Checkpoint placement strategies for fault tolerance on networks of workstations

Darryl V. Pace

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.
To the Graduate Council:

I am submitting herewith a thesis written by Darryl V. Pace entitled "Checkpoint placement strategies for fault tolerance on networks of workstations." I have examined the final electronic 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:

Brad Vader Zanden

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)
To the Graduate Council:

I am submitting herewith a thesis written by Darryl V. Pace entitled Checkpoint Placement Strategies for Fault Tolerance on Networks of Workstations. 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.

Dr. James S. Plank, Major Professor

We have read this thesis and recommend its acceptance:

Brad Vander Linden

Accepted for the Council:

Lawrence

Associate Vice Chancellor
and Dean of the Graduate School
Checkpoint Placement Strategies for
Fault Tolerance on Networks of
Workstations

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Darryl V. Pace
December 1997
Abstract

Checkpointing is a functionality that enables users of distributed systems to perform job swapping, process migration and fault-tolerance. While checkpointers typically provide job swapping and process migration with reasonable overhead, the overhead for fault-tolerance is often too high. The reason for this is not inherent in the act of checkpointing, but instead stems from how the checkpoints are placed on stable storage.

This thesis explores two placement strategies for checkpointing in distributed systems. These are called Single Processor Fault Tolerance, and Reed-Solomon coding. Both strategies are adaptations of RAID techniques [16, 41] for checkpointing systems, and aim to improve performance at the expense of fault coverage. We detail an implementation of these strategies in MIST, a checkpointer for PVM, and present performance results of these and standard checkpoint placement strategies. The conclusions that we draw are that both strategies can improve the performance of checkpointing, and should be employed by users who desire improved performance over wholesale failure coverage.
## Contents

1 Introduction .......................... 1
   1.1 Statement of purpose ............... 3

2 Coordinated Checkpointing on Networks of Workstations 4
   2.1 Transparent Checkpointing of Uniprocessors ............ 5
      2.1.1 Transparent Checkpointing ...................... 8
   2.2 Checkpointing on Networks of Workstations ............ 8
   2.3 Checkpointers ............................ 10
      2.3.1 MIST ................................. 12

3 RAID Techniques for Checkpointing 19
   3.1 What RAID is ............................ 19
   3.2 Trading Error Coverage for Performance .................. 21
   3.3 The Single Processor Fault Tolerance Algorithm ............ 22
   3.4 The Reed-Solomon Algorithm ...................... 24
3.4.1 The Vandermonde matrix ........................................... 27
3.4.2 Gaussian Elimination in RS recovery ........................... 28
3.4.3 Galois Fields ......................................................... 30
3.4.4 RS Algorithm Summary ............................................ 31

4 Implementation and Performance ....................................... 33

4.1 Machine Configuration ................................................ 33
4.2 SPFT Implementation .................................................. 34
  4.2.1 Implementation of CGJS in mpvm .............................. 34
  4.2.2 The Parity Process ............................................... 35
  4.2.3 Parity Checkpoint Creation .................................... 36
  4.2.4 SPFT Recovery .................................................... 37
4.3 RS Implementation .................................................... 38
  4.3.1 RS Checksum Creation ......................................... 39
  4.3.2 RS Recovery ..................................................... 40
4.4 Experiments ............................................................ 41
  4.4.1 The CELL Program ............................................... 43
  4.4.2 Shallow Water Model (PSTSWM) .............................. 45
  4.4.3 Discussion of Results .......................................... 45

5 Related Work and Summary .............................................. 51
List of Tables

4.1 Test results of CELL with 13 megabyte checkpoints . . . . . . . 43
4.2 Test results of CELL with 7.3 megabyte checkpoints. . . . . . . . 44
4.3 Test results of CELL with 7.3 megabyte checkpoints sent to /tmp. 44
4.4 Test results of PSTSWM with 14 megabyte checkpoints. . . . . . . 45
List of Figures

2.1 Logical representation of a process in memory. . . . . . . . . . . 6

4.1 Percentage reduction in ckpt time, per application, over FT-ckpting 47

4.2 Percentage reduction in ckpt overhead for each program we tested. 48
Chapter 1

Introduction

In today's scientific computing environment, applications that require a wall clock time of hours, days, or longer to complete, are commonplace. Often, for the increased computing power, these programs are executed on parallel computing systems. As networks of computers are becoming the platform of choice for performing parallel computations, they are frequently used to run these applications. Because the chance of processor failure increases with the number of processors incorporated in a networked machine, fault-tolerance is both desirable and needed.

