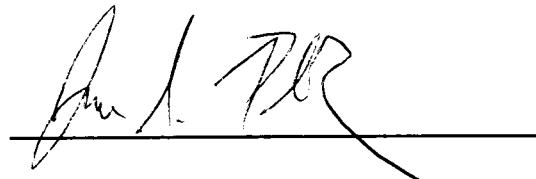


To the Graduate Council:

I am submitting herewith a thesis written by Michael A. Puening entitled "Diskless Checkpointing." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


James S. Plank, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:


Associate Vice Chancellor
and Dean of the Graduate School

Diskless Checkpointing

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Michael A. Puening

May 1997

Abstract

As the choice of parallel platforms shifts from dedicated parallel machines to networks of workstations, the need for program fault-tolerance has never been greater. Checkpointing is the only means to provide programs with fault-tolerance in general-purpose computing environments. Checkpointing usually involves saving program states to disk. However, in parallel environments, stable storage becomes a bottleneck that prevents efficient checkpointing. Presented in this thesis are algorithms to provide parallel programs with fault-tolerance without relying on stable storage. An implementation of these algorithms was created and compared with the traditional disk-based algorithms. Results show that disk-less checkpointing is a viable option to provide efficient fault-tolerance with low overhead.

Contents

1	Introduction	1
1.1	Diskless Checkpointing	2
1.2	Statement of Purpose	4
2	Diskless Checkpointing Algorithms	5
2.1	Basic Diskless Checkpointing	5
2.2	A Simple Diskless Checkpointing Algorithm	9
2.2.1	Analysis of the Simple Algorithm	10
2.2.2	Further Optimizations to the Simple Algorithm	12
2.3	An Incremental Diskless Checkpointing Algorithm	14
2.3.1	An example	17
2.3.2	Analysis of the Incremental Algorithm	20
2.4	Compression to Reduce Network Latency Overhead	23
2.5	Tolerating Failures of More Than One Processor	24
3	Design And Implementation	26
3.1	The PVM Diskless Checkpointing System	26
3.1.1	PVM System State	27
3.1.2	UNIX Kernel State	28
3.1.3	PVM System Protection	28
3.1.4	PVM Message Flushing	29
3.1.5	Task Synchronization	30

3.1.6	Checkpoint Reduction	30
3.2	Diskless Checkpointing Variants	31
3.2.1	Task Startup	33
3.2.2	Checkpointing a UNIX Process	34
3.2.3	Simple Diskless Checkpointing	34
3.2.4	Incremental Diskless Checkpointing	36
3.2.5	Adding Compression	38
3.2.6	Recovery	39
3.3	Disk-based Checkpointing Variants	40
3.3.1	Simple Checkpointing	41
3.3.2	Incremental Checkpointing	42
3.3.3	Adding Compression	42
3.3.4	Recovery	43
3.4	Combinations of Checkpointing Variants	43
4	Experiments And Results	45
4.1	Experiments	45
4.1.1	Parallel Matrix Multiplier (MAT)	47
4.1.2	Cellular Automata (CELL)	51
4.1.3	N-Body Interactor (NBODY)	54
4.1.4	Shallow Water Model (STSWM)	56
4.1.5	Iterative equation solver (PCG)	57
4.1.6	Failures	60
4.2	Results and Discussion	61
4.2.1	Checkpoint Times	61
4.2.2	Checkpointing Overhead	62
4.2.3	Recovery	67
5	Related Work	69
5.1	Previous Work	69

5.2	Current Related Work	70
6	Conclusion and Future Work	72
6.1	Diskless versus Disk-based Checkpointing	73
6.2	Future Work	75
	Bibliography	76
	Vita	81

List of Tables

3.1	Summary of Checkpointing Variants	44
4.1	MAT Results	50
4.2	CELL Results	52
4.3	NBODY Results	55
4.4	STSWM Results	57
4.5	PCG Results	59
4.6	Recovery Results	60

List of Figures

2.1	A simple algorithm for diskless checkpointing.	11
2.2	System state immediately after checkpoint 0.	17
2.3	System state slightly after checkpoint 0.	18
2.4	Processor P_1 initiates checkpoint 1.	19
2.5	Processor P_2 , P_3 , and P_4 take checkpoint 1.	20
2.6	Checkpoint 1 is complete.	21
4.1	The communication steps in Cannon's algorithm on 16 processors.	48
4.2	Checkpoint time versus size for simple diskless checkpointing.	61
4.3	Percentage reduction in checkpoint time using diskless checkpointing.	62
4.4	Diskless checkpointing overhead by program.	64
4.5	Checkpointing overhead by algorithm.	65
4.6	Reductions in overhead using checkpoint optimizations.	66
4.7	Recovery time for application and Checkpoint task failures.	68

Chapter 1

Introduction

As networks of workstations become the platform of choice for performing parallel computations, the need for providing fault-tolerance for them is crucial. Networks are usually very dynamic working environments, and parallel applications have to contend with many different obstacles that prevent useful work from being done. Machines and networks develop different load patterns as users do work. Downtime due to disk backups, regular maintenance, and upgrades can make machines unavailable. Lastly, there is the chance where a machine can fail due to some hardware problem or system crash. Checkpointing can provide programs with fault-tolerance that can make their parallel environment less hostile.

Checkpointing is an important topic in computer science as it is the only way to provide fault tolerance in a general-purpose computing environment [AL81]. It is the method of choice for providing fault-tolerance in parallel and distributed systems [Bir86, BBG⁺89, EZ92, Joh89, KMBT92, PL94b]. Checkpointing traditionally requires the saving of one or more processors' address spaces to stable

storage so that after a failure, the machine's state may be restored from the saved checkpoint. Additionally, checkpointing has been used for process migration, job swapping and debugging.

1.1 Diskless Checkpointing

The major source of overhead in traditional checkpointing is writing the checkpoint to disk. Results from implementations have shown that the overriding concern in making checkpoints fast is either reducing or hiding the overhead of disk writing. This is especially a concern in parallel and distributed systems, where the number of processors is often vastly larger than the number of disks. Proposed solutions to reducing the effect of disk writing have been to use incremental checkpointing [EJZ92, FB89, WM89], compiler support [LF90], compression [LF90, PL94b], copy-on-write [LNP90], non-volatile RAM [KMBT92], and pre-copying [EJZ92]. Although these methods succeed to varying degrees, they all default to the speed of the storage medium as the bottleneck in decreasing overhead.

Diskless checkpointing was first introduced by Plank and Li [PL94a] as a way to use processors instead of disks for fault-tolerance. This removes the disk as the major bottleneck in checkpointing performance, and places the burden on the network. Since networks are usually an order of magnitude faster than disks and show better scalability, diskless checkpointing should show a marked improvement in performance over disk-based checkpointing.

Diskless checkpointing algorithms revolve around $N + 1$ parity, previously used

by Gibson [Gib90] to provide reliability in disk arrays. The idea is to maintain a checkpoint encoding in an extra processor so that if an application processor fails in the middle of a computation, it can be replaced instantly by the extra processor. The algorithms can be extended to use m extra processors such that if any m processors fail at the same time, they can also be replaced by the extra processors.

By eliminating disk writing from the checkpointing algorithm, programmers should be able to checkpoint more frequently than when they have to write to disk. Instead of checkpointing on the order of once an hour, programmers may checkpoint as frequently as every ten minutes. This should drastically reduce the amount of lost work due to processor failures.

Diskless checkpointing allows one's computational model to be one of a continuously running parallel system. If a processor, or up to m processors fail, they can be replaced instantly with any available processor or processors. The system does not have to be halted and restarted when the failing processors are reset. Moreover, the amount of work lost due to failures is small – on the order of minutes. Additionally, the diskless checkpointing algorithm may be used for process migration and/or load balancing in a reconfigurable distributed system. Finally, as there is no reliance on disk, there are no problems concerning the availability of stable storage following a failure or migration.

Combined with an all encompassing checkpointing method for total system failure [EZ92, Joh89, LY87, LNP92, PL94b, SY85], this algorithm provides an efficient composite system for fault-tolerant computing. The faster algorithm is

used to checkpoint at small intervals, like once every ten minutes, and the other all-encompassing algorithm is used to checkpoint at large intervals, like once every few hours. Thus, the more frequent case of a few processors failing (or being shutdown or reclaimed by owners in a distributed workstation environment) is dealt with swiftly, involving no disk interaction, and a minimal loss of computation. The rarer case of whole system failure is handled as well, albeit slowly, as it has more saved state from which to recover.

1.2 Statement of Purpose

This thesis presents algorithms for diskless checkpointing and provides experimental results of implementing these algorithms on a PVM-based network of workstations comprised of 24 SunSPARC workstations. The diskless checkpointing algorithms are compared to standard disk-based checkpointing algorithms with several well known optimizations utilized. The goal of this thesis is to provide a complete understanding of diskless checkpointing and how it may be used to improve standard checkpointing systems on networks of workstations.

In Chapter 2, the diskless checkpointing algorithms are presented along with several optimizations that can be used to improve performance. The details of the PVM-based implementation is discussed in Chapter 3. The experimental results of the checkpointing algorithms on several types of applications are presented and discussed in Chapter 4. Related work is discussed in Chapter 5. Finally, the conclusions are given in Chapter 6.

Chapter 2

Diskless Checkpointing Algorithms

Research has shown that the main source of overhead in creating checkpoints is writing them to disk. This is especially true in parallel environments where disk contention is high. Presented here are algorithms designed to provide fault tolerance to parallel programs without relying on stable storage. Given extra physical memory and an extra m processors, a diskless checkpointing system can provide a parallel program with tolerance of at most m failures with less overhead than conventional disk-based checkpointing algorithms.

2.1 Basic Diskless Checkpointing

Consider a parallel application, called the *target program*, that runs on a collection of n processors $p_1, \dots, p_n$. These processors, termed *application processors*, have

distinct, unshared memories and are connected by a message passing system. To provide fault tolerance for the target program, another collection of m similar *checkpoint* processors $p_{n+1}, \dots, p_{n+m}$ periodically cooperate with the application processors to take a global checkpoint. This global checkpoint consists of all the necessary information such that the collection of $n + m$ processors may tolerate up to m processor failures at any one time. It is important to note that none of the checkpoint information resides on secondary storage, but rather in memory. In the event of a failure, unused processors outside the collection of application and checkpoint processors may replace the failed processors. If no unused processors are available, a checkpoint processor can be used instead with lessened ability to tolerate failures in the future. Recovery of the target program can begin from the point in the program when the last checkpoint was taken using the new replacement processors.

Given a sufficiently large pool of processors, a diskless checkpointing system can provide target programs with a high performance computation engine resilient to processor failures, overly loaded processors and/or processor reclamations by their owners. The pool of processors may grow or shrink depending on circumstances, but as long as there are n processors plus some number of checkpoint processors to provide fault tolerance, the target program can continue running in an efficient manner. To provide fault tolerance for total loss of all or most processors, a diskless checkpointing system must be combined with algorithms to checkpoint to disk at less frequent intervals.

Presented here are explanations of the simplest forms of diskless checkpointing

where $m = 1$. Thus, with $n + 1$ processors, the target program uses n application processors $p_1, \dots, p_n$ and checkpoints to processor p_{n+1} . Descriptions where m is greater than one are presented in Section 2.5.

The diskless checkpointing algorithm centers on the idea that at all points in time, there exists a valid *consistent* checkpoint of the target program in memory. Consistent checkpointing has been well-documented and well-studied [CL85, KT87, LY87, LNP92]. The checkpoint consists of the address spaces of all n application processors, plus a log of messages in transit at the time of the checkpoint. Note that it is possible to eliminate the message state for each checkpoint by having the processors complete a flushing protocol, such as “Sync-and-Stop” [PL94b], to guarantee that all messages have been received.

With diskless checkpointing, the consistent checkpoint is distributed across the processors such that each application processor stores its address space in extra physical memory. Moreover, the checkpoint processor stores an encoding of all these address spaces in its physical memory. If there is a message log, it is stored at the relevant processor as part of its checkpoint. The encoding used here is $N + 1$ parity as used by Gibson to provide fault tolerance in RAID systems [Gib90]. This “parity” checkpoint is defined as follows:

- Let the size of each application processor p_i 's checkpoint be S_i .
- The checkpoint processor maintains a record of each value of S_i for $1 \leq i \leq n$.
- The size S_{n+1} of the parity checkpoint is the maximum S_i for $1 \leq i \leq n$.
- Let $b_{i,j}$ be the j -th byte of p_i 's checkpoint if $j \leq S_i$, and 0 otherwise.

- Each byte $b_{n+1,j}$ of the parity checkpoint is equal to the bitwise exclusive-or ($\oplus$) of the other bytes: $b_{n+1,j} = b_{1,j} \oplus b_{2,j} \oplus \dots \oplus b_{n,j}$ for $1 \leq j \leq S_{n+1}$.

If an application processor p_i fails, the target program can recover from the last checkpoint as follows. First, a replacement processor is selected from a pool of unused processors, or chosen to be the checkpoint processor if none is available. Next, each non-failed application processor recovers its execution state by copying back the address space it stored previously in the extra physical memory. Finally, the replacement processor calculates its address space by receiving its address space size S_i and the parity checkpoint from the checkpoint processor, and all checkpoints from the other non-failed processors. The replacement processor can calculate each byte $b_{i,j}$ for $1 \leq j \leq S_i$ of p_i 's address space as follows:

$$b_{i,j} = b_{1,j} \oplus \dots \oplus b_{i-1,j} \oplus b_{i+1,j} \oplus \dots \oplus b_{n+1,j} \text{ for } 1 \leq j \leq S_i.$$

When this calculation is finished, the replacement processor can restore p_i 's state from its checkpoint. If any messages are present in a message log, then they are also resent at this time. This system can tolerate any one processor failure without relying on stable storage.

Should the checkpoint processor fail, the target program does not have to be rolled back. If there is a processor available to replace the failed processor, the application processors may recalculate the old parity checkpoint using their local checkpoints, or take a new checkpoint.

2.2 A Simple Diskless Checkpointing Algorithm

The simplest diskless checkpointing algorithm consists of saving the state of the target program in its entirety every time a checkpoint is taken. At the start of the target program, all $n + 1$ processors synchronize to take the first checkpoint. Each application processor p_i determines its address space size S_i . This information is sent to, and recorded by, the checkpoint processor p_{n+1} . Next the application processors send the contents of their address spaces to the checkpoint processor. If the network allows for simultaneous communication between multiple pairs of processors, a tree-based fan-in algorithm can be used to reduce the time necessary to take a checkpoint. The checkpoint processor computes the “parity” checkpoint as described above. After the parity checkpoint is calculated, each application processor copies its local address space in its entirety to extra physical memory on the processor. Once the address space is copied, the application processors are free to begin running the target program. The extra memory is essential for this algorithm. Assuming the maximum size S_i of any application processor p_i for $1 \leq i \leq n$ is M , the memory requirements for each processor, including the checkpoint processor is $2M$.

