SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(11), 985–1004 (1999)

An Implementation of Using Remote Memory to Checkpoint Processes

SHANG-TE HSU AND RUEI-CHUAN CHANG*
Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC
(email: gis79535@os.nctu.edu.tw, rc@cc.nctu.edu.tw)

SUMMARY

Process checkpointing is a procedure which periodically saves the process states into stable storage. Most checkpointing facilities select hard disks for archiving. However, the disk seek time is limited by the speed of the read-write heads, thus checkpointing process into a local disk requires extensive disk bandwidth. In this paper, we propose an approach that exploits the memory on idle workstations as a faster storage for checkpointing. In our scheme, autonomous machines which submit jobs to the computation server offer their physical memory to the server for job checkpointing. Eight applications are used to measure the remote memory performance in four checkpointing policies. Experimental results show that remote memory reduces at least 34.5 per cent of the overhead for sequential checkpointing and 32.1 per cent for incremental checkpointing. Additionally, to checkpoint a running process into a remote memory requires only 60 per cent of the local disk checkpoint latency time. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: fault tolerance; remote memory; checkpoint

INTRODUCTION

To provide fault-tolerant services, a computation server needs to periodically checkpoint users' processes. If a failure occurs, the interrupted process can be restarted from the last checkpointed state [1]. Since the hard disk is a generally stable medium, most checkpointing facilities use a local disk to store process states, but disk access time has not improved as quickly as other computer components; therefore, the checkpointing performance is bounded by the disk access time. How to reduce or avoid disk access delay has become an important issue in the design of checkpointing.

There have been many approaches to reduce the checkpoint overhead [2,3]. *Incremental checkpointing* saves the modified data only. *Checkpoint buffering* overlaps the I/O operation in checkpointing with program execution; and the *copy-on-write* technique avoids unnecessary data copy. All of these techniques have successfully improved checkpointing performance. Another area of research involves avoiding disk accesses during checkpointing. The major principle is to use memory or processor redundancy to store checkpoints [4–8]. Checkpoints are either stored on a backup host or are parity checkpointed to another host. In this paper,

*Correspondence to: Ruei-Chuan Chang, Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC. Also with Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan, ROC.

CCC 0038–0644/99/110985–20$17.50
Copyright © 1999 John Wiley & Sons, Ltd.

Received 4 August 1998
Revised 14 April 1999
Accepted 21 May 1999

we propose using *remote memory* as a virtual disk for the checkpoint archives, which not only avoids disk access during checkpointing, but also improves the utilization of computing resources in the network.

A *remote memory* is a collection of memory resources located on other machines. As the bandwidth in various communication mediums rises (from 10/100 Mb to Gigabit Ethernet), accessing a memory located on another machine is no slower than the local disk. This *remote memory* becomes another kind of shared resource in today's computing environment. Anderson *et al.* [9] commented that fast network communication allows the aggregate DRAM of a Network Of Workstations (NOWs) to perform as a giant cache for a disk. Many previous studies [10,11] have been reported in which efficient use of the resources of other machines can improve the total file system performance throughput. The other studies have been proposed using remote memory, instead of the local disk, as paging storages [12,13].

Furthermore, remote memory has many characteristics that are different than disks. First, there is no seek time in a read/write access. Accessing any remote memory address has the same latency time. Secondly, although each machine's CPU speed may be different, the memory access time does not differ much. Therefore, using remote memory as a virtual disk can improve checkpoint performance.

Although the reliability and security of a remote memory is not necessarily better than the local disk, many technologies on RAID and data encryption can make up for these native defects. In addition, we periodically backup checkpoint files from the remote memory to the local disk, which means a failed process can be restarted even if the communication link is broken at the same time.

To verify this proposal, we implemented remote memory process checkpointing on OSF/1 version 1.3 [14] and evaluated the impact on sequential, incremental and copy-on-write checkpointing. 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 [15] 3.0 kernel developed by Carnegie Mellon University. Although we used a microkernel-based operating system as our development environment, the principles of our model can also be applied to conventional monolithic operating systems.

CHECKPOINT SCHEME

There are four methods usually used to checkpoint process states: sequential, incremental, copy-on-write (or forked) checkpointing and checkpoint buffering. These techniques do not exclude one another in use. In practice, the copy-on-write or checkpoint buffering technique is often combined with sequential and incremental checkpointing to acquire better performance. To evaluate these checkpointing schemes, there are two criteria–checkpoint *overhead* and checkpoint *latency* [16]. Checkpoint overhead is the increase in execution time of an application due to the checkpoint process. Checkpoint latency is the duration of time required to generate a checkpoint. In performance evaluation, the reduction of checkpoint overhead is more important than the checkpoint latency time.

(a) *sequential checkpointing*–to checkpoint a process, the simplest and most straightforward method is to stop the running process, save the entire process state to files, and then resume the checkpointed process. Since the process is suspended during checkpointing, the checkpoint overhead is identical to checkpoint latency.
(b) *incremental checkpointing*–it is not necessary to save the unmodified data in every checkpoint. Only the modified data during two continuous checkpoint times is saved

in this method. In a practical implementation, this method applies the virtual memory primitives to identify the modified data between checkpointing times. Incremental checkpointing reduces the data that will be checkpointed. Thus, the checkpoint latency time, in general, is shorter than that of sequential checkpointing. However, the handling of modified pages will increase the checkpoint overhead.

(c) *checkpoint buffering*–checkpoint buffering is an approach that overlaps the execution of the checkpointed process and the I/O operation in checkpointing. The basic idea is to use a memory buffer to make a copy of the process states, and then execute in parallel the process and checkpointing. The process being checkpointed is only suspended at the memory copy phase. The technique used in this approach can be described as 'overlap the I/O operation with CPU execution.'

(d) *copy-on-write checkpointing*–although checkpoint buffering is a good approach to parallelize the checkpointing with the application execution, the memory space is not always enough for buffering. When the amount of memory is not enough for buffering, the additional memory paging will increase the checkpoint overhead. Most implementations use copy-on-write checkpointing instead of checkpoint buffering. The principle of this approach is that, most of the time, the process being checkpointed will not write the pages which have not yet been checkpointed. Thus, the application execution and checkpointing can share the same memory space, which is protected by the operating system. Only the pages which cause a memory protection fault will be copied first. The technique used in this approach can be described as 'avoid unnecessary data copy.' The checkpoint latency time in checkpoint buffering and copy-on-write checkpointing is often larger than other methods, but the checkpoint overhead can be reduced because the CPU can perform useful computations during the I/O operation.

Because the performance of sequential and incremental checkpointing is determined by the speed of storage, we focused on using remote memory in these two checkpointing methods. In addition, we examined the effect when sequential and incremental checkpointing are combined with the copy-on-write technique.

The remote memory applied to sequential and incremental checkpointing is shown in Figure 1. In sequential checkpointing, remote memory replaces the role of the local disk in checkpointing. The checkpointer saves the process' states to a remote memory instead of the disk. Figure 1 presents a practical case of sequential checkpointing when at most two checkpoint files exist in storage. One is the latest checkpoint that has been made, and the other is the checkpoint being generated. When the new checkpoint has been stored completely, the current checkpoint can be removed and the new checkpoint becomes the current checkpoint. The incremental checkpointing approach must first generate a checkpoint base. This can be created from a complete snapshot of the image, or be directly based on the original execution file. After that, the checkpointer performs an incremental checkpoint at each checkpoint time. For the sake of reliability, the system can also use a local disk to save the checkpoint from the remote memory.

