important in future computing systems. Improving system reliability while not decreasing throughput is an important design issue. Although there are many user-level checkpoint packages, kernel-level checkpointing mechanisms are still superior.

Kernel-level and user-level implementations of checkpointing mechanisms both have their pros and cons. When implemented in the kernel-level, the checkpointing mechanism can obtain more system resources and information making checkpointing more efficient. Since most users cannot modify or install new operating systems to checkpoint their jobs, a user-level checkpoint package or library becomes a feasible solution for them. Portability is the greatest advantage of these packages. Their main disadvantage is that they cannot rebuild the state of the operating system directly. In particular, user-level checkpointers cannot identify pages that have been swapped to disk, and therefore must copy these pages to their checkpoint files. As a kernel-based technique, continuous checkpointing can merge checkpointing with swapping pages to disk, and gain significant performance advantages over user-level checkpointers. Thus we emphasize that checkpointing of memory contents should be a kernel-level mechanism and can be integrated into memory paging.

Our results have shown that the proposed continuous checkpointing can efficiently checkpoint process memory contents. For a checkpointed process, especially one requiring extensive memory resources, most disk accesses are eliminated. Even if all dirty pages are resident in physical memory, our approach pays a lower cost to find checkpointed pages than conventional checkpointing methods.

ACKNOWLEDGEMENTS

This project was partially supported by the National Science Council of the Republic of China under grant No. NSC85-2213-E009-079.

REFERENCES

1. P. A. Lee and T. Anderson, *Fault Tolerance – Principles and Practice*, Springer-Verlag, Wien, New York, 1990.
2. P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice-Hall, 1994.
3. Convex Computer Corporation, *ConvexOS Extensions User's Guide*, DSW-053, CONVEX Press, December 1991.
4. B. A. Kingsbury and J. T. Kline, 'Job and process recovery in a UNIX-based operating system', *Proc. of the Winter 1989 USENIX Conference*, January, 1989, pp 355–364.
5. T. Tannenbnum and M. Litzkow, 'The Condor distributed processing system – checkpoint and migration of UNIX processes', *Dr. Dobb's Journal*, February 1995, pp 40–48.
6. J. S. Plank, M. Beck, G. Kingsley and K. Li, 'Libckpt: transparent checkpointing under UNIX', *Proc. of the Winter 1995 USENIX Conference*, January 1995, pp 213–224.
7. N. S. Bowen and D. K. Pradhan, 'A virtual memory translation mechanism to support checkpoint and recovery', *Proc. of Supercomputing '91*, November 1991, pp 890–899.
8. N. S. Bowen and D. K. Pradhan, 'Virtual checkpoints: architecture and performance', *IEEE Trans. on Computers*, **41**(5), 516–525 (1992).
9. E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel, 'The performance of consistent checkpointing', *Proc. of 11th Symp. on Reliable Distributed Systems*, 1992, pp 39–47.
10. K. Li, J. F. Naughton and J. S. Plank, 'Real-time, concurrent checkpoint for parallel programs', *Proc. of the 1990 Conference on the Principles and Practice of Parallel Programming*, March 1990, pp 79–88.
11. K. Li, J. F. Naughton and J. S. Plank, 'Low-latency, concurrent checkpointing for parallel programs', *IEEE Trans. on Parallel and Distributed Systems*, **5**(8), 874–879 (1994).
12. M. Russinovich and Z. Segall, 'Application-transparent checkpointing in Mach 3.0/UX', *Proc. of the 28th Hawaii International Conference on System Science*, January 1995.
13. S. I. Feldman and C. B. Brown, 'Igor: a system for program debugging via reversible execution', *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, **24**(1), 112–123 (1989).
14. C. R. Landau, 'The checkpoint mechanism in KeyKOS', *Proc. of the Second International Workshop on Object Orientation in Operating Systems*, September 1992.
15. A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau and J. S. Shapiro, 'The KeyKOS nanokernel architecture', *Proc. of Micro-kernels and Other Kernel Architectures*, 1992, pp 95–112.

16. R. A. Lorie, 'Physical integrity in a large segmented database', *ACM Trans. on Database Systems*, **2**(1), 91–104 (1977).
17. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, 'Mach: a new kernel foundation for UNIX development', *Proc. of the Summer 1986 USENIX Conference*, July 1986, pp 93–113.
18. D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan and D. Bohman, 'Microkernel operating system architecture and Mach', *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp 11–30.
19. J. Boykin, D. Kirschen, A. Langerman and S. LoVerso, *Programming under Mach*, Addison-Wesley, 1993.
20. D. Golub, R. Dean, A. Forin and R. Rashid, 'Unix as an application program', *Proc. of the Summer 1990 USENIX Conference*, June 1990, pp 87–96.
21. K. Loepere, *OSF Mach MK5.0 Kernel Principles*, Open Software Foundation, Inc. and Carnegie Mellon University, 1993.
22. Open Software Foundation, *Design of the OSF/1 Operating System : Release 1.2*, Prentice-Hall, 1993.
23. J. W. Young, 'A first order approximation to the optimum checkpoint interval', *Communications of the ACM*, **17**(9), 530–531 (1974).
24. D. Long, A. Muir and R. Golding, 'A longitudinal survey of internet host reliability', *Proc. of 14th Symposium on Reliable Distributed Systems*, September 1995 pp. 2–9.
25. J. F. Bartlett, 'A nonstop kernel', *Proc. of the Eighth Symp. on Operating Systems Principles*, December 1981, pp 22–29.
26. A. Borg, J. Baumbach and S. Glazer, 'A message system supporting fault tolerance', *Proc. of the Ninth ACM Symp. on Operating Systems Principles*, October 1983, pp. 90–99.
27. A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, 'Fault tolerance Under UNIX', *ACM Trans. on Computer Systems*, **7**(1), 1–24 (1989).
28. Ö. Babaoğlu, 'Fault-tolerant computing based on Mach', *Proc. of 1990 Mach Workshop*, October 1990, pp 185–199.
29. R. Chen and T. P. Ng, 'Building a fault-tolerant system based on Mach', *Proc. of 1990 Mach Workshop*, October 1990, pp 157–168.
30. A. Clematis, G. Dodero and V. Gianuzzi, 'Process checkpointing primitives for fault tolerance: definitions and examples', *Microprocessors and Microsystems*, **16**(1), 15–23 (1992).
31. D. J. Taylor and M. L. Wright, 'Backward error recovery in a UNIX environment', *Proc. of the 16th International Symp. on Fault-Tolerant Computing*, July 1986, pp 118–123.
32. Y. H. Liu, L. M. Ni and C. F. E. Wu, 'Application data checkpointing for multi-threaded UNIX systems', *Proc. of International Computer Symp.*, December 1994, pp 173–178.
33. C. J. Li and W. K. Fuchs, 'CATCH – Compiler-assisted techniques for checkpointing', *Proc. of the 20th International Symp. on Fault-Tolerant Computing*, June 1990, pp 74–81.
34. J. Long, W. K. Fuchs and J. A. Abraham, 'Compiler-assisted static checkpoint insertion', *Proc. of the 22th International Symp. on Fault-Tolerant Computing*, July 1992, pp 58–65.
35. C. J. Li, E. M. Stewart and W. K. Fuchs, 'Compiler-assisted full checkpointing', *Software—Practice and Experience*, **24**(10), 871–886 (1994).