At user-specified intervals, the processors again synchronize to take the next checkpoint. This can be accomplished by a timer interrupt on a coordinating process. The target program is halted and the processors coordinate to determine or eliminate the message state, and then calculate the parity checkpoint. The checkpoint processor calculates this next checkpoint in an area of memory distinct from the previous checkpoint. The validity of the previous checkpoint must remain

intact until the next checkpoint is complete. Once the next parity checkpoint is calculated, the application processors can overwrite the old local checkpoint with their current address space while still being able to recover from a failure. Figure 2.1 pictorially describes the algorithm.

If a failure occurs, the target program is capable of being rolled back to the previous checkpoint. First, the remaining processors decide which checkpoint to use. If the processors have completed checkpoint $i + 1$ then that checkpoint is used. If the failure occurred during the calculation of a checkpoint $i + 1$, then checkpoint i is used. Because checkpoint i is not overwritten until checkpoint $i + 1$ is complete, there is never any disagreement on which checkpoint to use. Once the decision is made of which checkpoint is valid, the process of recovery begins as described above. Note this process works regardless of whether the failed processor was an application or checkpoint processor.

2.2.1 Analysis of the Simple Algorithm

This algorithm imposes both time and space overheads on the target program. Time overhead includes synchronization, parity checkpoint calculation, and checkpoint copying. Compared to the calculation and copying of a checkpoint, synchronization overhead is quite small so its component can be ignored [EJZ92, PL94b]. There are two ways in which the parity checkpoint can be calculated. All n application processors can send their address spaces to the checkpoint processor and have it calculate the parity checkpoint, or the parity checkpoint can be calculated in a distributed fashion using a fan-in algorithm. In the former case, the time

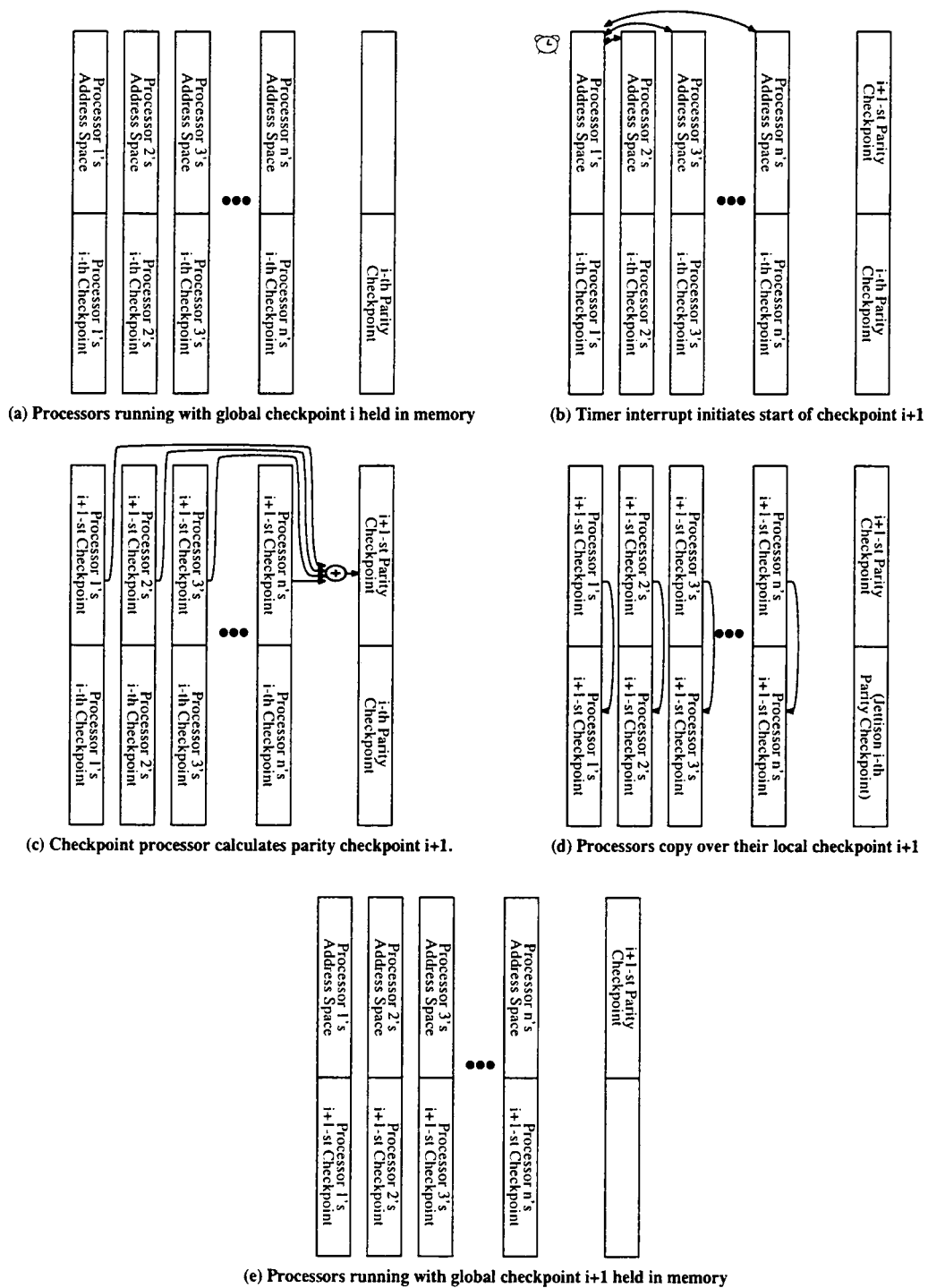


Figure 2.1: A simple algorithm for diskless checkpointing.

overhead for each checkpoint will be

$$Ov_{Simp} = M(\frac{n}{R_{MsgTrans}} + \frac{(n-1)}{R_{Xor}} + \frac{1}{R_{Copy}})$$

where Ov_{Simp} represents time overhead for simple diskless checkpointing. The factors $R_{MsgTrans}$, R_{Xor} , and R_{Copy} represent the rate (in bytes per second) of sending messages over the network, performing exclusive-or operations, and copying memory to memory respectively. M is the size in bytes of an address space. In the case where fan-in is used, the time overhead will be

$$Ov_{Simp} = M(\frac{\log n}{R_{MsgTrans}} + \frac{(\log n)-1}{R_{Xor}} + \frac{1}{R_{Copy}})$$

The choice of which method to use depends on the network. On networks that allow simultaneous communication between multiple pairs of processors, the fan-in algorithm is preferable. Broadcast networks such as Ethernet do not allow such communication, thus the former algorithm must be used.

Space overhead of this algorithm is M bytes per application processor and $2M$ bytes for the checkpoint processor.

2.2.2 Further Optimizations to the Simple Algorithm

The simple algorithm requires all processors to halt the target program for the entire period that processors are taking a checkpoint. This can leave many processors idle while others finish. That time can be better spent executing the target program. However, processors may not overwrite the previous checkpoint until all processors have finished. In order to begin executing the target program again, processors must have an additional M bytes of memory to hold a copy of the latest

checkpoint. The algorithm can be altered as follows. After synchronizing, each application processor copies its address space to extra memory distinct from the previous checkpoint. The parity checkpoint is calculated using the same method as before. However, when a processor is not occupied sending bytes or calculating parity, it can be running the target program. This does not affect the previous checkpoint. Using the extra M bytes of memory should reduce the time overhead by making the algorithm more asynchronous at the expense of $2M$ bytes of space overhead per application processor.

If a processor fails during the calculation of checkpoint $i + 1$, processors that have finished checkpoint $i + 1$ can still roll back to checkpoint i because it has yet to be discarded. Checkpoint i is only discarded when all processors have finished checkpoint $i + 1$.

Some operating systems allow for additional threads or child processes to execute with a shared same address space. The algorithm can make use of these features by having a thread or child process calculate the checkpoint while another thread resumes execution of the target program. This algorithm improves on the algorithm above since the address space does not have to be explicitly copied. A copy-on-write feature of operating systems protects the checkpoint memory and handles the copying of the address space should the target program's address space change from the checkpoint.

2.3 An Incremental Diskless Checkpointing Algorithm

The above algorithm can be improved by using incremental checkpointing [FB89, WM89]. This technique solves two problems with the simple algorithm. First is the exorbitant use of memory. With simple diskless checkpointing, less than half of an application processor's memory is used by the target program. Second is the overhead imposed by sending the entire address space over the network regardless of whether it needs to be. Incremental checkpointing solves these problems by only checkpointing the portions of the address space that have changed since the last checkpoint.

In order for a target program to be checkpointed incrementally, programs must have access to the memory management unit (MMU) of each processor. This enables the program to protect pages of the address space as *read-only* or *read-write* and catch resulting page faults. As with the simple algorithm, this algorithm also requires some additional physical memory. This extra memory is allocated for *primary* and *secondary* checkpoint buffers which are initialized to be empty.

The algorithm works as follows. At the beginning of the target program the processors synchronize to take a first checkpoint. This is similar to taking the first checkpoint using the simple algorithm. Application processors send the size and contents of their address space to the checkpoint processor. The checkpoint processor records the sizes, and calculates the parity checkpoint. Next, application processor protects all the pages of its address space as *read-only*. This differs from the simple diskless algorithm because no copies of any address spaces are made. The application processors now begin executing the target program.

While executing, each processor generates page faults as the program attempts to write a *read-only* page. The program catches these faults and does the following. A page from the primary checkpoint buffer is allocated and the faulting page is copied to this page. Next, the protection on the faulting page is set to *read-write* and the program resumes execution.

The checkpoint for the target program consists of the parity checkpoint, the read-only pages of each application processor's address spaces, and the copied pages in all the primary checkpoint buffers. Should an application processor fail at this point, the surviving processors can roll back to their previous checkpoints by copying (or mapping) pages from the primary buffer back to their address spaces. After calculating the failed processor's checkpoint, all processors reprotect their address spaces to *read-only* and continue processing as before.

At some later point in time, the processors must again synchronize to take the next checkpoint. The algorithm used to achieve consistency can be the same as in the simple algorithm. Once the processors are synchronized and the message state dealt with properly, they perform the following algorithm to calculate the next checkpoint. For each *read-write* protected page $page_k$ in its address space, a processor will:

- Calculate $diff_k = page_k \oplus buf_k$, where buf_k is the saved copy of $page_k$ in the processor's primary checkpointing buffer.
- Send the page number k along with $diff_k$ to the checkpoint processor.
- Set the protection of $page_k$ to be *read-only*.

After sending all the pages, the processor swaps the identity of the checkpoint buffers. Execution of the target program resumes, and the processor now copies faulting pages to what used to be the secondary buffer. When all processors have finished the checkpoint, the processor can reset the secondary buffer to be empty.

Upon receiving page $diff_k$ from a processor, the checkpoint processor copies its $page_k$ from the parity checkpoint to its primary buffer. Next, it updates the copy of $page_k$ in the stored parity checkpoint as follows:

$$page_k = page_k \oplus diff_k$$

This has the effect of subtracting the application processor's old page from the parity checkpoint and adding the new page. The checkpoint processor needs to copy $page_k$ to the primary buffer in case there is a failure during a checkpoint. The page need only to be copied once should it receive multiple $diff_k$'s from several processors.

If a processor fails during the calculation of checkpoint $i + 1$, the system can still be rolled back to checkpoint i . First consider an application processor that has not started checkpoint $i + 1$. This processor simply copies or maps the pages in the primary buffer back to the address space and resets them to *read-only*. If instead a processor has completed checkpoint $i + 1$, it too can roll back to checkpoint i . First it copies or maps pages from the primary buffer to the address space to bring it to the state of checkpoint $i + 1$. Next, the processor does the same for the secondary checkpoint buffer to roll back to checkpoint i . Recall that the secondary buffer is not cleared until all processors have finished checkpoint $i + 1$. The checkpoint processor can roll back to checkpoint i by copying or mapping

pages from the primary buffer back to the parity checkpoint. Once all processors have rolled back to checkpoint i , recovery of the failed processor can begin.

2.3.1 An example

Consider an example system of four application processors $P_1, \dots, P_4$ and one checkpoint processor P_5 running the incremental diskless checkpointing algorithm. At the beginning of the program, all processors take checkpoint 0. The application processors send the size and contents to the checkpoint processor and protect their address spaces as *read-only*. The checkpoint processor calculates and stores the parity checkpoint. All checkpoint buffers are empty at this time. The state of this system is represented in Figure 2.2.

The application processors now run the target program. When page faults occur, the faulting pages are copied to the primary checkpoint buffer. Their protections are set to *read-write* and the application continues. Figure 2.3 shows the

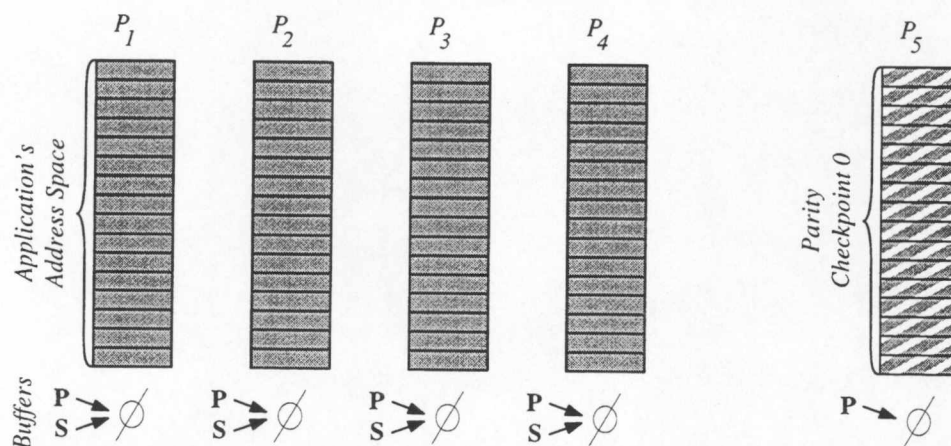


Figure 2.2: System state immediately after checkpoint 0.

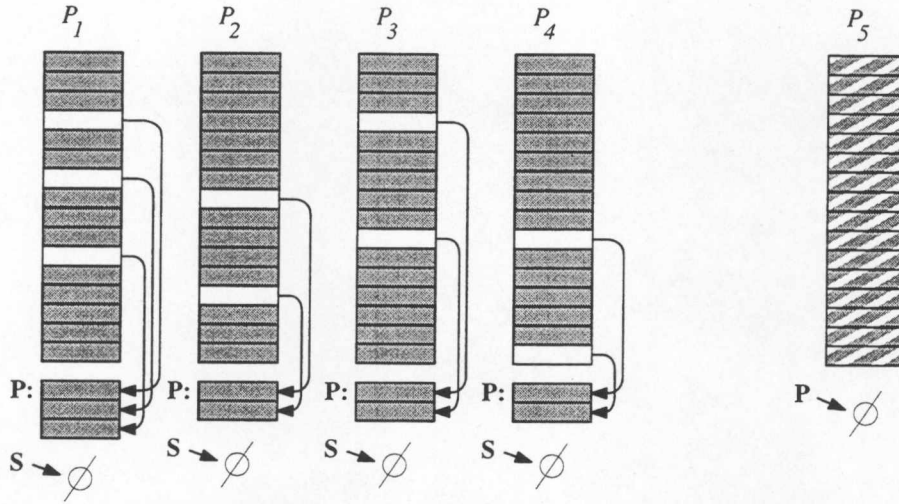


Figure 2.3: System state slightly after checkpoint 0.

example program slightly after the first checkpoint. Processor P_1 has copied three of its faulting pages to primary checkpoint buffer. Processors P_2 through P_4 have copied two pages each. If a failure were to occur to one of the application processors, the remaining processors can roll back to checkpoint 0 by simply copying or mapping the pages in the primary buffer back to their corresponding place in the address space. These pages would then be reset to *read-only*. With the restored address spaces and the parity checkpoint, the failed processor's address space can be reconstructed. If P_5 fails, it may be reconstructed by the application processors without requiring them to roll back. They can do this by sending the replacement checkpoint processor all their *read-only* pages and the pages in the primary checkpoint buffer.