Security considerations

One may argue that a checkpoint residing in another machine's memory will be susceptible to security violations, allowing an unauthorized user to modify the process states. The system designer can use the following approaches to solve this security problem:

(a) apply authentication protocols between computation server and idle machines;

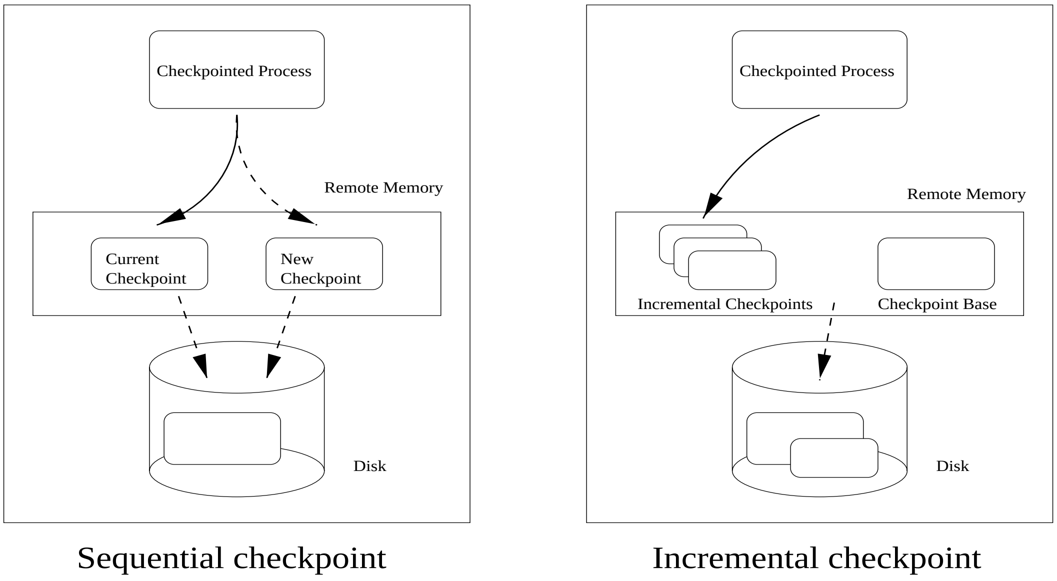


Figure 1. Remote memory used in different checkpoint approaches

(b) encrypt and decrypt the data during transmission;
(c) implement the memory server into the kernel of idle workstations so that no-one can access the contents.

REMOTE MEMORY SERVICE

The proposed checkpoint scheme relies on a reliable remote memory service to prevent data loss. In this section, we present an overview of the remote memory concept and the techniques used to create a reliable remote memory.

Remote memory model

Remote memory refers to the computing model in which a machine uses the memory of other machines as another faster-than-disk storage medium. The remote memory providers can be the dedicated machines that provide their memory as the primary service, the autonomous machines that provide their memory at idle time and revoke the memory when they become active, or a hybrid of these two types. Figure 2 presents these remote memory models. An example of a dedicated remote memory server is the remote memory model by Comer and Griffioen [12]. A memory server uses only its own physical memory and local disk. When the physical memory is exhausted, portions of the memory contents will be stored to disks for memory paging. The Global Memory Service (GMS) is a distributed remote memory system [17]. Every machine is an element of the remote memory service. The model by Narten and Yavatkar [18] is a hybrid remote memory system. The remote memory server can use other machines' memory when they become idle. The remote memory model in this paper is a cooperative model that consists of a central computing server and many thin client machines. This model is different from the distributed remote memory model, in that the remote memory provider provides only the access mechanism. The remote memory is

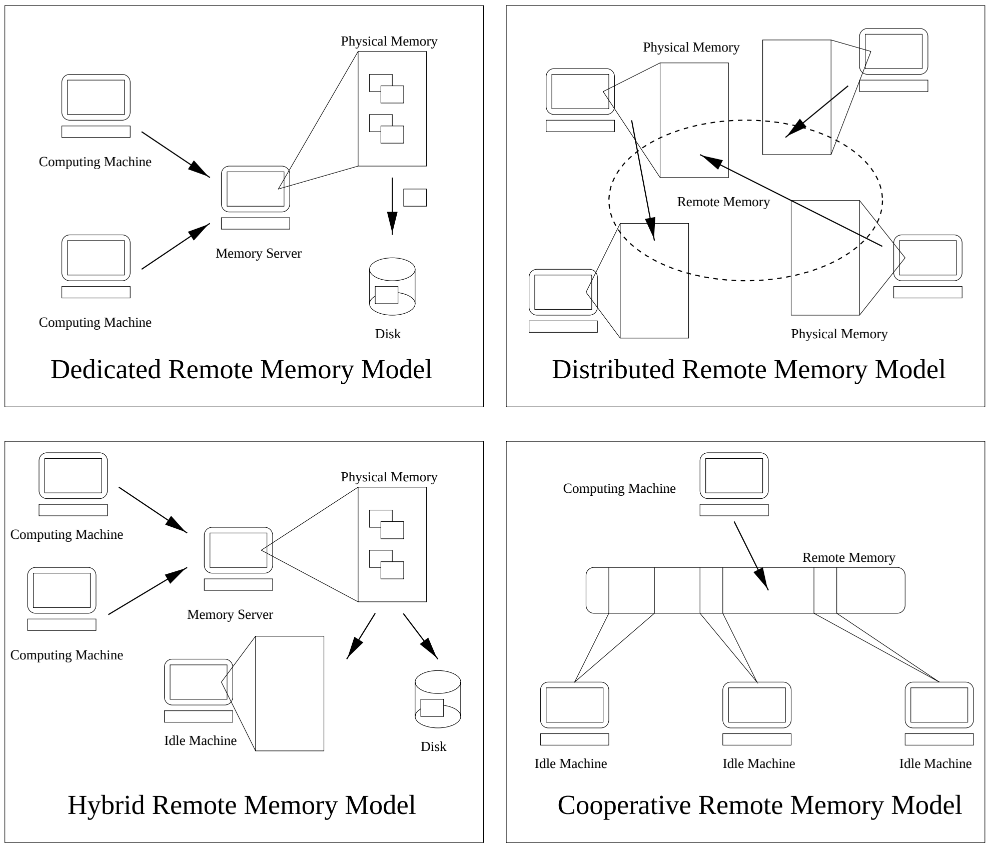


Figure 2. Dedicated, distributed, hybrid and our remote memory models

directly controlled and managed by the computing machine. In a practical environment, the remote memory provider may not be as powerful as the computing machine; it can be a PC or some kind of thin client machine. It dynamically adds and revokes its memory resource to the remote memory system.

The cooperative model consists of all of the workstations in a local area network. Some of the machines are computing servers. Other machines, which do not have much computing server capability, are autonomous workstations. All of the machines are connected using a fast communication medium. When any one of these autonomous machines is left idle for a long time, it becomes a memory server. However, the memory resource can be revoked when a user activates the machine.

We summarize the characteristics of above the four models as follows:

(a) The dedicated and hybrid remote memory model has one or more machines performing the centralized service control.
(b) The distributed remote memory model does not have a central manager to serve every machine. Its management is distributed.
(c) In a cooperative remote memory model, remote machines only provide an access mechanism. The usage and management of remote memory is managed by the computing machine.

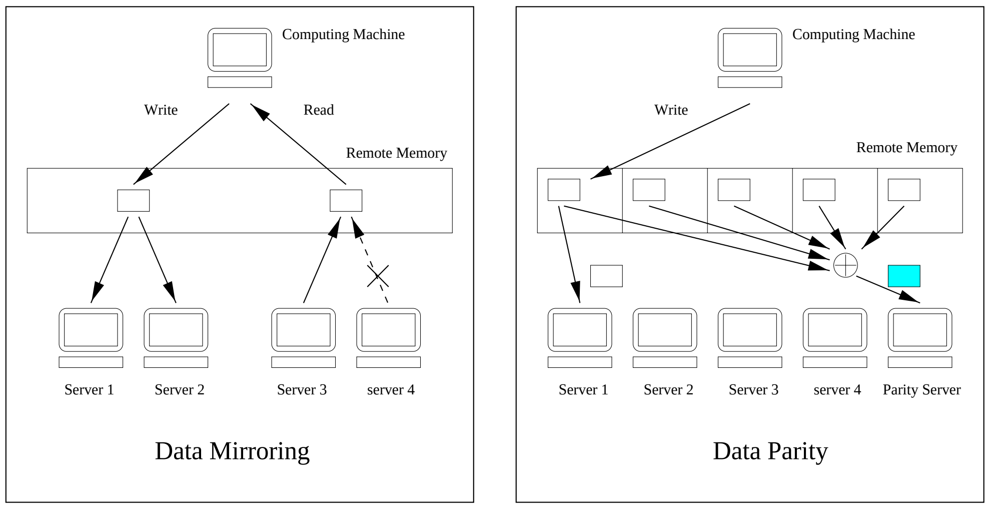


Figure 3. Data mirroring and parity techniques used in remote memory service

Remote memory can be used for file caching, memory paging or temporary storage in program compiling. Most of the previous research used remote memory to improve disk access read performance. When files are written, they are written to both the remote memory and disk. When the local system performs a disk read operation, it first checks the contents of the remote memory. If the remote memory contains the data that the local system needs, that data are read from the remote memory instead of the local disk. In this case, the remote memory acts as a write-through disk cache for this machine. The data located on the remote memory is clean and consistent with that on the local disk.