Checkpointing is the only means of providing fault-tolerance in a general-purpose computing environment [1]. It is also the method of choice for providing fault-tolerance in parallel and distributed systems. Checkpointing is the act of saving, at some set interval of time, the address space of each processor involved in a running application. After a processor failure, the machine state can be
restored to that of the most recent checkpoint. Thus, if a checkpoint is taken once per hour, a long running program will lose at most an hour of computation in the event of a system failure. This is preferable to restarting the program from scratch after each failure.

Currently, distributed checkpointing systems [4, 10, 11, 24, 34, 40] perform checkpointing by writing the address space of each involved processor to a centrally located disk. Thus if any or all of the processors fail, they may be replaced by any available processors, which set their states from the checkpoints stored on the central disk. Because the act of checkpointing sends multiple checkpoints across the supporting network, significant contention for network resources often results. This can lead to one of the main deterrents to using checkpointing: checkpoint overhead. Checkpoint overhead is the amount of time that is added to an application by the incorporation of checkpointing.

There have been several proposed solutions to the overhead problem. These include incremental checkpointing [9, 24, 45], compiler support [26], pre-copying [24], non-volatile RAM [11], copy-on-write [38], and compression [26, 40]. These methods succeed in varying degrees, but all default to the speed of storing the checkpoints as their bottleneck.

The idea explored in this thesis is that improved overhead results when the requirements for fault-tolerance are reduced. Reduction in those requirements can be attained by providing for single, double, or triple processor failures using
two checkpoint placement strategies: Single Processor Fault Tolerance and Reed-Solomon coding.

1.1 Statement of purpose

This thesis shows the results of the implementations of two checkpoint placement strategies — Single Processor Fault Tolerance (SPFT) and Reed-Solomon coding — on a PVM-based network of Sun workstations. The performance of these algorithms is compared with that of standard checkpoint placement strategies. The goal of this thesis is to explain the SPFT and Reed-Solomon algorithms fully, and to show that they may be used to trade overhead for failure coverage and improve the performance of checkpointing on networks of workstations.
Chapter 2

Coordinated Checkpointing on Networks of Workstations

Checkpointing was first employed on single processor machines, and later became available in distributed environments. This chapter describes the process of checkpointing as it is implemented on a uniprocessor, followed by a discussion of checkpointing on a network of workstations. Lastly, we talk about a checkpointer for distributed systems, MIST, and a derivative of MIST, called mpvm, which is the checkpointer we used.
2.1 Transparent Checkpointing of Uniprocessors

Checkpointing on a uniprocessor is the process of saving the state of a running program to file on a disk. The goal of checkpointing is to establish a recovery point in the execution of the program, and to save enough state that in the event of a machine failure, the program can be restored back to that recovery point.

In order to fully understand the mechanics of checkpointing and recovery, one needs a clear picture of what is meant by the “state” of a process. On a Unix system, the address space of every process is broken up into segments, and it is from these segments that the state is created. Each process contains a text segment, a data segment, a stack segment, system state information, and processor state information (see Figure 2.1 for a logical picture of a process’ address space). These process components are defined, and their role in the composition of a checkpoint is briefly discussed below.

- The **text** contains the program’s machine instructions. Statically linked programs are instantiated with their entire text loaded into virtual memory by the kernel. Since the same executable file is used for both the original invocation of the program as for a restart of the program, there is nothing that needs to be done in order to save and restore this segment. It is not, therefore, saved during a checkpoint.

- The **data** contains initialized global variables, uninitialized global variables,
and the heap. Once a process begins execution, initialized data may be modified, uninitialized variables can be given values, and the heap can grow. As a result, the information in this segment may change from its initial state at startup. Thus the entire data segment is written to the checkpoint file at checkpoint time.