At some later time, P_1 initiates checkpoint 1. All the processors take the necessary steps to synchronize and prepare for a consistent checkpoint. Processor

P_1 calculates $diff_k$ for each changed page k and sends it to P_5 . Before P_5 updates checkpoint 0, it first copies the corresponding page to its primary checkpoint buffer. After sending P_5 all of its checkpoint information, P_1 protects its pages to be *read-only* and swaps the identity of its primary and secondary checkpoint buffers. It can then continue running the target program. Figure 2.4 shows P_1 checkpointing its state to P_5 .

Should an application processor other than P_1 fail at this time, the system can again roll back to checkpoint 0. Processor P_1 copies pages from its primary buffer, then from its secondary buffer. This will restore it to the state of checkpoint 0. Other non-failed processors only need to copy pages from the primary buffer since they have not started checkpoint 1. As before, if P_5 fails, its state can be reconstructed by the application processors without them having to roll back.

Figure 2.5 shows processors P_2 , P_3 , and P_4 checkpointing to P_5 . As P_1 did,

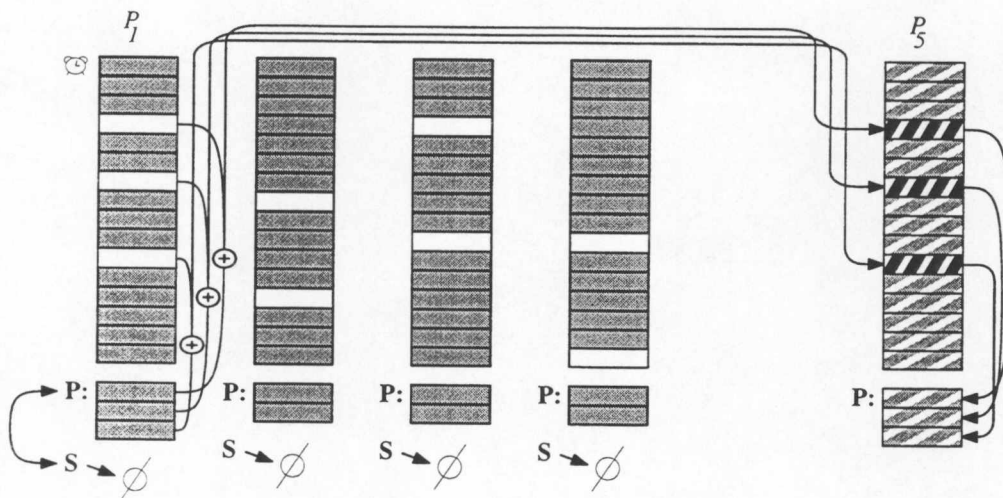


Figure 2.4: Processor P_1 initiates checkpoint 1.

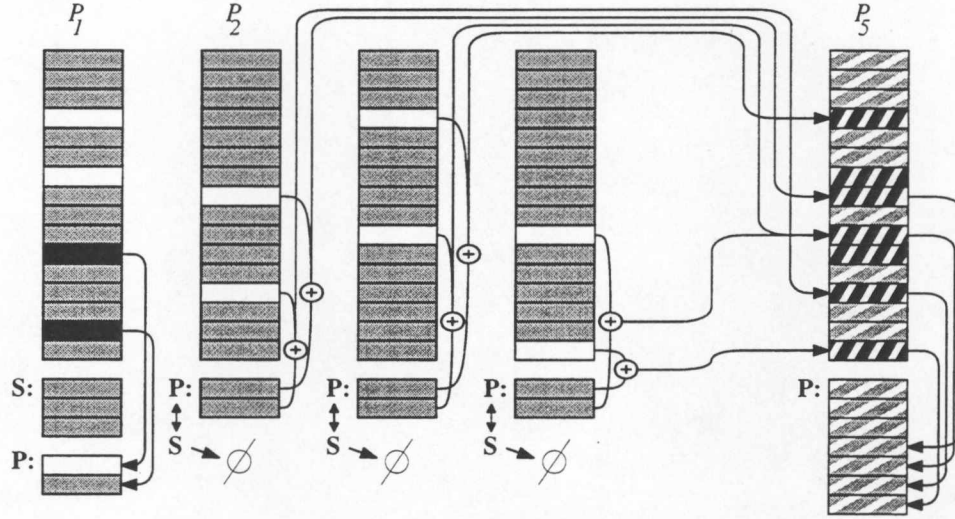


Figure 2.5: Processor P_2 , P_3 , and P_4 take checkpoint 1.

they calculate their *diff*'s and send them to P_5 . Next, they protect their address space to be *read-only* and swap the identity of their checkpoint buffers.

Finally, Figure 2.6 depicts the state of the system when all local checkpoints are finished. Parity checkpoint 1 is now stored by P_5 which means that all application processors can clear their secondary checkpoint buffers. Processor P_5 can also clear its primary buffer because it is no longer needed. Failures between now and the time checkpoint 2 is complete can recover to checkpoint 1.

2.3.2 Analysis of the Incremental Algorithm

Incremental diskless checkpointing imposes several sources of overhead to a target program. These overheads are synchronization, page fault processing, copying pages to checkpoint buffers, calculating *diff*'s, and sending *diff*'s to the checkpoint processor. As with the simple diskless algorithm, synchronization overhead

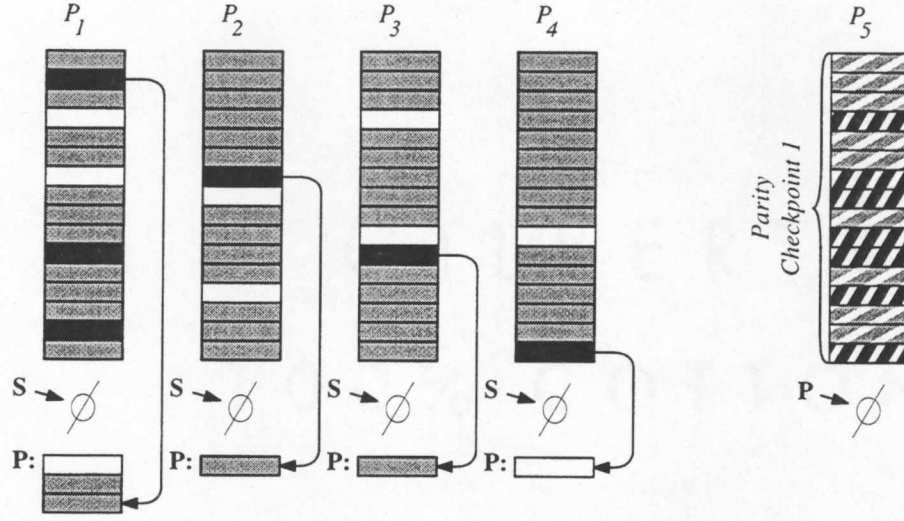


Figure 2.6: Checkpoint 1 is complete.

is quite small and can be ignored.

Let M_{prim} be the number of bytes in the primary checkpoint buffer. The total time overhead of each incremental diskless checkpoint on a broadcast network will be

$$Ov_{Inc} = M_{prim} \left(\frac{1}{R_{pf}} + \frac{1}{R_{copy}} + \frac{n}{R_{Xor}} + \frac{n}{R_{MsgTrans}} \right)$$

where R_{pf} , R_{copy} , R_{Xor} , and $R_{MsgTrans}$ are the rates (in bytes per second) for processing page faults, copying memory to memory, calculating parity, and sending messages respectively. On a switched network the overhead will be

$$Ov_{Inc} = M_{prim} \left(\frac{1}{R_{pf}} + \frac{1}{R_{copy}} + \frac{\log(n)}{R_{Xor}} + \frac{\log(n)}{R_{MsgTrans}} \right)$$

Thus, incremental diskless checkpointing should outperform simple diskless checkpointing whenever M_{prim} is small enough to offset the extra work of processing page faults. Expressed in terms of simple overhead, Ov_{Simp} , the total overhead

per incremental diskless checkpoint will then be

$$Ov_{Inc} = Ov_{Simp} - (M - M_{prim})\left(\frac{1}{R_{copy}} + \frac{n}{R_{Xor}} + \frac{n}{R_{MsgTrans}}\right) + \frac{M_{prim}}{R_{pf}} + \frac{M}{R_{Xor}}$$

and

$$Ov_{Inc} = Ov_{Simp} - (M - M_{prim})\left(\frac{1}{R_{copy}} + \frac{\log(n)}{R_{Xor}} + \frac{\log(n)}{R_{MsgTrans}}\right) + \frac{M_{prim}}{R_{pf}} + \frac{M}{R_{Xor}}$$

for broadcast and switched checkpoints respectively. Since R_{pf} should be small compared to R_{copy} , R_{Xor} , and $R_{MsgTrans}$, incremental diskless checkpointing should decrease the time overhead in most cases. It will only perform worse when the target program displays bad locality and all (or almost all) of the program's address space is modified between checkpoints.

The space overhead of incremental diskless checkpointing is the size of the primary and secondary checkpoint buffers at the instant before the secondary buffer is discarded after a checkpoint. In the worst case, this is $2M_{prim}$. Since M_{prim} can be as large as M , the size of the address space, the maximum space overhead is $2M$. However, in many cases, programs maintain some data locality so processors can use much less space overhead.

The space overhead of the checkpoint processor can range anywhere from $M + M_{prim}$ if the processors change the same pages and $M + nM_{prim}$ if they each change different pages. Since pages are copied the primary checkpoint once, the worst case space overhead is $2M$.

2.4 Compression to Reduce Network Latency Overhead

When using the incremental checkpointing algorithm, $diff_k$'s may frequently have their bytes be zeros. Consider a page k that has been sparsely written during the checkpointing interval. During the next checkpoint, the calculation of $diff_k$ will yield a page that is mostly zeros. Therefore, there is an opportunity to use run-length encoding or bitmap-based algorithms to quickly compress away the zeros. Sending the compressed page can reduce the overhead latency of sending pages to the *checkpoint* processor.

The speed of the compression algorithm is important. In order to decrease the time overhead, the compression factor (F_{Comp}) of the algorithm must be large enough to offset the rate of compression (R_{Comp}) and uncompression (R_{UnComp}) compared to the speed of the network ($R_{MsgTrans}$). Because pages must be compressed before sending, and uncompressed before being used in the parity calculation, the following condition must be satisfied for compression to be used successfully [PL94b]:

$$F_{Comp} > \frac{R_{Comp} + R_{UnComp}}{R_{MsgTrans}}$$

The non-incremental checkpointing algorithms can also benefit by using compression. Because checkpoint i is in memory at the time checkpoint $i + 1$ is being calculated, the simple algorithm can be changed to calculate and send $diff_k$ instead of page k itself. The checkpoint processor can calculate the parity checkpoint as before but it must also exclusive-or checkpoint i with checkpoint $i + 1$ since the $diff$ pages only represent changes since the last checkpoint.

2.5 Tolerating Failures of More Than One Processor

The algorithms described above allow a checkpointing system to tolerate one failure using one checkpoint processor. In this section systems that can tolerate multiple failures using more than one checkpoint processor are described.

The simplest scheme to tolerate more than one failure still uses one dimensional parity, but the parity is partitioned across processor groups [GHK⁺89] rather than all of the processors. If there are m *checkpoint* processors, the *application* processors are divided into m equally sized groups. Each group has one checkpoint processor which calculates the parity checkpoint of just the processors in that group. In order for the checkpoint to still remain consistent however, all processors must synchronize together even though they are in separate groups. This system can tolerate up to m processor failures as long as each failed processor is in a separate group. On a network allowing simultaneous communication of more than one pair of processors, this algorithm also relieves the bottleneck of the single checkpoint processor since the checkpoint calculation is now more distributed.

To tolerate two simultaneous failures of arbitrary processors, the parity calculation can be extended to EVENODD parity [BBBM94] using two checkpoint processors. Here each of the checkpoint processors calculates an even and odd parity respectively. This provides the system with enough information to restore the system after any two failures at one time.

With two-dimensional parity, $2\sqrt{n}$ checkpoint processors are used to store even more parity information. The checkpoint processors are assigned to either a row or column as if the application processors lay on a square grid. Parity checkpoint

encodings are made for each row and column and stored on the corresponding checkpoint processor. This creates a system that can tolerate any 2 failures at a time and up to $2\sqrt{n}$ failures [GHK⁺89] since each application processor is covered by two checkpoint processors.

General *Reed-Solomon coding* provides the means for the most general use of checkpoint processors. In such a system, each checkpoint processor calculates and stores a coding using a set of linearly independent functions. $N + 1$ parity is an example of one such function. A system using Reed-Solomon coding can be built that can tolerate any m failures with exactly m checkpoint processors [MS77, BM93, PFL94].

None of these algorithms is perfect for all applications. Each has its advantages and disadvantages based on checkpoint calculation overhead, checkpoint processor needs, and fault coverage.

Chapter 3

Design And Implementation

A full implementation of all algorithms presented in the previous chapter was written and incorporated into the Parallel Virtual Machine (PVM). Furthermore, complementary disk based algorithms were also implemented so that a full evaluation of the diskless checkpointing algorithms can be made.

3.1 The PVM Diskless Checkpointing System

The PVM system [SGDR93, BDG⁺93] allows a collection of heterogeneous machines connected by a network to work as single parallel computer. PVM has rich message passing capabilities, plus the added advantages of excellent process control and signaling and automatic fault detection. A version of PVM 3.3.8 was altered so that parallel applications could benefit from checkpointing. This version was used on a collection of 24 SunMicrosystem SPARC-5 workstations running SunOS 4.1.3. Each machine contains 96 megabytes of memory and ma-

chines are connected by switched Fast-Ethernet (100Mb) as well as a broadcast Ethernet (10Mb). The Fast-Ethernet switch allows for simultaneous communication between separate pairs of computers, so the diskless checkpointing algorithms benefit from using a fan-in algorithm. Programs may be written in either C or FORTRAN. The checkpointing algorithms were added to the same library to which the programs must be linked in order to obtain PVM capabilities. All checkpointing capabilities are transparent to and independent of the application. The following sections discuss changes and additions to PVM to create a diskless checkpointing system.