The main difference between our work and the previous work is that the remote memory is used as a write-back disk cache, thus writing reliability and performance is the major concern in this design.

Reliability consideration

The most important function of checkpointing is to ensure that interrupted processes can be recovered. Thus, the reliability of the remote memory is the most important issue in this design. Many studies [6,7,19] have addressed this issue using RAID techniques [20] in the remote memory. The basic idea is to use data replication or parity to ensure reliability. Figure 3 presents these two techniques used in remote memory service.

The RAID-1 (data mirroring) technique replicates data to multiple hosts. When the remote memory system adopts the RAID-1 technique, every remote memory write request is sent to at least two remote hosts. If any remote machine fails and there is still a replica, the local machine can remain functional.

When the remote memory system adopts the RAID-5 (data parity) technique, the remote host is configured to *n* data hosts and one parity host. The local machine generates a parity block for each remote memory write. If any one of the data hosts fails, the failed block is re-calculated from the un-failed host and the parity host.

Memory revocation

The remote memory model requires a fair policy to reclaim the exported memory resource. This issue is the same as the Condor [21] and Sprite [22] operating system, which use idle machines to perform computations. Of course, the owner of the idle machine will have the highest priority for using his/her own machine. Thus, when a machine is activated, the machine must respond immediately. There are two considerations in the design of remote memory systems: first, to meet acceptable response time when the user reclaims the machine, it is better not to acquire all of the free memory to memory service; secondly, when a machine becomes active, it is not necessary to revoke the exported memory resource immediately. It is possible that the end user is using the machine temporarily, to read email and then leave, for example. Revoking the exported memory immediately may induce system overhead. In Condor, the process migration is triggered after the idle machine becomes active for at least five minutes.

A memory revocation from any autonomous machine is similar to the case when a remote machine fails. The computation server cannot access the memory resource of that machine any more. To gracefully reduce the impact from this case, the remote memory system must provide a method to solve the memory revocation. Felten and Zahorjan [23] have proposed four policies when a remote memory is revoked:

(a) If there is a replication of memory contents, then this machine can directly close service.
(b) If there is no replication, then another idle machine is chosen and the server is instructed to migrate the memory contents.
(c) Transfer to the memory server's disk.
(d) Transfer to the computation machine's disk.

In our model, the memory contents on the remote memory servers are important in process recovery. Thus, the third approach listed above was not implemented in our system. If there is no idle machine for data migration, the memory contents will be read back and saved to the local disk.

IMPLEMENTATION

We have implemented a remote memory model on OSF/1 1.3 and tested the checkpoint performance. OSF/1 1.3 is a microkernel based on Mach 3.0. All the functionality of UNIX (including the file system) is provided by a server at the user-level. Although the platform is a microkernel-based operating system, most of the code we modified is in the device driver and the virtual memory management unit. The remote memory and checkpoint concept can probably be implemented more easily on other operating systems.

The architecture is shown in Figure 4. The remote memory communication link uses a 100 Mb Fast Ethernet. Each Ethernet packet consists of a 64-byte header and variable length data. The first 14 bytes of the header are the destination/source address and packet type. The following 50 bytes contain the information used in the remote memory access, i.e. the accessing address, size and sequence number. The data consistency is maintained by sequential number, and only in-order packets are accepted. We believe this is not a problem in a local area network, since out-of-order packets are rare.

For the sake of performance and security, the memory server is implemented in the kernel. There is a thread in the kernel that is activated when the system becomes idle. This thread first checks the number of free pages and then allocates adequate memory pages to be used in the remote memory service. It then modifies the high-water and low-water mark of the free page

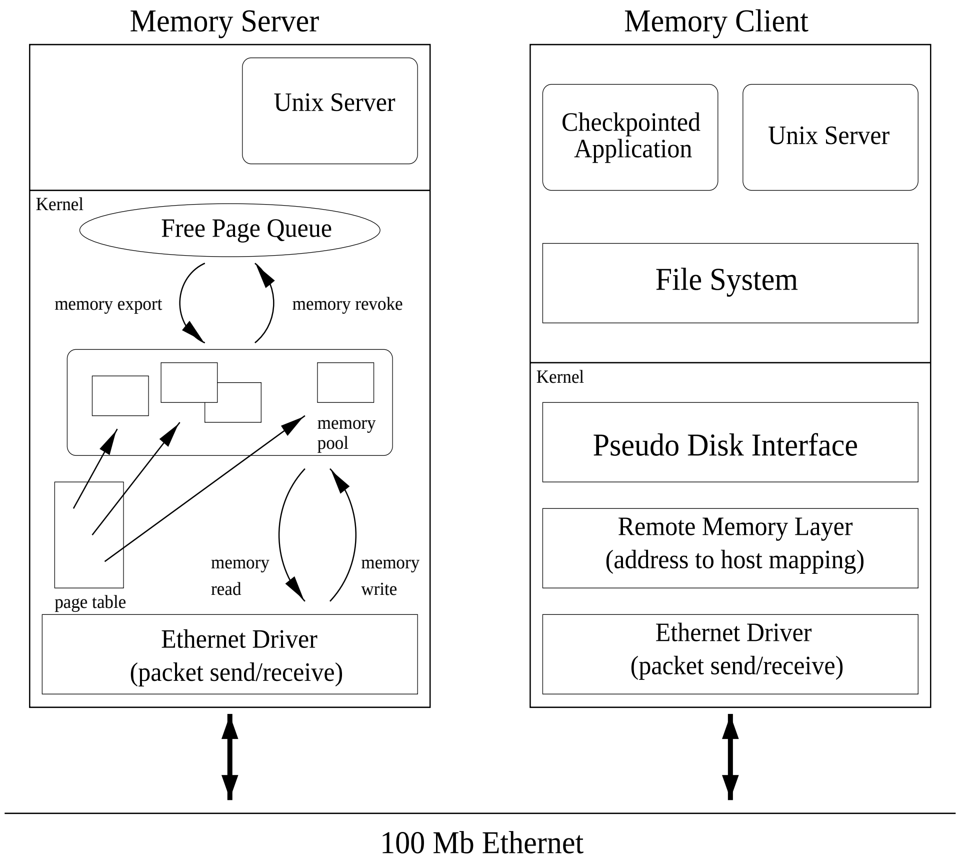


Figure 4. The remote memory model architecture

queue to ensure that the LRU mechanism of virtual memory is still working. Note that there is no governing principle in deciding how many memory pages are adequate in the remote memory service. In the default policy, we begin by allocating half of the free pages. However, as described in the next section, we have adjusted the allocation number to meet the actual requirement. All the allocated memory pages are direct-mapped from the end of kernel virtual space to ensure each memory access is served within a bounded time.

In memory clients, the kernel provides a pseudo-disk interface to the application. The advantage of this approach is that the Unix server in the user-level does not need modification. In addition, it has no relationship with remote memory. The middle layer implements the mapping from disk block to remote memory. This layer also implements the data replication and reserves several backup servers in the event that the user wishes to revoke the memory. The bottom layer is the same Ethernet driver as in the memory server. For the purpose of data replication, this driver uses multicast to transmit packets, therefore not causing any additional loading on the network.

Using a layered approach, the design and implementation of the checkpointing facility becomes much easier, and requires less modification. Only a background daemon periodically copies the checkpoint file from remote memory to a local disk to cover cases of remote memory failure during process recovery.

In practice, the size of the remote memory varies from time to time. When the available remote memory is not enough to hold all the checkpoint data, the computation machine must use local disk checkpointing. A feasible approach is to translate remote memory access to the local disk in the remote memory layer of the computation server. When the size of the remote memory is lower than a threshold, the remote memory layer reads all of the data back and saves it to a disk. Further remote memory accesses will be translated to the local disk. Once the size of the remote memory is larger than the threshold, the computation server then writes of the saved data to the remote memory and reuses it. Thus, a change in remote memory servers will not influence the pseudo-disk layer, file system and checkpointing packages.

The address translation and remote host is recorded in a table of memory clients. Because the remote memory is built on a local area network, the maximum size of the remote memory pool is pre-defined. When a system is recovered, it will first check whether the memory servers are still in-service, then try to remap all of the remote memory addresses. The file system then checks the file consistency. When these tasks are completed, the interrupted processes will be restarted from the remote memory. If these procedures cannot be completed, the interrupted processes are restarted from the checkpointed states in the local disk, which represents an earlier state than the checkpoint in the remote memory.

PERFORMANCE EVALUATION

The experimental environment is based on four PC's linked by a 100 Mb/s Fast Ethernet. Each machine has a 133 MHz Intel Pentium processor, 128 MB RAM and a local disk attached to an Adaptec 1542B SCSI card. The SCSI card can support up to 5 MB per second synchronization transfer rate. One of the machines was the computation server in the experiments. The other three provided 100 MB free memory each in remote memory service. This memory was configured to a total of 300 MB memory or 100 MB memory with one replica host and one backup host.

Micro benchmark

We first measured the raw bandwidth of the Fast Ethernet from the device driver layer. The client sends a special packet to one of the servers repeatedly until 100 MB of data is transferred. The server sends acknowledgment packets only at the beginning and end of the transmission. The elapsed time of this test was 8.9 seconds. From this test, we determined that the driver layer can support a bandwidth up to 11.2 MB per second, which is 89.6 per cent of the maximum bandwidth of the link.

To measure the bandwidth from the remote memory layer, we wrote a system call that repeatedly accessed the remote memory. The remote memory was aggregated to form a 300 MB memory pool. Due to the DMA boundary and Ethernet maximum packet size restriction, each 4K memory page was split into three packets. When a server received a packet, it sent back an acknowledgment packet. In our experiments, we found that the bandwidth from the remote memory layer to be about 9 MB per second. The speed slowdown was caused by the protocol between the client and server and the flow control.

Next, we built a file system on top of the remote memory and used the `lat_fs` in the *lmbench* benchmark [24] to compare the latency time to create/delete files with that of a local file system. From Table I, we found that the file system on the remote memory to be three to four times faster than using local disk in this test. However, most of operations in `lat_fs` were metadata updates and the largest created/deleted file size was only 10K, which

Table I. Number of files created/deleted per second in the file system latency test

File size	Created on local disk	Created on remote memory	Deleted on local disk	Deleted on remote memory
0K	37	134	81	265
1K	27	94	35	143
4K	27	91	33	142
10K	19	68	33	142

did not expose the actual capability of remote memory and local disk. As a result, we used the `lmdd` command to measure the time to read 100 MB data from both raw devices. The remote memory bandwidth in this test was 4.98 MB per second and the local disk was 1.54 MB per second.

Finally, we used `fwrite()` to write 100 MB data to both file systems. The elapsed time of this test on the local disk was 195 seconds and the remote memory was 52 seconds. From this test, we found that the remote memory bandwidth (1.92 MB/sec) in this test reduced to 17 per cent of the bandwidth test from the driver. The reason for the poor performance is that the OSF/1 file system was built on the user-level. Thus, every file access required more context switches than monolithic operating systems. In addition, although the layer design approach avoids the file system modification, it also induces a degradation in performance.

Data migration occurs when a user uses a memory server. To measure the data migration speed during memory revocation, we configured one of the remote memory machines as a backup, with the other two as primary memory servers. When one of the primary servers issued a memory revocation message, the computation server then commanded the active memory server to migrate the memory contents to the backup machine through the remote memory layer. The elapsed time in this test was 10 seconds (including setup time) for 100 MB. We believe that this is an acceptable response time under a system in which memory is allowed to be revoked.

Checkpoint performance

To evaluate the performance of remote memory in checkpointing, we used the checkpoint library designed by Plank *et al.* [25] to checkpoint an *All-Pairs Shortest Paths* solving program, a matrix multiplication program and six scientific programs selected from the SPLASH-2 [26]. The programs' sizes, execution times and their brief descriptions are summarized in Table II. For the sake of experimentation, we adjusted the precision parameters of the six SPLASH-2 programs so that each program will take about 23 to 40 minutes to complete. The checkpoint interval was set to 5 minutes, forcing each program to take 4–8 checkpoints during the test. The execution times of the All-Pairs Shortest Paths and matrix multiplication programs were one hour and three hours respectively. The checkpoint interval for the All-Pairs Shortest Paths was set to 10 minutes and the checkpoint interval for matrix multiplication program was set to 25 minutes. The checkpointed data included heap and stack.

Figures 5 and 6 show the total checkpoint overheads for eight programs that applied a sequential or incremental checkpointing policy combined with and without copy-on-write optimization to remote memory and local disk, respectively. The copy-on-write optimization is a built-in mechanism in Mach's virtual memory system and can be achieved by `fork()`

Table II. Memory usages and execution times of eight applications in the checkpoint testing

Application	Memory used (MB)	Time (sec.)	Brief description
LU	25.7	1784	performs the decomposition of 1280 × 1280 matrix
Ocean	29.0	1345	simulates ocean movements of 258 × 258 grids
Raytrace	38.9	1515	renders a three-dimensional scene of car using ray tracing
Volrend	27.3	1818	renders a three-dimensional volume using a ray casting technique
Water-Nsquared	10.2	2480	evaluates forces and potentials of 9261 water molecules in $O(n^2)$
Water-Spatial	10.1	1818	evaluates forces and potentials of 9261 water molecules in $O(n)$
All-Pair	25.7	3632	solve 2048 × 2048 All-Pairs Shortest Paths problem
Matrix	50.8	10690	matrix multiplication of two 2048 × 2048 matrices

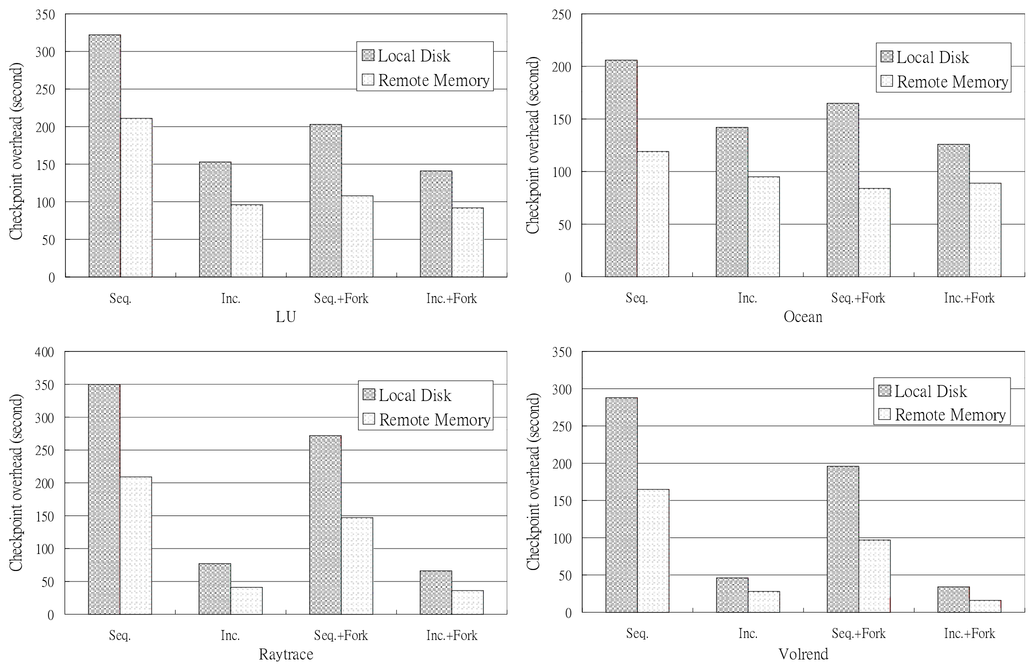


Figure 5. The total checkpoint overhead comparison of LU, Ocean, Raytrace and Volrend

system call. The detailed numbers for program execution times and checkpoint sizes are listed in Table IV in the Appendix. Since the first checkpoint size in incremental checkpointing approximates the size of a sequential checkpointing, the average incremental checkpoint size did not include this into the calculation. LU and Water-Nsquared have variable checkpoint sizes. The detailed numbers of every checkpoint latency and remote memory utilization of these two programs are reported in Tables V and VI.