3.1.1 PVM System State

PVM works as a single parallel computer by having a daemon reside on each processor in the virtual machine. Target programs are divided into multiple tasks with one task per processor. The daemons provide tasks with message passing ability along with other functions. When a task begins, it enrolls into PVM by contacting the daemon. The daemon determines the authenticity of the task and if valid, assigns the task an identification number. This *tid* is used by PVM to identify tasks and determine routing for messages to and from this task. In the event of a failure, it is necessary for a replacement daemon and task to regain the identity of the failed ones. In this version of PVM, daemons are started using the lowest unused identifier available. This guarantees that a replacement daemon will get the identity of the failed daemon. The *tid* of an application task can be obtained from the recovered address space. When a recovering task re-enrolls

into PVM, the recovered *tid* is sent to the daemon as part of the hand-shaking procedure. This allows a recovering task to regain the identity of the failed task. With the dynamic message passing ability of PVM, the system state is now fully recoverable. Only minor changes to PVM were required to obtain this ability.

3.1.2 UNIX Kernel State

PVM tasks also have the ability to use operating system facilities like file access and thus should be recoverable. However, to recover most items requires changes to the operating system. Since doing so is difficult and not always possible, only the file system state of the process is recovered. This can be done in UNIX without having to alter the kernel. Other facilities like process identifiers and time are not recovered and must be avoided by the application. The file system state is captured by intercepting the system calls `open()`, `close()`, `read()`, `write()`, and `lseek()`. Filenames, access permissions, and seek pointers are recorded for every checkpoint. After recovering from a failure, a task will reopen all files with the proper permissions and return their seek pointers to the correct position.

3.1.3 PVM System Protection

Checkpointing signals sent to application tasks are asynchronous. Thus, there exists a danger when they are sent. PVM functions like sending and receiving messages, as well as memory allocation functions like `malloc()` should not be interrupted. If they are, internal data structures can get corrupted causing the process to fail or produce incorrect results. In order to provide protection, two

functions `ckpt_lock()` and `ckpt_unlock()` prevent signals from damaging the system state. The locking function is called at the beginning of every critical function. It increments a semaphore variable. The unlocking function decrements this variable. The signal handling function of every task first checks this semaphore variable, and if non-zero, the function may not continue. Instead, the signal is marked for resending and the handler returns. When the unlocking function decrements the semaphore variable to zero, it checks whether or not any signals need to be resent. If so, they are done at this time for they now pose no danger to the task.

3.1.4 PVM Message Flushing

Upon receiving a signal to checkpoint, a task must synchronize with the others of the target program. To do so, all tasks perform a message flushing algorithm. This procedure provides two functions, one to synchronize the tasks, and the other to eliminate any message state. Given PVM tasks 1 and 2, where task 1 sends message *A* followed by *B* to 2, PVM guarantees that task 2 will receive message *A* before *B* even if *B* actually arrives before *A* due to routing traffic. Therefore, if every task sends and receives a “flush” message to and from every other task, the checkpoint can be guaranteed to have no message state. However, this algorithm is $O(n^2)$ in the number of messages sent where n is the number of tasks in the target program. If more changes were to be made to PVM, an $O(n)$ algorithm could be made, but results from this implementation show that this algorithm is quite acceptable. Assuming no large messages need to be copied over the network,

a target program with 16 tasks can synchronize and flush messages in less than a second. With checkpoint times several orders of magnitude larger, this time is negligible.

3.1.5 Task Synchronization

Task synchronization in the checkpointing system is provided by the PVM group server and signaling functions. All PVM applications wishing to be checkpointed are required to have tasks join a common group using the PVM group server, `pvmgs`. Applications are started by a host program called `pvmckpt`. This program is responsible for detecting failures and starting periodic checkpoints. To begin a checkpoint, a new group function `pvm_groupckpt()` is provided. It is used to checkpoint a target program to either a disk or a checkpoint processor. When the group server receives this message, it sends a UNIX signal (SIGUSR1) using `pvm_sendsig()` to all members of the specified group. Group members, upon receiving the signal, begin a new checkpoint. When each task completes a checkpoint, the group server is informed of the completion through another group function `pvm_ckptack()`. This function is used to propagate errors back to the `pvmckpt` program. The `pvm_groupckpt()` function returns any errors when the checkpoint is complete.

3.1.6 Checkpoint Reduction

This implementation was built for a switched network. Therefore a fan-in algorithm was able to be used to reduce the time to calculate a parity checkpoint.

After the tasks eliminate the message state, they build TCP sockets between themselves to perform a reduction similar to that done on hypercube architectures. TCP sockets are used since they are much faster than using PVM messages which must travel through at least two daemons. The parity checkpoint is calculated in a pipelined reduction fashion on a per page basis, thus allowing multiple pages of the parity checkpoint to be in different stages of calculation simultaneously.

3.2 Diskless Checkpointing Variants

Many checkpointing algorithms were implemented to explore diskless checkpointing fully. The algorithms differ along four degrees of freedom: address space considerations, compression, type of storage, and synchronization. These degrees refer to the following:

- **Simple/Incremental:** With simple checkpointing, every task saves its entire address space for every checkpoint. Incremental checkpointing has each task examine its address space and only checkpoints those pages that have changed since the last checkpoint. The tradeoff is having less amount of memory to checkpoint while having to determine which pages need to be checkpointed using page protection hardware.
- **Compression:** Compression causes the checkpointing algorithms to compute a $diff_k$ if not done already, and compress it using fast compression techniques before sending it across the network. If not enabled, all pages are

sent uncompressed.

- **Diskless/Disk:** To judge the benefits of diskless checkpointing, disk based algorithms were also implemented. Diskless checkpointing algorithms require tasks to maintain an in-memory checkpoint and calculate a new checkpoint to a *checkpoint task*. Disk based checkpointing algorithms simply have tasks write their address space to a disk file. Maintaining in-memory checkpoints in disk-based algorithms is unnecessary except when compression is used. In that case, tasks need to calculate $diff_k$'s. Therefore memory must be set aside so the task can have a copy of its previous checkpoint.
- **Sync/Async/Fork:** When the target program begins to take a checkpoint, all tasks are sent a signal. The target program must respond to the signal and take a checkpoint. There are three distinct cases in how tasks can behave relative to each other when taking a checkpoint.
 - **Sync:** With full synchronized checkpointing, the target program is held frozen for the entire period that all tasks are checkpointing. This is required in the case of simple diskless checkpointing where tasks only retain a single checkpoint at one time. If tasks were to continue before others finished, the system could be in a state unable to recover from a failure.
 - **Async:** In the case of asynchronous checkpointing, tasks may restart its share of the target program immediately after finishing checkpointing regardless of whether other tasks have completed their checkpoint. For

example, when checkpointing to disk, tasks do not need to wait for other to finish because previous checkpoints are on disk to take care of failures during a checkpoint. When extra memory is available on each processor, diskless checkpointing algorithms can be asynchronous by having tasks store two checkpoints at any one time. This way, an old checkpoint is not discarded until all tasks have finished checkpointing.

- **Fork:** Finally, tasks may fork off a child and have it complete the checkpoint while it resumes executing the target program. Because the operating system performs the `fork()` system call with copy-on-write, the process of forking is fast, and uses minimal memory. Having a child process perform the checkpoint is a standard and very effective method for reducing checkpoint overhead [PL89, EJZ92, LNP94, PBKL95].

3.2.1 Task Startup

Every task of a target program must yield control to an initialization function at startup. The purpose of this function is to allocate data structures and buffers to store checkpoint data depending on how the target program wishes to be checkpointed. It also sets up signal handling functions to catch signals. The function first reads a configuration file. This file specifies flags that determine what buffers the tasks must allocate and how they will checkpoint. Checkpoint buffers are allocated using the `sbrk()` system call. Conventional memory allocation routines are not used since they create bookkeeping information that must be checkpointed, and thus cannot be mixed with checkpointing buffers. Other items that may be

allocated include page tables and miscellaneous buffers for reducing and compressing pages from other tasks.

3.2.2 Checkpointing a UNIX Process

Tasks in PVM are implemented as UNIX processes. A UNIX process consists of four segments: code, data, heap, and stack. To checkpoint a process properly, the data, heap, and stack segments must be saved. The code segment can be ignored because it is a *read-only* segment and is always stored on disk. The size and addresses of the data segment remain constant throughout the execution of the process, whereas the stack and heap can grow dynamically. During a checkpoint, the task must call several different functions to compute the checkpoint. Thus before the checkpoint is started, the stack is copied into the heap. Copying it into the heap allows the process to be defined as two unchanging segments, the data and heap segments. To identify checkpoint data, each task maintains a data structure that specifies data and heap segment sizes and addresses. Also recorded in this *identifying* structure are the name of the executable and the PVM task identifier. This data structure is important as it is used in recovery routines to rebuild a failed process.

3.2.3 Simple Diskless Checkpointing

The simple diskless checkpointing algorithm saves address spaces in their entirety for each checkpoint. At the beginning of the program, the application tasks initialize themselves. This includes possibly allocating buffers to store checkpoints.

If the checkpoints are to be made by a child process, a checkpoint buffer is unnecessary. If they are to be done synchronously, one buffer is needed. Two are required when asynchronous checkpointing is used.

At the beginning of every checkpoint, the checkpoint task needs to record the sizes of all the tasks. This is done implicitly by doing a parity reduction on the *identifying* data structures. Each task initializes its data structure and then performs the reduction. The checkpoint task stores the result. In the event of a failure, a parity reduction on the remaining tasks' structures with the one stored by the checkpoint task will yield the *identifying* structure that the failed task built.

The tasks next determine the sizes of the address spaces that need to be checkpointed. The size is the sum of the data and heap segment sizes since the stack is copied into the heap. Because checkpoints are calculated in a pipelined reduction fashion, the tasks perform a reduction of the sizes of all the tasks. Tasks not responsible for reducing anything simply send their sizes. Tasks responsible for reducing address spaces read and record these sizes and send the maximum of these and its size to the task that is responsible for reducing it. The checkpoint task receives the maximum task size of the program since it receives the result of the reduction.

Reduction of the pages in the address spaces occurs next. The order of reducing pages is linear beginning from the data segment. Pages are said to be *corresponding* if their addresses have the same offset from the start of their corresponding data segments. If a task does not have a corresponding page, then it

must have sent all of its address space. Tasks know this since they know the size of all the tasks they are responsible for reducing.

Once a task has sent all the necessary pages, it performs some final activities. If the algorithm is synchronous, tasks wait for a signal from the checkpoint task. If asynchronous, each task has an extra checkpoint buffer to copy its address space without destroying the previous checkpoint, and resumes execution of the target program. If a child process is doing the checkpoint, it simply waits until needed by the parent process for recovery. After the checkpoint task finishes the reduction, it informs all the tasks that it is complete by sending them a signal. If the checkpoint algorithm is asynchronous, the tasks are free to discard the old checkpoint. If the algorithm is synchronous, each task copies its data and heap segments to its checkpoint buffer, overwriting the old checkpoint. If the algorithm uses a child process, the parent process, which has already resumed execution of the target program while the checkpoint was being calculated, kills the child process for the previous checkpoint, and the new child remains as the current checkpoint.

3.2.4 Incremental Diskless Checkpointing

The incremental diskless algorithm uses page protection hardware to track page writes in a task. During a checkpoint, only those pages that have changed since the last checkpoint need to be checkpointed. The first checkpoint is the exception since it must be checkpointed in its entirety. The simple diskless algorithm is used to do this.

Along with the checkpoint buffers, a page table is allocated by the initialization routine. Instead of copying the address space to a memory buffer after the first parity checkpoint is calculated, the data and heap segments are protected as *read-only* using the `mprotect()` system call. As the program executes, page faults occur and the process catches them. The faulting page's address is marked in the page table and if the task has a checkpoint buffer, the page is copied to it. The original page is needed to calculate the $diff_k$ page. In the case of a child process, the child will maintain the original copy through the operating system's copy-on-write feature. The protection of the faulting page is set to *read-write* and the program can continue execution.

When time arrives to take the next checkpoint, the algorithm is quite similar to the simple algorithm. The main difference is that the page tables are reduced instead of the size of a task's address space. The page table lets a reducing task know which pages correspond to pages of another task.

For a dirty page k of a task, the task first calculates $diff_k$. This is the parity of $page_k$ and buf_k . Buf_k is either stored in a checkpoint buffer or in the child checkpoint process. The parent and child communicate over a pipe. After it is calculated, the task reads any corresponding $diff_k$'s from other tasks it must also reduce. It calculates the parity of all these pages with $diff_k$ and sends the result to the task responsible for reducing it. The checkpoint task calculates the parity of all the $diff_k$'s on a copy of buf_k from the previous checkpoint. After the checkpoint is completed, the checkpoint task sends a signal to all the application tasks. The tasks finish the checkpoint in the same manner as the simple algorithm except for

copying the address space. It is write-protected instead.

3.2.5 Adding Compression

Compression is simple to add to both the simple and incremental algorithms. The compression techniques use run-length encoding and bit-mapped algorithms to compress pages quickly. It achieves good compression factors when compressing sparse $diff_k$ pages. The incremental diskless algorithm always computes these pages. Thus it is simple to compress them before they are sent to another task and uncompress them when they are received. When using compression in the simple algorithm, the previous checkpoint is used to calculate $diff_k$'s. These pages are calculated, compressed, and sent instead. When compression is used, the checkpoint task must also exclusive-or the parity checkpoint with the previous checkpoint to calculate the new checkpoint.

The run-length encoding algorithm works as follows. The page is examined at the long word (8 bytes) level. Pages on the SPARC workstations are 4096 bytes long, so it treats $diff_k$ as an array of 512 long words. It run-length encodes the zeros into bytes, and alternate the count of zeros with non-zero elements. Since the number of zeros can be any value from 0 to 512, and bytes can only hold values from 0 to 255, it has to use multiple bytes for counts greater than 254. The compressed page is a stream of data consisting of tags marking a run of zeros or a non-zero word.

The bitmap algorithm also works at the long word level. It reserves the first 64 bytes as a bitmap of the non-zero elements of $diff_k$, and then if the i -th element

is non-zero, then the i th long word following the bitmap is set to that non-zero value.

Obviously, each algorithm has a range of non-zero values for which it outperforms the other. Thus, the compression algorithm uses the following hybrid algorithm. For z being the number of non-zero values in $diff_k$:

- When $0 \leq z < 61$, use the run-length encoding algorithm.
- When $61 \leq z < 504$, use the bitmap algorithm.
- When $504 \leq z \leq 512$, do no encoding.

This yields a fast compression algorithm which always provides good behavior, and can satisfy the condition to be used successfully in reducing network latency. There are other types of compression which can yield better compression ratios, such as Huffman coding, or LZW compression. However, these algorithms work on the bit-level, and thus are much slower than the above algorithms.

3.2.6 Recovery

The `pvmckpt` host program is responsible for handling failures of either tasks or machines. PVM already has the ability to detect both of these types of failures. The host program is notified of these failures by PVM. When a machine failure is detected, the host program will add a replacement machine to the PVM out of a pool of available processors. Because of the changes made to PVM, the newly created daemon will obtain the identity of failed one. If a task was running on a machine that failed, the host program will be notified of this as well. With diskless

checkpointing, the host program notifies the checkpoint task that a failure was detected. The checkpoint task is responsible for rebuilding the failed task.