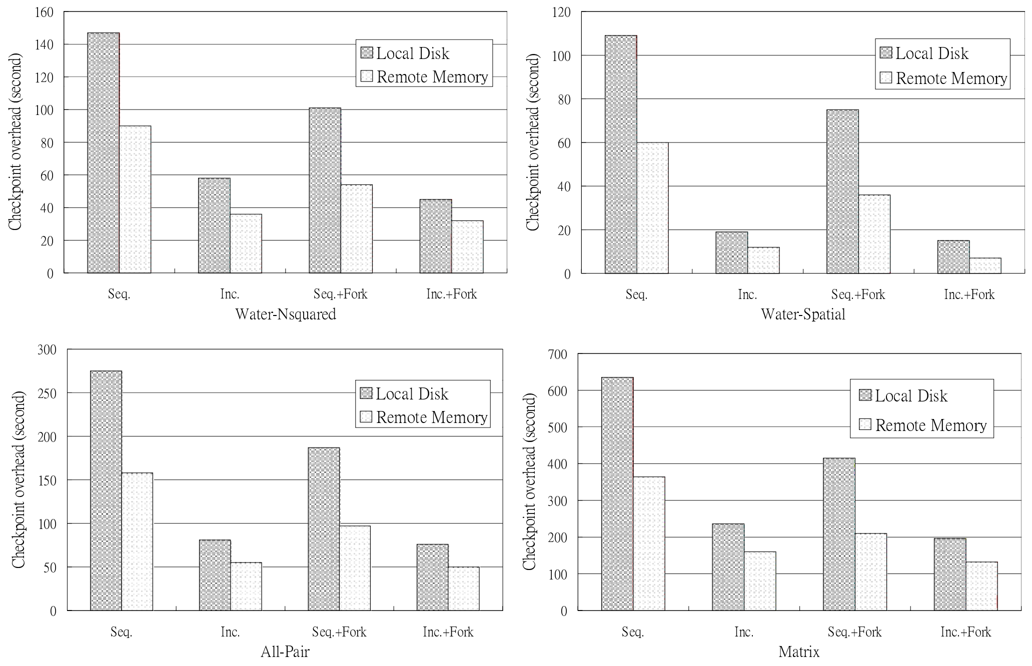


Figure 6. The checkpoint overhead comparision of Water-Nsquared, Water-Spatial, All-Pairs and Matrix

Table III. The checkpoint overheads and the remote memory improvement ratios in four checkpoint policies. Columns marked R/D represent remote memory checkpoint overhead versus local disk checkpoint overhead. Columns marked % are calculated using $(1 - R/D) \times 100$

	Seq.		Inc.		Seq. + Fork		Inc. + Fork	
AP	R/D	%	R/D	%	R/D	%	R/D	%
LU	211/322	34.5	96/153	37.3	108/203	46.8	92/141	34.8
Ocean	119/206	42.2	95/142	33.1	84/165	49.1	89/126	29.4
Raytrace	209/350	40.3	41/77	46.8	147/272	46.0	36/66	45.5
Volrend	165/288	42.7	28/46	39.1	97/196	50.5	16/34	52.9
Water-Nsquared	90/147	38.8	36/58	37.9	54/101	46.5	32/45	28.9
Water-Spatial	60/109	45.0	12/19	36.8	36/75	52.0	7/15	53.3
All-Pair	158/275	42.5	55/81	32.1	97/187	48.1	50/76	34.2
Matrix	364/635	42.7	160/236	32.2	210/415	49.4	132/196	32.7

Improvement by remote memory

Table III reports the remote memory improvement in the four checkpoint policies. The R/D columns represent the remote memory (R) and local disk (D) checkpoint overhead. Columns marked with % are the improvement ratios of using remote memory to replace the disk in the same checkpoint policy. The improvement ratios were calculated using $(1 - R/D) \times 100$.

In sequential checkpointing, checkpoint overhead comes from the I/O access mostly. Thus, a faster storage can reduce the checkpoint overhead significantly. Using remote memory to do sequential checkpointing reduces 34.5 per cent (LU) to 45 per cent (Water-Nsquared) of

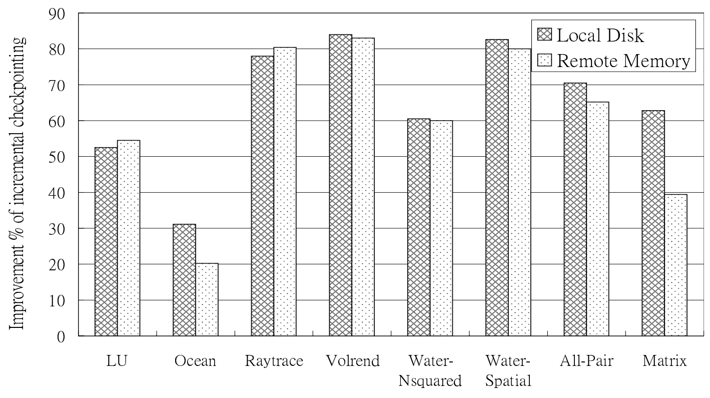


Figure 7. The checkpoint overhead improvement by incremental checkpoint over sequential checkpointing

checkpoint overhead. However, in incremental checkpointing, the improvement by remote memory drops slightly for most programs. This is the cost coming from dirty page handling. Since the operations to identify dirty pages are independent of the speed of storage, it counteracts the effect from using the remote memory. When copy-on-write optimization is applied to sequential checkpointing, using the remote memory has 46–52 per cent checkpoint overhead improvement. This effect comes from both the copy-on-write optimization and using the remote memory.

Improvement by checkpoint policy

Incremental checkpointing reduces the amount of data that must be saved to files. This effect depends upon how many dirty pages exist in each checkpoint interval. Figure 7 shows the improvement ratios of incremental checkpointing over sequential checkpointing. The ratios were calculated using

$$\left(1 - \frac{\mathit{Overhead\_Inc}}{\mathit{Overhead\_Seq}}\right) \times 100$$

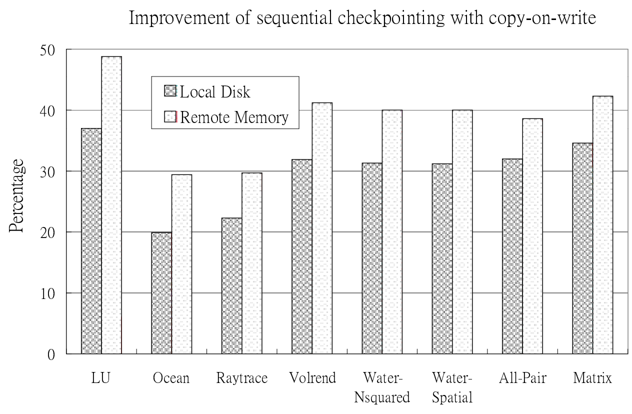
where *Overhead_Inc* is the checkpoint overhead of incremental checkpointing and *Overhead_Seq* is the checkpoint overhead of sequential checkpointing.

The sequential checkpoint sizes of Raytrace, Volrend and Water-Nsquared were 38.9, 27.3 and 10.1 MB, respectively, but their incremental checkpoint sizes were less than 200 KB. The improvement by incremental checkpointing for these three programs was evident. For Raytrace, incremental checkpointing into a local disk reduced 78 per cent of the checkpoint overhead and 80.4 per cent in remote memory checkpoint. For LU and Ocean, their incremental checkpoint sizes were larger than others, thus the incremental checkpointing improvement ratios for these two programs were less than other programs.

Figure 8 shows the variation improvement ratios of the copy-on-write optimization. The left side of Figure 8 presents copy-on-write *versus* non-copy-on-write sequential checkpointing. The right side of Figure 8 is the same comparison, but applied to incremental checkpointing.

These ratios were calculated using

$$\left(1 - \frac{\mathit{Overhead\_COW}}{\mathit{Overhead}}\right) \times 100$$



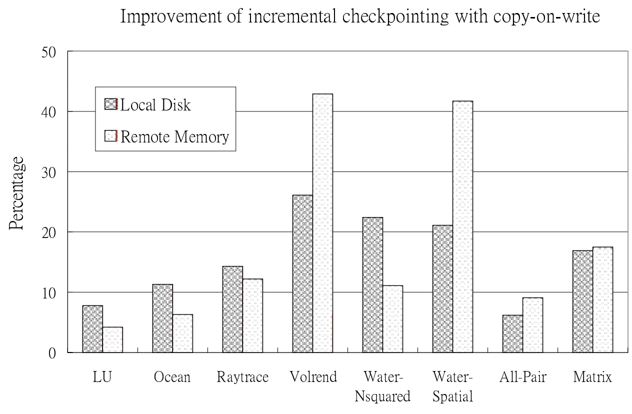


Figure 8. Improvement of copy-on-write optimization on sequential and incremental checkpointing. The top figure is sequential checkpointing with copy-on-write over non-copy-on-write. The bottom figure is incremental checkpointing with copy-on-write over non-copy-on-write

where *Overhead_COW* is the checkpoint overhead when checkpoint policy combined with copy-on-write optimization and *Overhead* is the overhead without copy-on-write.

From left side of Figure 8, the copy-on-write optimization reduces at least 20 per cent of checkpoint overhead in sequential checkpointing. In copy-on-write incremental checkpointing, most of the incremental checkpoint sizes are smaller than sequential checkpoint sizes. The effect of parallel execution is not as noticeable as copy-on-write sequential checkpointing, thus the improvement ratios in copy-on-write incremental checkpointing are no more than 25 per cent. Note that this 25 per cent is the additional improvement of using the copy-on-write optimization. From Figure 7, using the incremental checkpointing approach already has 20–84 per cent improvement over sequential checkpointing. When applying the copy-on-write optimization to incremental checkpointing, the checkpoint overhead was reduced up to 90.3 per cent (Volrend, remote memory).

Improvement of checkpoint latency

Figures 9 and 10 show the latency time in each checkpoint. The latency time using remote memory in sequential checkpointing is about 60 per cent of using a local disk checkpoint. The ratio of incremental checkpoint latency time is the same as in sequential checkpointing.

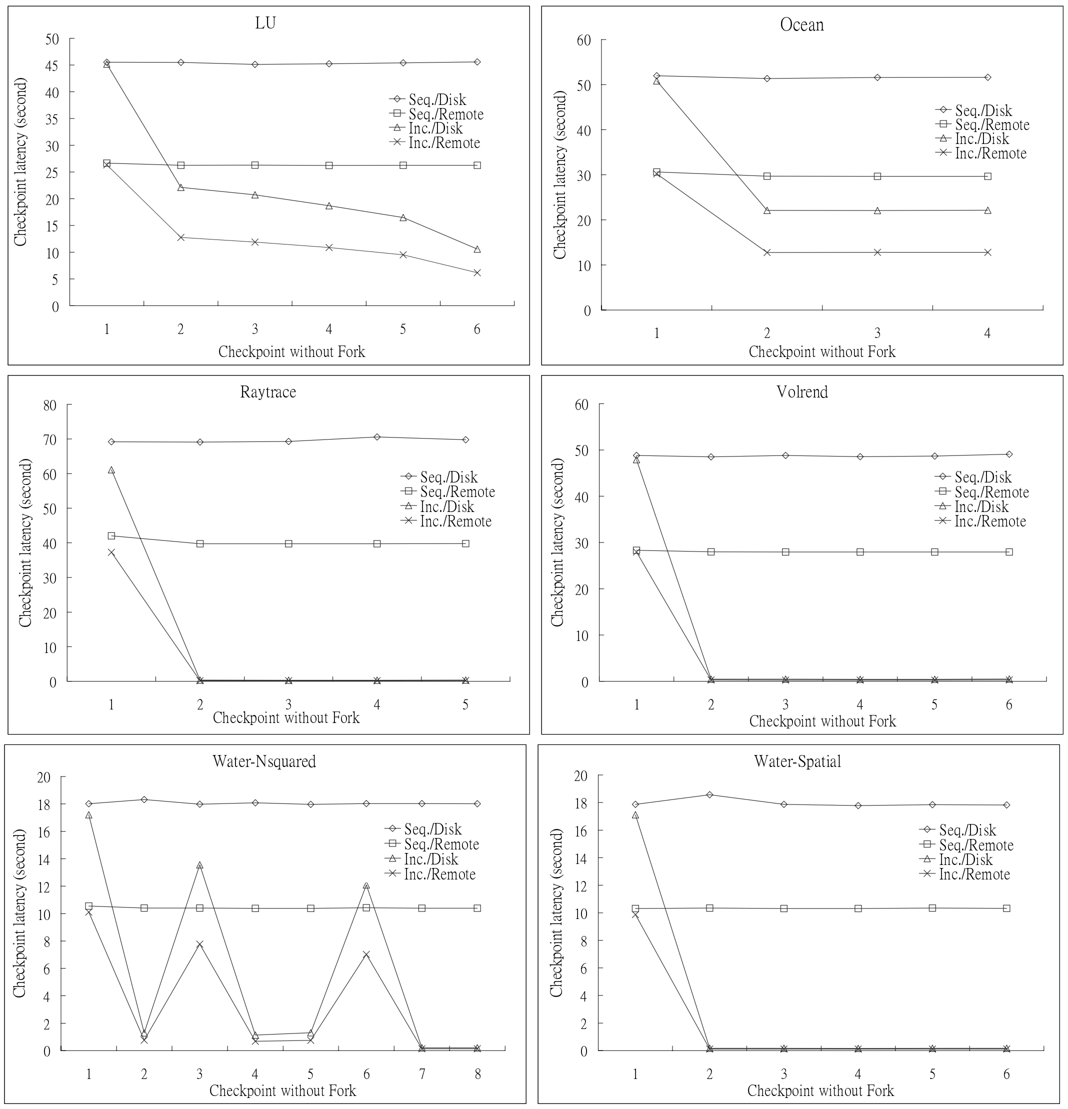


Figure 9. The latency time of every checkpoint for LU, Ocean, Raytrace, Volrend, Water-Nsquared and Water-Spatial when applying sequential or incremental checkpointing to either remote memory or local disk

However, from Figures 9 and 10 their differences are not very clear. For the sake of comparison, we used a program that generates an incremental checkpoint in every additional MB of dirty data. Figure 11 plots the incremental checkpointing latency curve of the remote memory and local disk. When there is no dirty page to checkpoint, the checkpointer is still required to save stack and file header. It took 0.2 seconds for the local disk and 0.15 seconds for the remote memory.