The checkpoint task performs recovery by sending a signal (SIGUSR2) to the remaining tasks and a message informing them that a failure occurred. The tasks then start a procedure much like a checkpoint. They flush all messages in transit, and do a parity reduction, not of their current address space, but with their previous checkpoint stored in memory. This will result in the identifying structure of the failed task and its address space. The checkpoint task spawns a new task and sends it the address space via a socket. The tasks that did not fail then roll back to the previous checkpoint by copying the previous checkpoint back to their address spaces. Execution proceeds as if the checkpoint was just completed. In the event that the failed processor was the checkpoint processor, the host program spawns a new checkpoint task to a new checkpoint processor. This new task will contact all tasks and inform them that it needs to recalculate the parity checkpoint. Again this procedure is much like a checkpoint. However, when complete the tasks continue where they left off since rolling back is unnecessary.

3.3 Disk-based Checkpointing Variants

Checkpointing to a disk file is simpler than computing a parity checkpoint. All that is needed for a checkpoint is a set of files, one for each task. The procedure for checkpointing a program to disk shares many aspects of the diskless algorithm. The `pvmckpt` program monitors the program as it executes and calls the `pvm_groupckpt()` function to start a checkpoint. The group server then sends

a signal to all the tasks of the group. To ensure that a consistent checkpoint is taken, all tasks involved in the checkpoint first eliminate the message state using the same algorithm used in the diskless algorithms. When the tasks have completed their checkpoint, they call `pvm_ckptack()` to return any errors that may have occurred during the checkpoint.

3.3.1 Simple Checkpointing

Simple checkpointing to disk involves writing the entire address space to a disk file every time a checkpoint is taken. After flushing the message state, the tasks copy their stack segments into the heap, and then open up a file. The file that they write to is on a central file server. File operations between the operating systems is handled by the Network File System (NFS). The network used for NFS is the standard Ethernet (10Mb). With all tasks writing to the same server, bandwidth is approximately 120 Kps. The first item written to the file is the *identifying* structure. This structure specifies the executable name, task identifier, and segment addresses and sizes. This structure is used for recovery in a similar way that the diskless algorithm uses them. Next the data and heap segments are written in their entirety. When that is completed, a task can resume execution of the target program if not already, as in the case of a child process being used to write the checkpoint.

3.3.2 Incremental Checkpointing

When incremental checkpointing is used, the memory protection hardware is used to track writes made by a task. The tasks use the simple algorithm to take an initial checkpoint. After writing the address space out to disk, the protections of data and heap segments are set to *read-only*. As the program executes, page faults occur and are caught by the task. The address of the faulting page is marked in a page table, and the page's protection is set to *read-write*. When time comes to take a next checkpoint, the tasks write to the file the page table and page contents of those pages that have changed since the last checkpoint. After writing this checkpoint, the data and heap segments are again set to *read-only*. Incremental checkpointing to disk requires that all files remain as the checkpoint. This differs in the case of the simple algorithm where a previous checkpoint can be deleted. Therefore, availability of disk space becomes more of a concern.

3.3.3 Adding Compression

When tasks need to compress their checkpoint, the same compression algorithm in the diskless algorithm is used here. In that algorithm, tasks must compute the $diff_k$ page for each page k . Therefore, tasks must retain their previous checkpoint in memory. This adds to the memory overhead of checkpointing to disk. At the time of a checkpoint, the tasks compute $diff_k$ for each page k and write the compressed version to disk. When using the simple algorithm, after writing the checkpoint file, the tasks copy the data and heap segments to their checkpoint buffer. When using the incremental algorithm, the address space is write-protected and pages are

copied to the checkpoint buffer only when a page fault occurs on their addresses. In the case of checkpointing using a child process, the child remains as the checkpoint.

3.3.4 Recovery

Failures are detected in the same manner as if the program was being checkpointed to a checkpoint processor. However, when a task or machine failure occurs, all tasks are killed. If a machine needs to be replaced, it is done so using a one from a pool of available machines. Next the host program spawns off new tasks so that the tasks read the checkpoint files and recover to the point of completing that last successful checkpoint.

3.4 Combinations of Checkpointing Variants

Considering all the various ways to checkpoint a PVM program, a summary is provided in Table 3.1. The different ways are identified by four letters: **S** and **I** for simple or incremental, **C** or **N** for compression or no compression, **D** or **P** for checkpointing to disk or a processor (i.e. diskless), and, **S**, **A**, or **F** for synchronous, asynchronous, or forked checkpointing. The exception is the label **BASE** which stands for the base case of no checkpointing.

Table 3.1: Summary of Checkpointing Variants

Label	Summary of Checkpointing Variant
BASE	Base case, no checkpointing.
SNPS	Simple, non-compressed, synchronous diskless checkpointing
SNPA	Simple, non-compressed, asynchronous diskless checkpointing
SNPF	Simple, non-compressed, forked diskless checkpointing
SCPS	Simple, compressed, synchronous diskless checkpointing
SCPA	Simple, compressed, asynchronous diskless checkpointing
SCPF	Simple, compressed, forked diskless checkpointing
INPS	Incremental, non-compressed, synchronous diskless checkpointing
INPA	Incremental, non-compressed, asynchronous diskless checkpointing
INPF	Incremental, non-compressed, forked diskless checkpointing
ICPS	Incremental, compressed, synchronous diskless checkpointing
ICPA	Incremental, compressed, asynchronous diskless checkpointing
ICPF	Incremental, compressed, forked diskless checkpointing
SNDA	Simple, non-compressed, asynchronous disk-based checkpointing
SNDF	Simple, non-compressed, forked disk-based checkpointing
SCDA	Simple, compressed, asynchronous disk-based checkpointing
SCDF	Simple, compressed, forked disk-based checkpointing
INDA	Incremental, non-compressed, synchronous disk-based checkpointing
INDF	Incremental, non-compressed, forked disk-based checkpointing
ICDA	Incremental, compressed, asynchronous disk-based checkpointing
ICDF	Incremental, compressed, forked disk-based checkpointing

Chapter 4

Experiments And Results

In order to understand the performance of diskless checkpointing, five programs were selected to be checkpointed. Each program is an example of a long-running numerical or scientific program that can benefit from checkpointing for fault-tolerance. This chapter first presents the programs and explains the experiments performed on each program. The chapter then discusses and analyzes the results.

4.1 Experiments

In checkpointing for fault-tolerance, there are three important metrics for analyzing checkpointing algorithms:

- **Checkpoint Time:** This is the average time to take a checkpoint from start to finish.
- **Checkpoint Overhead:** This is the amount of time added to the running time of the application program due to checkpointing. In synchronous

checkpointing, the overhead is equal to the total checkpoint time. With copy-on-write (`fork()`) checkpointing, the overhead is smaller than the total checkpoint time since checkpointing is performed in parallel with the execution of the application.

- **Checkpoint size:** This is the size of the checkpoint.

Of the three, checkpoint overhead is considered the most important, since it is the most visible from a user's point of view [Vai95]. If overhead is too high, a user is more likely not checkpoint an application in favor of taking the risk of a failure. Checkpoint size is important because a user must provide either the disk space or extra memory in order to checkpoint an application. Checkpoint time is a concern for it affects how many times a program can be checkpointed. The main goal in checkpoint optimization is to reduce these metrics.

The programs used for experimentation were chosen for their differences in implementation language, memory needs, locality, and network usage. The results obtained by the experiments are used both to compare the diskless and disk-based checkpointing algorithms as well as to analyze how well the diskless algorithms perform according to the performance metrics.

For each program, parameters were chosen that would be typical of an actual instance of the application and would make extensive use of the network of workstations. Each program was executed in the absence of checkpointing as a base case to measure overhead. Each program was then checkpointed using the simple diskless algorithm in the synchronous, asynchronous and copy-on-write variations. Next, the incremental and compressing optimizations were applied to see how well

they performed with the application. If possible, disk based checkpointing was done to compare how well it performs compared to diskless checkpointing. All disk checkpoints were made using the copy-on-write optimization only. Considering the sizes of the disk checkpoints, synchronous checkpointing imposes too much overhead, and thus is not considered.

4.1.1 Parallel Matrix Multiplier (MAT)

The MAT program is a parallel matrix multiplier that calculates $AB = C$ using Cannon's algorithm [KGGK94]. It is written in the C language. The algorithm uses p processors where P is a perfect square. The processors are labeled $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$. The matrices A and B are partitioned into p square blocks. Blocks $A_{i,j}$ and $B_{i,j}$ are initially assigned to processor $P_{i,j}$. To calculate $C_{i,j}$, processor $P_{i,j}$ needs to multiply the block $A_{i,k}$ with $B_{k,j}$ for $0 \leq k < \sqrt{p}$. Cannon's algorithm works by scheduling the blocks of A and B over $\sqrt{p}$ steps such that at each step the processors have the correct blocks of A and B needed to calculate C . Figure 4.1 shows the steps involved in communicating the blocks of A and B for a 16 processor system.

To begin, the blocks of A and B must be aligned such that each processor has the correct blocks to begin calculation. This requires a circular shift of the rows of A and the columns of B . Block $A_{i,j}$ is shifted to the left (with wraparound) by i steps. Block $B_{i,j}$ is shifted up (with wraparound) by j steps. Figures 4.1(a) and (b) show how the blocks are shifted and Figure 4.1(c) shows the locations of the blocks after being aligned. The first stage in the calculation of C can now

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

(a) Initial alignment of A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

(b) Initial alignment of B

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{2,2}$	$A_{2,3}$	$A_{2,0}$	$A_{2,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{3,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$

(c) A and B after initial alignment

$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{1,2}$	$A_{1,3}$	$A_{1,0}$	$A_{1,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{2,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$

(d) Submatrix locations after first shift

$A_{0,2}$	$A_{0,3}$	$A_{0,0}$	$A_{0,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{1,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$

(e) Submatrix locations after second shift

$A_{0,3}$	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{3,2}$	$A_{3,3}$	$A_{3,0}$	$A_{3,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$

(f) Submatrix locations after third shift

Figure 4.1: The communication steps in Cannon's algorithm on 16 processors.

Source: Kumar, Grama, Gupta, Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings Publishing Company, Inc., 1994, pg 172.

begin. After the blocks are multiplied, the processors communicate by sending the blocks of A to the processor on the left (with wraparound) and the blocks of B to the processors above (again with wraparound). The communication steps are shown in Figure 4.1 (c). The second stage in calculating C occurs next. This process of communicating and multiply the blocks occurs for $\sqrt{p}$ steps as shown in Figures 4.1 (d) through (f).

The matrix multiplier was executed on 16 processors, multiplying two $N \times N$ matrices where N was 4,608. The memory usage per task is approximately 15.5 megabytes. The interval for checkpointing was chosen to be 14 minutes. This allows time for seven checkpoints to be completed.

This matrix multiplier is interesting in that during communication, processors reuse the memory for their portions of matrices A and B to calculate more of matrix C . However, the interval of checkpointing is small enough that two checkpoints can be taken between the time the processors need to communicate. Thus there is the opportunity to use incremental checkpointing to improve performance.

The incremental behavior also gives reason for compression to work well. For the same checkpoints where matrices do not have to be checkpointed in the incremental case, they can be compressed away to only a few bytes to lower network latencies.

In the disk-based checkpointing algorithms, the checkpoint times are so high that only three checkpoints can be taken during the runtime. This eliminates the chance to use incremental checkpointing to improve performance since all of the memory has been overwritten. Results of MAT program runs are in Table 4.1.

Table 4.1: MAT Results

LABEL	NRuns	AvgTime (sec)	StdDev	NCkpts	AvgCkptSize per node (Mb)	AvgCkptTime (sec)	Ov (sec)	%Ov
BASE	3	6602.0	39.1					
SNPS	3	7021.0	31.2	7.0	15.5	61.3	419.0	6.3
SNPA	3	6976.3	17.5	7.0	15.5	61.0	374.3	5.7
SNPF	3	6913.7	54.4	7.0	15.5	63.0	311.7	4.7
INPS	3	6861.0	25.5	7.0	9.1	43.0	259.0	3.9
INPA	2	6841.0	4.0	7.0	9.1	41.1	239.0	3.6
INPF	3	6735.3	3.4	7.0	9.2	44.0	133.3	2.0
SCPS ¹	2	6871.5	9.5	7.0	15.5	46.0	269.5	4.1
SCPA ¹	2	6878.5	13.5	7.0	15.5	47.5	276.5	4.2
SCPF ¹	2	6788.5	0.5	7.0	15.5	60.4	186.5	2.8
ICPS ²	2	6871.5	26.5	7.0	9.1	44.7	269.5	4.1
ICPA ²	1	6850.0	0.0	7.0	9.1	42.0	248.0	3.8
ICPF ²	2	6788.5	9.5	7.0	9.1	51.4	186.5	2.8
SNDF	3	6648.7	37.3	3.0	15.5	1955.0	46.7	0.7

¹ Average factor of compression is 0.5² Average factor of compression is 0.07

For a runtime of nearly two hours, checkpoint overhead of synchronous diskless checkpointing is 6.3%. As expected, the asynchronous and copy-on-write checkpoint optimizations further reduce the overhead since each node has the physical memory to support these optimizations.

Incremental checkpointing even further reduces the overhead since some checkpoints need only save the changed portions of matrix C . Results confirm that the third, fifth, and seventh checkpoints are indeed smaller than the rest (three megabytes as opposed to 15.5 megabytes.) Adding the copy-on-write optimization

to incremental checkpointing reduces the overhead to an overall minimum of 2% for the diskless algorithms.

Compression combined with the simple algorithm is interesting that it can nearly perform as well as incremental checkpointing. Combining compression with incremental checkpointing is detrimental though, because floating point data does not compress well. Attempting to compress such pages only adds overhead since little compression can be done.

4.1.2 Cellular Automata (CELL)

The CELL program executes cellular automata over an initialized grid for a specified number of generations. The program uses two grids: one representing generation i , the other representing $i + 1$. The grid representing generation i is used to calculate generation $i + 1$, and then the identity of the grids is swapped. The grids are partitioned in column striped fashion and communication of the border information is done at each generation. After two generations all the grid memory of each task has been written. The program is written in C.

The CELL program was executed on 16 processors calculating 475 generations on an 18,512 by 18,512 grid. The size of each checkpoint was approximately 42 megabytes per task. The checkpoint interval was chosen to be 18 minutes and five checkpoints were taken for all diskless algorithms tested. Since the interval exceeds the time needed to calculate two generations, incremental checkpointing can only raise overhead due to page faulting on the entire address space after each checkpoint. However, since certain types of input data result in large portions of

the grids going unchanged from iteration to iteration, the use of the compression optimization is possible to decrease overhead. The large size of the checkpoints makes disk-based checkpointing ineffective, as only one checkpoint can be taken during the run of the program. Results of this program are in Table 4.2.

Checkpoint overhead in the simple synchronous algorithm for five checkpoints was 12.1% of the total runtime. The copy-on-write optimization lowers the overhead to 7.9%.

The asynchronous optimization worsens the overhead. This is due to paging.