RELATED WORK

Comer and Griffioen [12] proposed a remote memory model on the Xinu operating system. In their model, one or more dedicated machines provide remote memory services. When

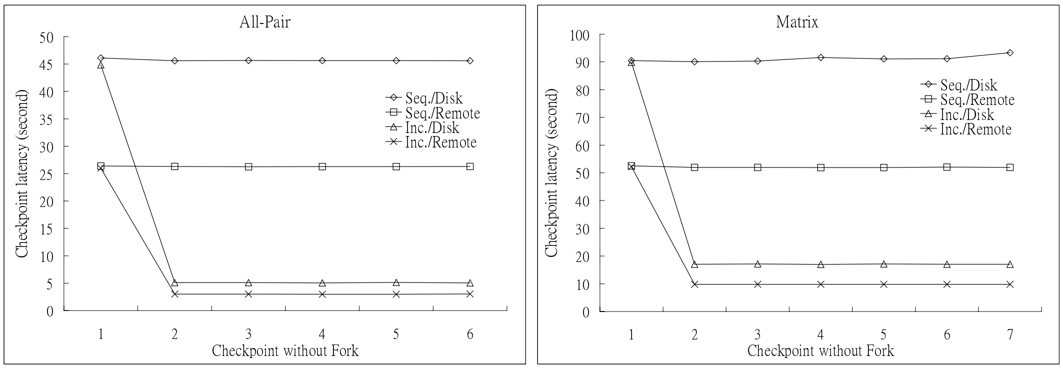


Figure 10. The latency time of every checkpoint for All-Pair and Matrix when applying sequential or incremental checkpointing to either remote memory or local disk

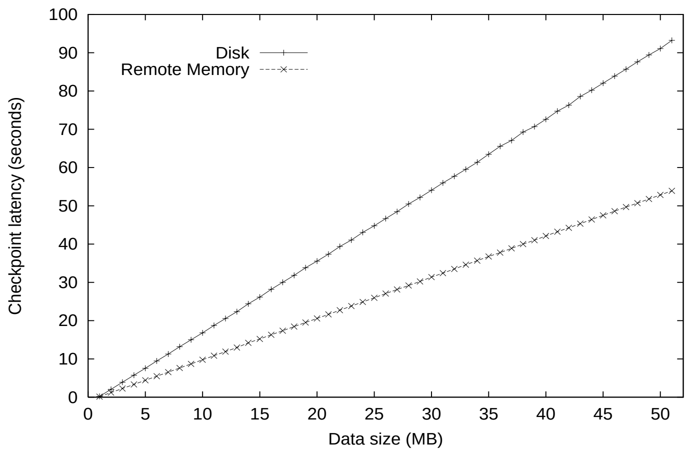


Figure 11. The checkpoint latency curve of incremental checkpoint to local disk and remote memory

a machine exhausts its own physical memory, it sends part of the memory pages to the memory server. Narten and Yavatkar [18] extended this model to utilize idle machines. Each workstation runs a daemon process that monitors the local memory utilization. When the free pages on a workstation have been idle for a long time, these free pages are used by memory servers. A memory server not only uses its own local memory to serve client's requests, but also utilizes memory on idle workstations. Objects stored on a memory server can be shared by multiple clients.

Iftode *et al.* [27] also implemented memory servers for multicomputers. In their model, multi-computer nodes are divided into computation nodes and memory server nodes. A computation node sends dirty pages to memory server nodes. Markatos and Dramitinos implemented a remote memory pager on DEC Alpha workstations and adopted RAID techniques to improve reliability [13]. Felten and Zahorjan [23] discussed many design issues when implementing a remote memory paging system. Schilit and Duchamp [28] implemented remote paging for mobile computers on Mach.

All of the above work used remote memory as a fast paging device. Dahlin *et al.* [11]

presented a simulation study that used remote memory to improve file system caching performance.

Feeley *et al.* [17] proposed a Global Memory Service (GMS) model on a network of workstations. The physical memory was classified as local and global memory. The active pages of the host were stored in a local memory, while the pages of the other machines were stored in a global memory. In their model, the global memory stores only cleaned pages. The remote memory acted as a disk cache. When a page miss occurred, GMS checked whether the missing page could be found in the global memory. If found, the operating system transfered that page into the local memory. However, the GMS did not avoid the disk-write operation. The GMS model was optimized for read performance; thus, small page sizes had short latencies.

In the research on fault-tolerant systems, the process-pair technique is often implemented in distributed systems. For each fault-tolerant application, the operating system creates a backup process located in the remote machine. The primary process periodically checkpoints its states to backup processes. If the primary process fails, the backup process can then take over the task. The basic requirement of this approach is that the primary machine and backup machine need the same instruction set.

Condor is a checkpoint library [10]. The goal of Condor is to use the CPU power of idle workstations and balance the load in the network. Processes in heavily loaded machines can migrate to idle machines and resume execution.

To avoid disk access during checkpointing, one approach involves *diskless checkpointing*, which uses memory and processor redundancy to checkpoint processes [6,19]. This approach is based on *coordinated checkpointing*. A set of machines in the computing environment makes checkpoints to each local memory at the same time and then generates a parity checkpoint to another machine. The parity checkpoint can use the techniques of RAID-5 (exclusive or by all the checkpoints), RAID-1 (mirroring), or other complicated arithmetic coding.

On transputer networks, Silva *et al.* [8] studied and implemented checkpoint schemes that used 1–4 neighbor processors to store checkpoints. Chiueh and Deng [7] studied mirror checkpointing, parity checkpointing and partial parity checkpointing for SIMD MPP machines. They implemented the first two checkpointing schemes and simulated partial parity checkpointing. In mirror checkpointing, the checkpointed data are stored into local memory and the memory of a neighbor processor. The memory of each processor is separated into four sections. Two sections are used for running process and its local checkpoint while the other two are for the neighbor processor. In the parity checkpointing scheme, the memory of each processor is separated into only two sections. One is used to run the process, while the other is used for its local checkpoint. The parity checkpoint is XORed by every processor checkpoint and stored to another control processor. From their experiments, the mirror checkpointing is much faster than parity checkpointing in performance, but produces more storage overhead.

Vaidya proposed a two-level recovery scheme for distributed systems [29,30]. The basic idea of the two-level recovery scheme is ‘make the common case fast,’ which means using the low overhead checkpoint method to tolerate more frequent failures and the high overhead method to tolerate less frequent failures. In their model, accessing the local disk has less overhead than the remote stable storage.

CONCLUSION

Using remote memory provides a different design approach in fault-tolerant services. A system not only uses its own resources, but also the resources of other machines to make fault-tolerant services more efficient. In an environment in which communication speed is becoming faster, exploiting network bandwidth and idle resources will become more and more common. In this paper, we show that efficient use of the physical memory of other workstations can reduce checkpoint overhead and latency when checkpointing.

From the experimental results of eight checkpointed applications, remote memory reduces 34.5 per cent of the overhead for sequential checkpointing and 32.1 per cent for incremental checkpointing. The checkpoint latency of the remote memory is also about 60 per cent of a local disk checkpoint latency. When sequential and incremental checkpointing are combined with copy-on-write optimization, the improvement by remote memory degrades, but it still reduces at least 28.9 per cent of the checkpoint overhead.

Most of the previous studies on remote memory used the remote memory in the data cache to improve the performance in disk reading. However, the concern in checkpointing is the cost to generate a checkpoint file. Thus checkpointing requires a device which has better write performance. Note that the only case in which data read becomes an issue is in process recovery, which is not a common case. Thus the major difference between our model and the previous research is that the characteristics of the application have been changed. Our goal is to provide a fast writing operation.

Although remote memory provides a fast temporary storage with no seek time, as in disk storage, it has an inherent defect of availability. How to solve this availability problem is a very important issue in the design. Fortunately, this is a commonly researched topic and experiences can be drawn from RAID techniques to solve this problem.

ACKNOWLEDGEMENT

This project was partially supported by the National Science Council of the Republic of China under grant No. NSC88-2213-E-009-055.