Table 4.2: CELL Results

LABEL	NRuns	AvgTime (sec)	StdDev	NCkpts	AvgCkptSize per node (Mb)	AvgCkptTime (sec)	Ov (sec)	%Ov
BASE	3	6351.3	16.9					
SNPS	3	7119.7	8.5	5.0	41.5	156.2	768.3	12.1
SNPA	3	7210.0	43.2	5.0	41.5	160.4	858.7	13.5
SNPF	3	6850.3	33.3	5.0	41.5	160.1	499.0	7.9
INPS	2	7345.0	55.0	5.0	41.3	173.7	993.7	15.6
INPA	1	7547.0	0.0	5.0	41.3	179.8	1195.7	18.8
INPF	2	7075.5	42.5	5.0	41.3	234.2	724.2	11.4
SCPS ¹	3	6927.0	47.1	5.0	41.5	122.7	575.7	9.1
SCPA ¹	3	7224.0	45.3	5.0	41.5	165.5	872.7	13.7
SCPF ¹	3	6755.0	5.0	5.0	41.5	140.3	403.7	6.4
ICPS ¹	2	7025.5	6.5	5.0	41.3	123.7	674.2	10.6
ICPA ¹	2	7339.5	23.5	5.0	41.3	133.0	988.2	15.6
ICPF ¹	1	6951.0	0.0	5.0	41.3	154.2	599.7	9.4
SNDF	3	6432.3	6.9	1.0	41.4	5346.0	81.0	1.3

¹Average factor of compression is 0.99

The asynchronous variation requires two checkpoint buffers ,making the virtual memory size per task nearly 125 megabytes. Physical memory on each node is 96 megabytes, so only one checkpoint can be fully in memory at a time with the application address space. The other buffer must be partially stored in swap space. When a task must make a copy of the address space for a new checkpoint, it pages in the remaining portions of the buffer, which causes the previous checkpoint to be paged out.

The compression optimization performs its purpose well, reducing the overhead to a minimum 6.4% overall combined with the copy-on-write option. This is because the input data chosen so that large portions of the grid go unchanged from iteration to iteration. Asynchronous checkpointing with compression performs exceptionally bad though. Because the algorithm uses the previous checkpoint to compute page differences and then writes a new checkpoint to the other buffer, the application's memory gets selected for displacement. The result is that portions of the application are paged to disk, only to be paged back in after the checkpoint is complete. This negates all benefits of trying to reduce overhead using compression.

The incremental checkpointing run demonstrates that as expected, this optimization does indeed perform worse. With disk-based checkpointing, only one checkpoint can be completed during runtime which makes it ineffective as a means for fault-tolerance for this set of parameters.

4.1.3 N-Body Interactor (NBODY)

The NBODY program is a scientific program used to compute N-body interactions among particles in a system. The program, written in C, uses the parallel multipole tree algorithm [Lea92]. This application uses a host program to communicate work to a specified number of slave tasks. Each slave task stores the particles that are contained in its area. Communication between slaves is done to move particles among different areas of the system after each iteration. Depending on the initial data, slaves may differ in size due to the number of particles they must store.

The NBODY program was executed with 16 slave tasks using 15000 particles for 10 iterations. The average checkpoint size per task is under three megabytes, but since this program uses a host task and the slaves differ in size, the size of each task varies. The largest checkpoint size is six megabytes. This program does not exhibit any incremental behavior since each particle is given a new location for each iteration. However, pages are only sparsely written, which allows good compressibility. Of the five programs tested, this is the only program where the time to take a disk checkpoint is small enough so that the number of disk checkpoints can equal the number of diskless checkpoints. Results of these experiments are shown in Table 4.3.

Because of the small checkpoint sizes of this application, overheads are small for this application. This program can be checkpointed efficiently, adding less than 2.5% overhead to the run time. The simple synchronous diskless algorithm only adds 1.5% overhead. The asynchronous versions barely perform better than the synchronous versions. As predicted, incremental checkpointing performs worse

Table 4.3: NBODY Results

LABEL	NRuns	AvgTime (sec)	StdDev	NCkpts	AvgCkptSize per node (Mb)	AvgCkptTime (sec)	Ov (sec)	%Ov
BASE	3	5722.0	15.6					
SNPS	3	5805.3	4.5	6.0	3.7	18.0	83.3	1.5
SNPA	3	5810.3	4.6	6.0	3.7	17.2	88.3	1.5
SNPF	3	5780.7	4.6	6.0	3.7	16.9	58.7	1.0
INPS	3	5840.3	3.7	6.0	3.6	17.5	118.3	2.1
INPA	2	5833.0	3.0	6.0	3.6	17.5	111.0	1.9
INPF	2	5814.0	13.0	6.0	3.6	19.6	92.0	1.6
SCPS ¹	3	5774.3	11.9	6.0	3.7	15.1	52.3	0.9
SCPA ¹	2	5772.0	3.0	6.0	3.7	14.7	50.0	0.9
SCPF ¹	2	5754.5	1.5	6.0	3.7	17.0	32.5	0.6
ICPS ¹	2	5855.5	25.5	6.0	3.6	17.8	133.5	2.3
ICPA ¹	2	5852.0	16.0	6.0	3.6	19.1	130.0	2.3
ICPF ¹	2	5801.5	5.5	6.0	3.6	19.8	79.5	1.4
SNDF	3	5864.3	44.9	6.0	3.7	430.3	142.3	2.5

¹Average factor of compression is 0.79

than simple checkpointing. Using compression with copy-on-write lowers overhead to 0.6% of runtime. Most importantly, these results show diskless checkpointing can outperform disk-based checkpointing when the number of checkpoints are equal for both types of algorithms. Checkpoint times and overheads are less for the diskless algorithms.

4.1.4 Shallow Water Model (STSWM)

The STSWM program, written in FORTRAN, is a scientific application used to solve the nonlinear shallow water equations on a rotating sphere using the spectral transform method [HJW93]. This is a hostless program that models the system every 15 minutes for a specified number of hours.

The STSWM program was set up to run a simulated 102 hours on a data set that consumed approximately 25 megabytes per task. Checkpoints were taken every 14 minutes. As with the CELL program, the checkpoint times for disk based checkpoints only allow for one checkpoint to be taken. The results for the STSWM program are given in Table 4.4.

Checkpoint overhead percentage is 8.9% for the simple synchronous variation. Since two checkpoints can be maintained in memory at a time, the asynchronous and copy-on-write optimizations improved the performance, decreasing the overhead to 5.2% for the copy-on-write case.

Since most of the memory is rewritten between checkpoints, incremental checkpointing barely decreases the overhead compared to its counterparts in the simple algorithms.

Using compression shows that this program is similar to the NBODY program in that pages are only sparsely written. Checkpoint times using compression are the lowest of all variations. Using compression with copy-on-write lowers the overhead to a minimum of 4.6% for the diskless algorithms.

Combining incremental checkpointing with compression lowers the compressibility to the point where overhead cannot be reduced further.

Table 4.4: STSWM Results

LABEL	NRuns	AvgTime (sec)	StdDev	NCKpts	AvgCkptSize per node (Mb)	AvgCkptTime (sec)	Ov (sec)	%Ov
BASE	3	5610.0	22.8					
SNPS	3	6110.0	5.7	6.0	25.3	86.1	500.0	8.9
SNPA	3	6063.3	2.6	6.0	25.2	86.6	453.3	8.1
SNPF	3	5904.0	18.5	6.0	26.3	97.3	294.0	5.2
INPS	3	6096.3	28.4	6.0	19.9	75.8	486.3	8.7
INPA	2	5970.0	20.0	6.0	20.9	73.9	360.0	6.4
INPF	3	5895.0	46.0	6.0	20.9	91.6	285.0	5.1
SCPS ¹	2	5916.5	19.5	6.0	25.1	49.8	306.5	5.5
SCPA ¹	2	5917.0	39.0	6.0	25.5	49.2	307.0	5.5
SCPF ¹	2	5825.5	17.5	6.0	25.4	66.0	215.5	3.8
ICPS ²	2	5928.0	2.0	6.0	20.4	53.0	318.0	5.7
ICPA ²	2	5911.5	10.5	6.0	20.4	57.2	301.5	5.4
ICPF ²	2	5870.5	9.5	6.0	21.1	70.8	260.5	4.6
SNDF	3	5655.7	15.5	1.0	24.4	3122.7	45.7	0.8

¹ Average factor of compression is 0.89² Average factor of compression is 0.87

Like the CELL application, checkpointing to disk is not worthwhile with this application since only one checkpoint can be taken during run time.

4.1.5 Iterative equation solver (PCG)

The last program, PCG, is an iterative solver using the preconditioned conjugate gradient method. Given a sparse matrix A and vector b , the PCG program finds the vector x that solves the equation $Ax = b$. Using an initial approximation to x , the program iteratively refines the approximation until $Ax = b$ to within a speci-

fied error. The PCG program is written in FORTRAN and has a curious layout of memory. The matrix A , is stored in a dense form. This matrix is broadcast to all tasks along with b and two preconditioners M_1 and M_2 . Afterwards, none of these arrays are altered during the execution of the program. This leaves much of the memory of each task as read-only memory.

The PCG is the largest program tested. The matrix A is 1,638,400 by 1,638,400 and is solved within the given error in 3,750 iterations. The checkpoints are nearly 67 megabytes in size per task and were taken every 16 minutes. Since the program contains large quantities of read-only memory, incremental and compressed checkpoints were tested. No disk checkpoints were taken since one could not be completed during the runtime. Results of the tests appear in Table 4.5.

As expected, incremental checkpointing performs exceptionally well, improving performance over every variation of simple algorithm. The lowest checkpoint overhead is 3.9% for the copy-on-write incremental algorithm.

The synchronous and asynchronous simple algorithms did not perform well since excessive paging must occur to complete a checkpoint. It is interesting to note the asynchronous case still performed better than the synchronous case even with the exorbitant use of memory. This was not the case with the CELL program. The reason it works well here is that the PCG program must perform paging for both the synchronous and asynchronous algorithms, whereas the CELL program did not need to perform paging for the synchronous algorithm. The copy-on-write optimization performed much better due to the large quantity of shared memory between the child and parent processes.

Table 4.5: PCG Results

LABEL	NRuns	AvgTime (sec)	StdDev	NCkpts	AvgCkptSize per node (Mb)	AvgCkptTime (sec)	Ov (sec)	%Ov
BASE	3	5873.7	19.3					
SNPS	3	8011.7	34.2	6.0	66.6	322.9	2138.0	36.4
SNPA	3	7994.7	35.0	6.0	66.6	317.4	2121.0	36.1
SNPF	3	6546.7	36.4	6.0	66.6	242.0	673.0	11.5
INPS	3	6525.3	17.9	6.0	16.6	85.9	651.7	11.1
INPA	3	6323.3	18.7	6.0	16.6	84.6	449.7	7.7
INPF	3	6103.7	6.9	6.0	16.6	101.6	230.0	3.9
SCPS ¹	2	8019.0	2.0	6.0	66.6	325.4	2145.3	36.5
SCPA ¹	1	7998.0	0.0	6.0	66.6	321.8	2124.3	36.2
SCPF ¹	2	6901.0	24.0	6.0	66.6	307.0	1027.3	17.5
ICPS ²	2	6488.5	52.5	6.0	16.6	79.7	614.8	10.5
ICPA ²	2	6385.5	3.5	6.0	16.6	82.2	511.8	8.7
ICPF ²	2	6100.5	8.5	6.0	16.6	93.0	226.8	3.9

¹ Average factor of compression is 0.88² Average factor of compression is 0.28

Asynchronous and synchronous checkpointing with compression perform extremely badly, since the algorithms must page in a previous checkpoint from swap space to calculate page differences. A checkpoint cannot totally be stored in physical memory without having to page out the application memory. Again, the copy-on-write optimization performs better since shared memory relieves some of the paging.

Like the MAT program, combining incremental checkpointing with compression does not yield any further reduction in overhead since floating point data does not compress well.

4.1.6 Failures

Each application was also tested to see how well recovery performed in the event of a failure. There are two types of failures that the diskless system can tolerate. The first is a failure of a single application processor, where a task dies and must be respawned by the checkpoint task using a recovered address space. The second type is the failure of the checkpoint task. Recovery of an address space is similar to taking a checkpoint using the simple synchronous algorithm. Each program was executed using this simple synchronous algorithm and allowed to take a checkpoint. Then, an application task was killed using PVM's `kill` function and the recovery time was recorded. Next, the checkpoint task was then killed using the same method, and its average recovery time was recorded. The average recovery times for three failures are shown in Table 4.6.

Table 4.6: Recovery Results

Application Name	Recovery Time for Application Task (sec)	Recovery Time for Checkpoint Task (sec)
MAT	49.7	46.0
CELL	161.0	138.3
NBODY	17.3	15.7
STSWM	78.3	66.3
PCG	739.3	375.3

4.2 Results and Discussion

The diskless checkpointing algorithms were designed as an alternative to disk-based checkpointing for fault-tolerance. Analysis of the results can give an understanding of diskless checkpointing and how the algorithms compare to the disk-based versions.

4.2.1 Checkpoint Times

Figure 4.2 shows that the checkpoint time for the simple diskless algorithms is a linear function of the checkpoint size. The synchronous and asynchronous algorithms show a slight upward trend as the checkpoint size increases. This is due to paging since the nodes did not have sufficient memory to always store a checkpoint in physical memory.

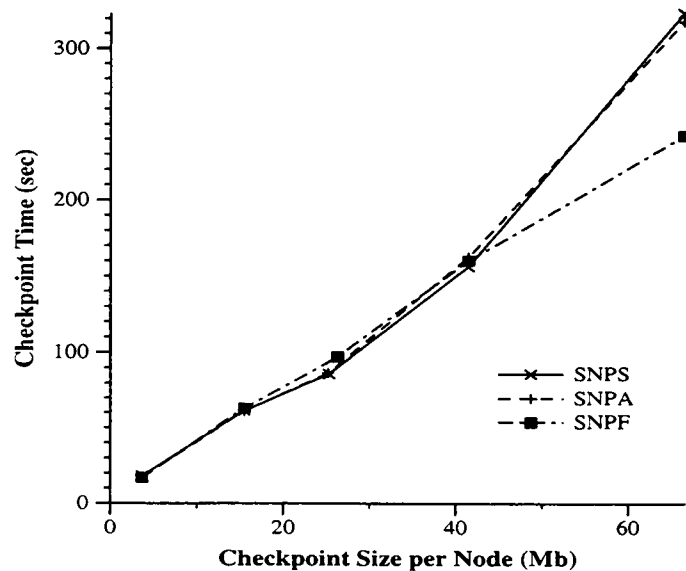


Figure 4.2: Checkpoint time versus size for simple diskless checkpointing.

Figure 4.3 shows the percentage reductions in checkpoint times for the simple synchronous diskless compared to the checkpoint time for a copy-on-write disk-based algorithm. In all cases, reductions were over 95% showing that the diskless algorithms are much faster than the disk-based algorithms.

4.2.2 Checkpointing Overhead

Although the checkpoint time to take a diskless checkpoint is an order of magnitude faster than a comparable disk-based checkpoint, a diskless checkpoint does not provide an order of magnitude less overhead. For a disk-based checkpoint using the copy-on-write optimization, the checkpoint overhead is extremely low because of the small processor needs of writing to disk. A diskless checkpoint requires extensive use of the processor to calculate parity. Therefore, a diskless checkpoint slows down an application more than a disk-based checkpoint. In Fig-

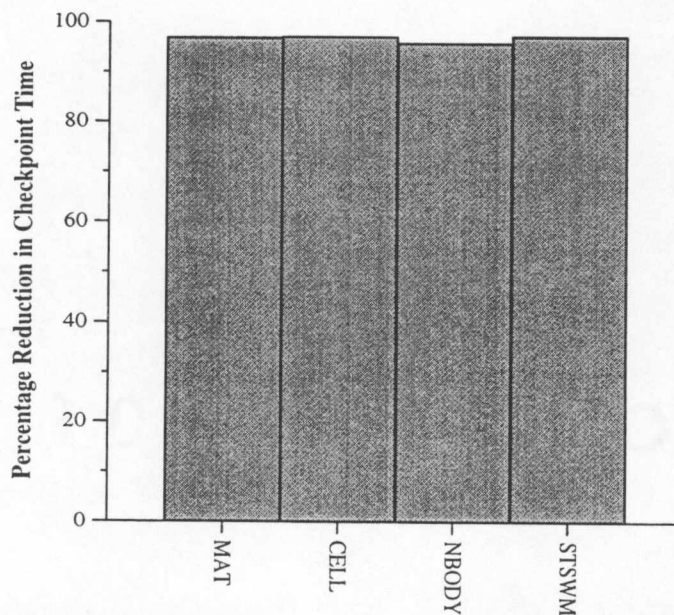


Figure 4.3: Percentage reduction in checkpoint time using diskless checkpointing.

ure 4.4, the overheads imposed by checkpointing are displayed. The times are normalized by the number of checkpoints taken and the size of the application. In Figure 4.5, the diskless checkpointing overheads are displayed by algorithms instead by programs. Figure 4.6 shows a summary of how well different optimizations performed in reducing overheads compared to other optimizations.

From Figure 4.4, it is clear that the asynchronous versions generally perform better than their synchronous counterparts, and the copy-on-write versions perform better than both. Checkpointing the CELL program asynchronously is the only case where overheads are not less than the the synchronous version, as was explained earlier with the paging required during checkpointing. This can be seen in Figure 4.6(a).

The MAT, STSWM, and PCG all exhibit good locality whereas the CELL and NBODY programs do not. Thus, the former programs show an improvement using incremental checkpointing and the latter ones do not. Figure 4.6(b) shows that using incremental checkpointing on CELL and NBODY add overhead rather than reduce it.

All programs show good compressibility when not combined with incremental checkpointing (all factors are above 0.50.) Thus it is expected that the optimization should improve performance. This is true for all programs except PCG. In that case, excessive paging required to compute the page differences negates any gain in performance. Figures 4.5(g) through (i) show that compression can nearly reduce by a third the time to checkpoint one megabyte per checkpoint when paging is not required.

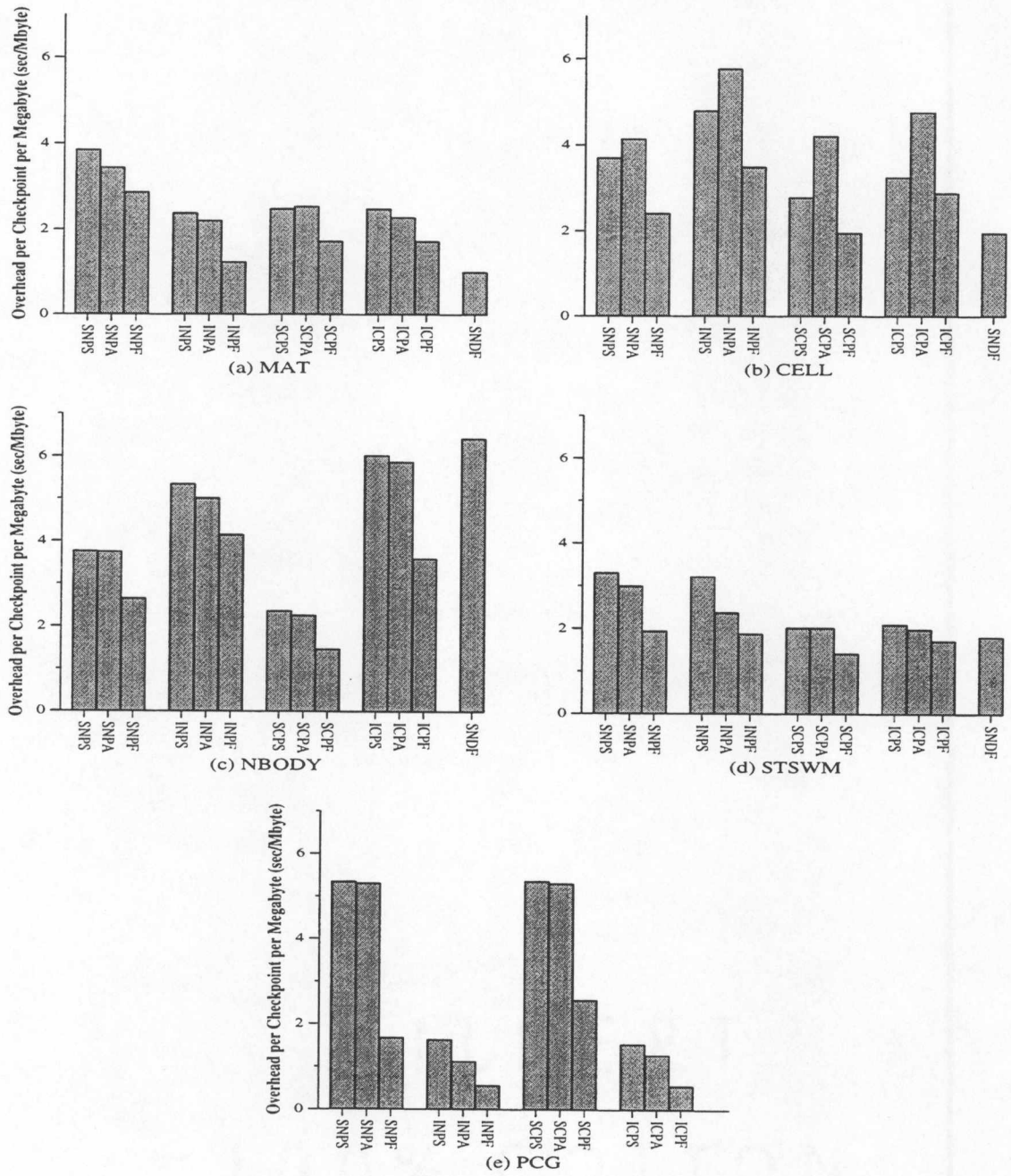


Figure 4.4: Diskless checkpointing overhead by program.

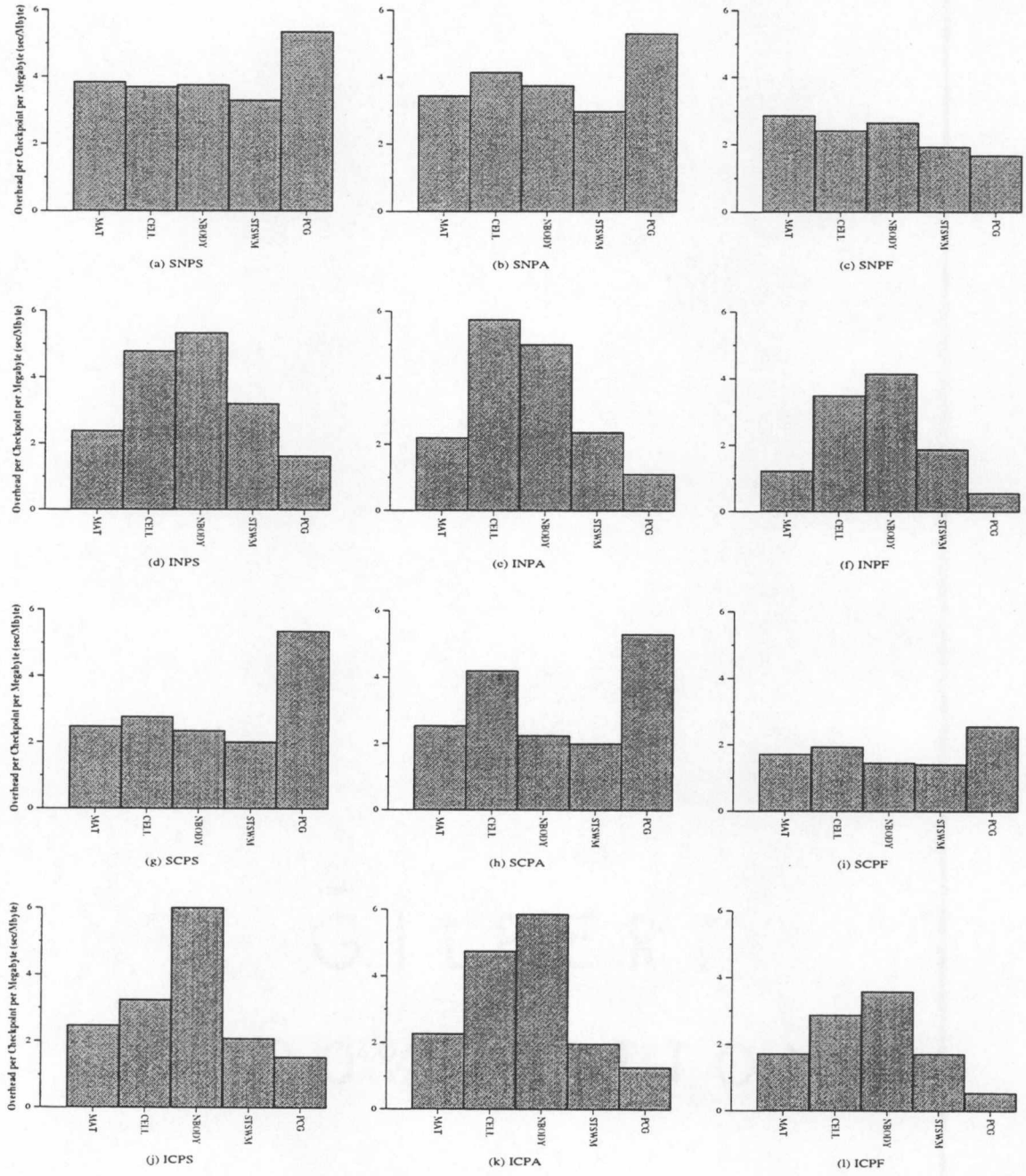


Figure 4.5: Checkpointing overhead by algorithm.

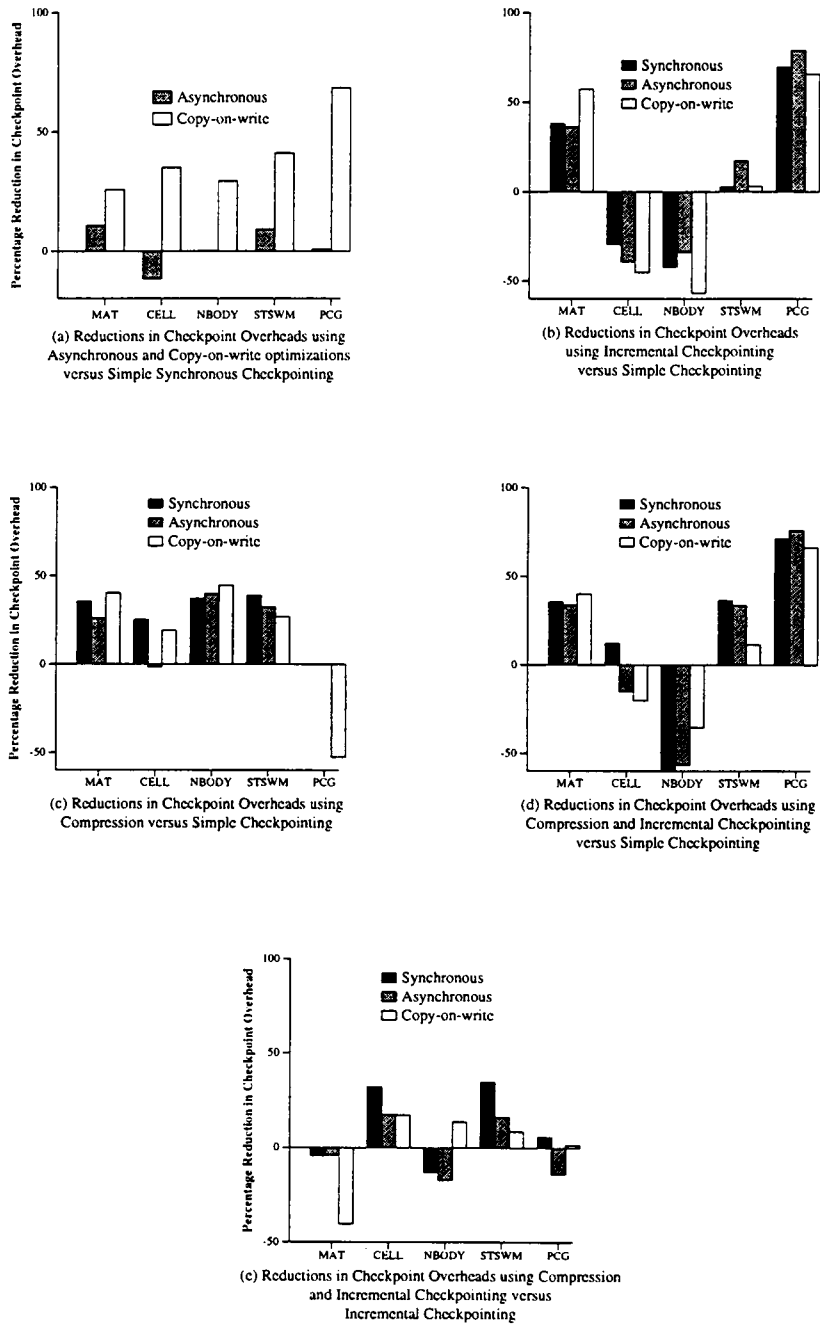


Figure 4.6: Reductions in overhead using checkpoint optimizations.

Incremental checkpointing combined with compression generally has lower overheads compared to simple checkpointing with the exception of the CELL and NBODY programs. This is seen in Figure 4.6(d). However, when compared to incremental checkpointing without compression, Figure 4.6(e) shows that compression only provides good benefits for the CELL and STSWM programs. Programs like the MAT program are left to checkpoint floating point data which requires more time to compress than that which is saved trying to compress it.

Considering all optimizations on checkpointing a program, Figure 4.4 shows that there exists for each tested application a set of optimizations that can be used to checkpoint an application using diskless checkpointing with better or nearly equal overhead per megabyte per checkpoint as checkpointing to disk using a simple checkpointing scheme. This is significant when combined with the huge savings in checkpoint time available with diskless checkpointing.