APPENDIX: RAW DATA

Table IV. The execution time, number of checkpoints and average checkpoint size of eight applications applying different checkpoint policies. Column marked with ‘NO’ is the applications’ execution times without checkpointing. Columns marked ‘S’ and ‘I’ stand for sequential and incremental checkpointing respectively. Columns with ‘+F’ mean the checkpoint policy is combined with fork(). Column marked with ‘#’ is the number of checkpoints generated

		Local disk				Remote memory				Size (MB)		
AP	NO	S	I	S + F	I + F	S	I	S + F	I + F	S	I	#
LU	1784	2106	1937	1987	1925	1995	1880	1892	1876	25.7	-	6
Ocean	1345	1551	1487	1510	1471	1464	1440	1429	1434	29.0	12.4	4
Raytrace	1515	1865	1592	1787	1581	1724	1556	1662	1551	38.9	0.1	5
Volrend	1818	2106	1864	2014	1852	1983	1846	1915	1834	27.3	0.2	6
Water-Nsquared	2480	2627	2538	2581	2525	2570	2516	2534	2512	10.2	-	8
Water-Spatial	1818	1927	1837	1893	1833	1878	1830	1854	1825	10.1	0.05	6
All-Pair	3632	3907	3713	3819	3708	3790	3687	3729	3682	25.7	2.8	6
Matrix	10690	11325	10926	11105	10886	11054	10850	10900	10822	50.8	9.6	7

Table V. The checkpoint latency in seconds of sequential or incremental checkpointing combined with/without fork system call to LU. The size column is the incremental checkpoint size. The last column is the percentage of remote memory utilization after each incremental checkpointing

	Local disk				Remote memory				Inc. size	
CKP	S	I	S + F	I + F	S	I	S + F	I + F	(MB)	%
1	45.51	45.17	45.62	45.37	26.67	26.35	28.48	34.60	25.3	8.4
2	45.49	22.13	45.56	22.23	26.25	12.75	28.82	15.26	12.5	12.6
3	45.12	20.72	45.70	20.93	26.27	11.89	29.25	15.29	11.6	16.5
4	45.22	18.70	45.88	19.22	26.23	10.86	28.85	14.36	10.5	20.0
5	45.42	16.48	46.09	17.15	26.24	9.52	28.33	11.99	9.2	23.0
6	45.58	10.58	47.21	14.38	26.26	6.15	28.28	7.12	5.9	25.0

Table VI. The checkpoint latency in seconds of sequential or incremental checkpointing combined with/without fork system call to Water-Nsquared. The size column is the incremental checkpoint size. The last column is the percentage of remote memory utilization after each incremental checkpointing

	Local disk				Remote memory				Inc. size	
CKP	S	I	S + F	I + F	S	I	S + F	I + F	(MB)	%
1	18.02	17.21	18.02	17.18	10.55	10.11	10.99	10.59	9.6	3.2
2	18.32	1.29	18.06	1.38	10.40	0.76	11.00	0.89	0.7	3.4
3	17.98	13.56	18.03	13.90	10.40	7.77	10.97	8.58	7.5	5.9
4	18.08	1.14	17.96	1.49	10.38	0.68	10.97	1.09	0.6	6.1
5	17.97	1.30	18.09	1.65	10.38	0.75	11.02	1.17	0.7	6.4
6	18.03	12.09	17.94	12.46	10.42	7.02	10.98	7.74	6.7	8.6
7	18.03	0.20	18.16	0.57	10.39	0.14	11.01	0.51	0.1	8.6
8	18.02	0.20	18.14	0.57	10.39	0.14	10.98	0.51	0.1	8.7

REFERENCES

1. P. A. Lee and T. Anderson, *Fault Tolerance–Principles and Practice*, Springer-Verlag Wien, New York, 1990.
2. E. N. Elnozahy, D. B. Johnson and W. Zwaenpoel, 'The performance of consistent checkpointing', *Proc. of 11th Symp. on Reliable Distributed Systems*, October 1992, pp. 39–47.
3. 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).
4. Ö. Babaoğlu, 'Fault-tolerant computing based on Mach', *Proc. of 1990 Mach Workshop*, October 1990, pp. 185–199.
5. J. S. Plank and K. Li, 'Faster checkpointing with $N + 1$ parity', *Proc. of 24th Annual International Symp. on Fault-Tolerant Computing*, June 1994, pp. 288–297.
6. J. S. Plank, K. Li and M. A. Puening, 'Diskless checkpointing', *IEEE Trans. on Parallel and Distributed Systems*, **9**(10), 972–986 (1998).
7. T. Chiueh and P. Dong, 'Efficient checkpoint mechanisms for massively parallel machines', *Proc. of 26th International Symp. on Fault-Tolerant Computing*, June 1996, pp. 370–379.
8. L. M. Silva, B. Veer and J. G. Silva, 'Checkpointing SPMD applications on transputer networks', *Scalable High Performance Computing Conference*, May 1994, pp. 694–701.
9. T. E. Anderson, D. E. Culler, D. A. Patterson and the NOW team, 'A case for NOW (networks of workstations)', *Proc. of 14th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1995.
10. T. Tannenbaum and M. Litzkow, 'The Condor distributed processing system–checkpoint and migration of UNIX processes', *Dr. Dobb's Journal: software tools for the professional programmer*, **20**(2), 40–48 (1995).

11. M. D. Dahlin, T. E. Anderson, D. A. Patterson and R. Y. Wang, ‘Cooperative caching: using remote client memory to improve file system performance’, *Proc. of First Symp. on Operating Systems Design and Implementation*, 1994, pp. 267–280.
12. D. Comer and J. Griffoen, ‘A new design for distributed systems: the remote memory model’, *Proc. of the Summer 1990 USENIX Conference*, June 1990, pp. 127–135.
13. E. P. Markatos and G. Dramitinos, ‘Implementation of a reliable remote memory pager’, *Proc. of USENIX 1996 Annual Technical Conference*, January 1996, pp. 177–189.
14. Open Software Foundation, Design of the OSF/1 Operating System : Release 1.2, Prentice-Hall, 1993.
15. 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.
16. N. H. Vaidya, ‘Impact of checkpoint latency on overhead ratio of a checkpointing scheme’, *IEEE Trans. on Computers*, **46**(8), 942–947 (1997).
17. M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin and H. M. Levy, ‘Implementing global memory management in a workstation cluster’, *Proc. of 15th ACM Symp. on Operating Systems Principles*, December 1995, pp. 201–212.
18. T. Narten and R. Yavatkar, ‘Remote memory as a resource in distributed systems’, *Proc. of Third Workshop on Workstation Operating Systems*, 1992, pp. 132–136.
19. J. S. Plank, ‘Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques’, *Proc. of 15th Symp. on Reliable Distributed Systems*, October 1996.
20. D. A. Patterson, G. Gibson and R. H. Katz, ‘A case for redundant arrays of inexpensive disks (RAID)’, *Proc. of ACM SIGMOD*, June 1988, pp. 109–116.
21. A. Bricker, M. Litzkow and M. Livny, ‘Condor technical summary’, Computer Sciences Department, University of Wisconsin, Madison, WI.
22. F. Douglis and J. Ousterhout, ‘Transparent process migration: Design alternatives and the Sprite implementation’, *Software Practice and Experience*, **21**(8), 757–785 (1991).
23. E. W. Felten and J. Zahorjan, ‘Issues in the implementation of a remote memory paging system’, *Technical Report TR-91-03-09*, University of Washington, Department of Computer Science and Engineering, March 1991.
24. L. McVoy and C. Staelin, ‘lmbench: portable tools for performance analysis’, *Proc. of USENIX 1996 Annual Technical Conference*, January 1996, pp. 279–294.
25. 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.
26. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, ‘The SPLASH-2 programs: characterization and methodological considerations’, *Proc. of 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.
27. L. Iftode, K. Li and K. Petersen, ‘Memory servers for multicomputers’, *Proc. of 38th IEEE International Computer Conference (COMPCON Spring’93)*, February 1993, pp. 534–547.
28. B. N. Schilit and D. Duchamp, ‘Adaptive remote paging for mobile computers’, *Technical Report CUCS-004-91*, University of Columbia, 1991.
29. N. H. Vaidya, ‘Another two-level failure recovery scheme’, *Technical Report TR94-068*, Texas A&M University, December 1994.
30. N. H. Vaidya, ‘A case for two-level distributed recovery schemes’, *Proc. of Joint International Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 64–73.