4.2.3 Recovery

Figure 4.7 graphs the recovery times for application and checkpoint task failures. The figure shows that recovery time is a linear function of the checkpoint size when there is sufficient memory to store a checkpoint in physical memory. Recovery time for the checkpoint task is less than the time to recover an application task since there is no need to spawn a new task or copy back any checkpoints.

When there is not enough memory to store a checkpoint in physical memory, each task must page in the checkpoint to rebuild the failed task's address space. This displaces the application memory. The tasks must page it back in to copy

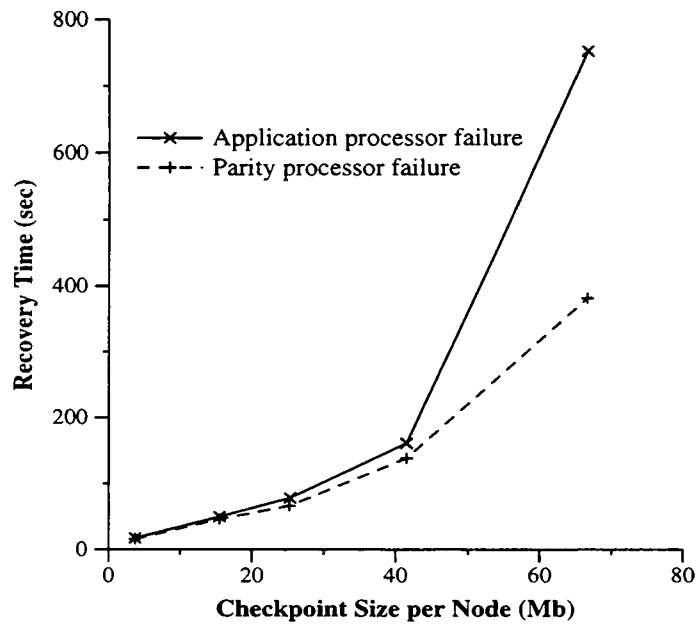


Figure 4.7: Recovery time for application and Checkpoint task failures.

the checkpoint back to it. The recovered task upon receiving its address space, must make a copy for its checkpoint. Thus it must do some paging of memory also. This excessive paging causes the upward shift in the graph.

Chapter 5

Related Work

Checkpointing is a well-researched and well-documented topic in fault-tolerance. In parallel and distributed systems, the field has been divided into three categories. They are pessimistic [BBG83, BBG⁺89, PP83], optimistic [Joh89, SY85], and consistent checkpointing [CL85, CJ91, KMBT92, KT87, LY87, LNP92, SK86]. Implementations have been tested on uniprocessors [FB89, LF90, LS92], multiprocessors [LMJ93, LNP90], distributed systems [EJZ92, KMBT92], and multi-computers [PL94b].

5.1 Previous Work

The work presented in this thesis is based on the diskless checkpointing algorithms using $N + 1$ parity presented by Plank and Li [PL94a]. $N + 1$ parity was previously used to provide single-site fault-tolerance by Gibson in his design and implementation of RAID disk arrays [Gib90]. Gibson also addresses multiple-site

failures, and gives an algorithm for tolerating 2-site failures with $2\sqrt{n}$ extra disks. This algorithm scales to tolerate m -site failures with $mn^{\frac{m-1}{m}}$ extra disks. Reed-Solomon codes were not used to reduce the number of extra disks to m because of the extra complexity that such codes require in the disk controller. This is not a problem in a diskless checkpointing system since the unit of fault-tolerance is a processor which is quite capable of handling the extra calculations [Pla96]. RAID techniques were also combined with memory management in a design of disk arrays to support transaction recovery in [MFS93].

Johnson and Zwaenepeol presented an algorithm to reconstruct the message state of a distributed system when at most one processor fails with no disk-writing [JZ87]. The algorithm has the sending processor save the messages so that it may resend them if the receiver fails. Processors still save their own execution states in disk checkpoints.

Keleher, Cox, and Zwaenepeol used *diff's* to propagate updates to shared pages in their distributed shared memory system "Munin" [KCZ92]. Plank, Xu, and Netzer relied on the fact that *diff's* are often quite compressible and showed that the latency of writing incremental checkpoints to disk can be reduced [PXN95]. The work in this thesis uses compressed *diff's* to lower latencies of sending pages to other processors.

5.2 Current Related Work

Currently there are at least three publicly available implementations of coordinated checkpointers for PVM [BDG⁺93, SGDR93] and MPI [MPIF94] based sys-

tems. They are MIST [CCG⁺95], CoCheck [Ste94, Ste96], and Fail-Safe PVM [LFS93]. Each of these projects' goal is to develop a system that provides transparent coordinated checkpointing for task migration, coarse-grained job swapping and wholesale fault-tolerance. In each system, checkpoints are made to a disk on a central file server. The checkpoint times in these systems can be very high for large programs. The only optimization used to lower checkpoint overhead in these systems is the copy-on-write technique. There are no techniques used to lower checkpoint times.

Plank presented algorithms to improve the performance of coordinated disk-based checkpointers on networks of workstations in [Pla96]. The algorithms use both local and central disks with checkpoint mirroring and RAID techniques to greatly reduce the latency of taking a disk-based checkpoint and still provide good fault coverage.

Plank, Kim, and Dongarra used diskless checkpointing to develop algorithm-based fault-tolerant matrix operations [PKD97]. An implementation showed that diskless checkpointing is quite feasible for fault-tolerance offering excellent checkpoint times and low overhead.

For SIMD applications, Chiueh and Deng presented diskless checkpointing algorithms for massively parallel machines using mirroring and parity techniques [CD96]. Diskless checkpointing is necessary in this case since processing elements do not have access to a disk.

Chapter 6

Conclusion and Future Work

As discussed in the first chapter, parallel applications running on networks of workstations face many obstacles that may prevent completion of a program. Checkpointing provides applications a means to tolerate failures. However, in parallel systems, traditional checkpointing using disk-based algorithms causes the disk to become a large bottleneck that prevents efficient checkpointing. Presented in Chapter 2 are algorithms for diskless checkpointing which do not rely on stable storage. Details of an implementation is given in Chapter 3, and results of checkpointing five applications are given in Chapter 4. These results provide an understanding of how well the diskless algorithms perform as well as allowing a comparison to be made between the disk-based and diskless algorithms.

6.1 Diskless versus Disk-based Checkpointing

Since the diskless algorithms shift the load of taking a checkpoint from the disk to the processors, the checkpoint times for a diskless checkpoint is an order of magnitude faster than a disk based checkpoint. This is a big benefit for users, for it allows checkpoints to be taken much more frequently. Should a processor fail, the work lost in a diskless checkpoint system would be far less than in a disk-based system. Of the five applications tested, only one, the NBODY program, could be checkpointed to disk as many times as it could be checkpointed disklessly. Two programs, STSWM and CELL, could only be checkpointed once during execution. In these two cases, checkpointing is not worthwhile at all since the checkpoint is a snapshot of the program at the beginning of execution. The PCG program could not even be checkpointed once during its run time.

Checkpointing more frequently has other benefits as well. All programs exhibit certain behaviors. Techniques such as compression and incremental checkpointing have been developed to take advantage of programs that have characteristics that can be exploited to reduce checkpoint overhead. In the example of the matrix multiplier program, checkpointing more frequently allows incremental checkpointing to be employed successfully to lower overhead in an otherwise non-incremental program. The disk-based algorithms are too slow to take advantage.

Despite the fact that a diskless checkpoint can be completed quickly, the algorithms compete for processor and memory resources with the application program. Disk-based algorithms do not need to compete for these resources. As checkpoint overheads show, a checkpoint taken to disk can be made in a manner that barely

impacts the application. Nevertheless, using various combinations of optimizations, all tested applications were able to be checkpointed disklessly with nearly equal or less overhead per checkpoint per megabyte than the disk-based algorithm used. This demonstrates that diskless checkpointing is indeed a viable option to provide efficient checkpointing with low overhead.

There are two concerns that might persuade one from using diskless checkpointing. One is the fact that a diskless checkpoint cannot survive wholesale failures such as power failures. But any checkpointing system can combine disk-based and diskless algorithms to cover the rare case of wholesale failures. The other concern is memory overhead imposed by the diskless algorithms. To checkpoint an application efficiently, each node must be able to store a checkpoint in physical memory. Should a user need to allocate all of physical memory to an application, checkpointing will perform badly if optimizations cannot make up for the lack of physical memory needed for a checkpoint buffer. The PCG program is an example of a program that allocates a large portion of memory for application use. Were it not for the fact that it could be incrementally checkpointed efficiently, the overheads for the simple algorithms might deter one from trying to checkpoint this program disklessly. However, as memory usage for applications increase, extremely large checkpoint times for disk-based algorithms makes the diskless algorithms more attractive.

6.2 Future Work

There is still much work that can be done to understand diskless checkpointing more fully. The implementation of the algorithms in this thesis is $N+1$ parity, and can tolerate single-site failures. Other diskless algorithms such as *Reed-Solomon* coding can be implemented to see how much performance is impacted by being able to recover from two, three or possibly more simultaneous failures.

Plank, in [Pla96], presents an algorithm to improve performance of disk-based checkpointers that uses the same disk-based algorithm in this thesis. The algorithms use RAID techniques and local disks to relieve the bottleneck of the disk on the central file server. Fault coverage of these algorithms is similar to the diskless algorithms presented in this thesis. There are several important differences though. Since checkpoints are stored on disk, there is no memory overhead imposed on the application. However, the algorithms must write a checkpoint to disk as well as calculate a parity checkpoint on another disk. It would be interesting to compare the algorithms on applications where memory overhead is a concern.

Finally, there are further optimizations that can be applied to make checkpoint times even smaller. Plank, Beck, Kingsley, and Li show that user-directed checkpointing can greatly improve the performance in uniprocessor checkpointers in [PBKL95]. The techniques they use are memory exclusion and synchronous checkpointing. Most coordinated checkpointers use asynchronous signals to notify tasks that a checkpoint must be taken as is done in this thesis. If users can modify their applications, barriers in the code can be placed in such a way that portions of the memory may not have to be checkpointed. This may lower overhead.

Bibliography

Bibliography

- [AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice/Hall International, Inc., Englewood Cliffs, New Jersey, 1981.
- [BBBM94] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual International Symposium on Computer Architecture*, pages 245–254, Chicago, IL, April 1994.
- [BBG83] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, GA, October 1983.
- [BBG⁺89] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.
- [BDG⁺93] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. Heterogeneous network computing. In *Sixth SIAM Conference on Parallel Processing*, 1993.
- [Bir86] K. P. Birman. ISIS: A system for fault-tolerant distributed computing. Technical Report TR 86-744, Cornell University, April 1986.
- [BM93] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, Toulouse, France, June 1993.
- [CCG⁺95] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM User's Group Meeting*, Pittsburgh, PA, May 1995.
- [CD96] Tzi-cker Chiueh, Peitao Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *26th International Symposium on Fault-Tolerant Computing*, June, 1996

- [CJ91] F. Cristian and F. Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.
- [CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):3–75, February 1985.
- [EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.
- [EZ92] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers Special Issue on Fault-Tolerant Computing*, 41(5), May 1992.
- [FB89] 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, Jan 1989.
- [GHK⁺89] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, April 1989.
- [Gib90] Garth Alan Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California, Berkeley, December 1990.
- [HJW93] J.J. Hack, R. Jakob, and D.L. Williamson. Solutions to the shallow water test using the spectral transform method. Technical Report TN-338-STR, National Center for Atmospheric Research, Boulder, CO, 1993.
- [Joh89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.
- [JZ87] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *17th International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

- [KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.
- [KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis. *Introduction to Parallel Computing* Benjamin/Cummings Publishing Company, Inc., 1994, pg 171-3.
- [KMBT92] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.
- [KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.
- [Lea92] James F. Leathrum, Jr. *Parallelization of the fast multiple algorithm: algorithm and architecture design*. PhD thesis, Duke University, 1992.
- [LF90] C-C. J. Li and W. K. Fuchs. CATCH – Compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.
- [LMJ93] L. A. Laranjeira, M. Malek, and R. M. Jenevein. Space/time overhead analysis and experiments with techniques for fault tolerance. *Dependable Computing and Fault-Tolerant Systems*, 8(3):303–318, 1993.
- [LFS93] J. León, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A portable package for distributed programming with transparent recovery. Rech. Rep. CMU-CS-93-124, Carnegie Mellon Univ., February 1993.
- [LNP90] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.
- [LNP92] K. Li, J. F. Naughton, and J. S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Processing*, 20(3):159–180, June 1992.
- [LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

- [LS92] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Conference Proceedings, Usenix Winter 1992 Technical Conference*, pages 283–290, San Francisco, CA, January 1992.
- [LY87] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.
- [MPIF94] Message Passing Interface Forum. MPI: A message-passing interface standard. *Int. Journal of Supercomputing App.*, 8(3/4), 1994
- [MFS93] A. N. Mourad, W. K. Fuchs, and D. G. Saab. Database recovery using redundant disk arrays. *Journal of Parallel and Distributed Computing*, 17:75–89, January 1993.
- [MS77] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.
- [Pla96] James S. Plank. Improving the performance of coordinated checkpoints on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, October 1996.
- [PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under unix. In *Conference Proceedings, Usenix Winter 1995 Technical Conference*, pages 213–223, January 1995.
- [PFL94] J. S. Plank, J. Friedman, and K. Li. A failure correction technique for parallel storage devices with minimal device overhead. Technical Report CS-94-243, University of Tennessee, August 1994.
- [PKD97] James S. Plank, Youngbae Kim, Jack J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. To appear in *Journal of Parallel and Distributed Computing*, June, 1997.
- [PL89] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.
- [PL94a] J. S. Plank and K. Li. Faster Checkpointing with $N + 1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

- [PL94b] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.
- [PXN95] James S. Plank, Jian Xu, Robert H. B. Netzer. Compressed Differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August, 1995.
- [PP83] M. L. Powell and D. L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100–109, October 1983.
- [SGDR93] V.S. Sunderam, G.A. Geist, J.J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Journal of Parallel and Distributed Computing*, 1993.
- [SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of The Sixth International Conference on Distributed Computing Systems*, pages 382–388, Cambridge, Massachusetts, May 1986. IEEE Computer Society.
- [Ste94] G. Stellner. Consistent checkpoints of PVM applications. In *First Eur. PVM User Group Meeting*, Rome 1994.
- [Ste96] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th Int. Par. Proc. Symp.*, April 1996.
- [SY85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
- [Vai95] N.H. Vaidya. On checkpoint latency. *Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, December, 1995.
- [WM89] P. R. Wilson and T. G. Moher. Demonic memory for process histories. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989.

Vita

Michael A. Puening was born in Cincinnati, Ohio on November 14, 1970. He attended public schools in the Forest Hills School District, and graduated from Turpin High School in June, 1989. In the fall of the same year, he entered Northern Kentucky University, where in May of 1993, he received his Bachelor of Science Degree in Computer Science. He entered the Computer Science Master's program at The University of Tennessee in Knoxville in August of 1993. His Master of Science Degree was received in May of 1997. He plans to return to the Cincinnati area after graduation.