- The stack is used to hold information needed by the function call mechanism, function call arguments, and automatic variables. This segment, like the data segment, is modified during the execution of the program and hence is saved as part of the checkpoint.

- The system information includes operating system (OS) data structures
that can be restored, plus the state of open files. It too is saved in the checkpoint.

- The **processor state information** includes processor registers (e.g., the stack pointer, frame pointer, program counter and general purpose registers). This data changes throughout the execution of a program, and is therefore saved in the checkpoint.

In summary, the saved state of a program on a uniprocessor consists of the data and stack portions of the program’s virtual address space, the system information, and the processor state. Taking a checkpoint on a uniprocessor is straightforward. The processor state is saved using the Unix subroutine `setjmp()`, and the OS state is recorded. Lastly, the stack and data segments are written to disk.

Recovery is a bit more complex, and involves the following four steps: process creation, data state restoration, system state restoration, and processor state restoration. Process creation consists of restoring the text portion of the process and beginning execution. Data state recovery requires the reading of the checkpoint file to restore the data and stack segments of the process. System state restoration is the restoration of the OS, as much as is possible, to its state at the time of the checkpoint. Lastly, processor state restoration entails restoring the program and stack pointers to their values at the time of the checkpoint. This is effected with Unix’s `longjmp()` command.
2.1.1 Transparent Checkpointing

Checkpointing is said to be transparent when the application program does not have to be modified in order for the checkpointing to take place. Transparent checkpointing is easily implemented at the kernel level, but is much more difficult at the user level. Typically, checkpointers achieve transparency by requiring that the application program be linked with a checkpointing library, and by controlling the checkpointing process through initialization files, command line arguments, or external processes. The checkpointer we used (mpvm, which will be discussed later) is not completely transparent. In fortran programs, it requires changing each program into a specially-named subroutine. This is effected by replacing the reserved word PROGRAM in the fortran programs with the word SUBROUTINE. In C programs, the reserved word main must be replaced with Main. These changes are necessary in order for the checkpointer to be able to take over the execution of the program when it first begins.

2.2 Checkpointing on Networks of Workstations

A network of workstations is composed of a collection of possibly heterogeneous computers, each with its own processor. These computers are joined by a network that could also be made up of heterogeneous parts. Target programs that run on such a "virtual machine" are divided into multiple tasks, one on each processor.
An example of such a machine is that created by using the PVM system [7, 20]. PVM allows a collection of heterogeneous machines connected by a network to act as a single parallel computer. It does this by having a daemon reside on each processor in the virtual machine. These daemons provide rich message passing capabilities along with other functions. The computational model this type of system represents is that of distinct processing elements that communicate via message passing. Checkpointing on such a system involves saving unprocessed messages to a log, and then having each processor checkpoint its state (as detailed above for uniprocessors). The difficulties in checkpointing on a network of workstations lie in deciding what to include in the message log, and defining the points at which each processor should checkpoint. Coordinated checkpointing is the tool used to address these difficulties.

Coordinated checkpointing has a rich history in the fault-tolerance research arena [23]. It is considered an attractive checkpointing method because it is simple, it never requires multiple rollbacks, and it can handle nondeterminism in the program. The aim of coordinated checkpointing is to obtain a consistent checkpoint for a collection of processors. The checkpoint consists of the saved state of each of the processors, and a log of messages. The process of recovery involves each processor recovering from its local checkpoint, and replaying the messages.

Although several algorithms have been developed for coordinated checkpointing [5, 18, 37, 39, 43, 55], empirical evidence has shown that the major source of
overhead in all of these algorithms is saving the processor states, and thus the
choice of coordinated checkpoint algorithm used is largely immaterial [24, 25, 40].
Some of the algorithms used are the Chandy-Lamport (CL) algorithm, the Network
Sweeping (NS) algorithm, and the Sync-and-stop (SNS) algorithm.

The algorithm used most often in coordinated checkpointing is the SNS algorithm. This algorithm is popular because of its simplicity. In the SNS algorithm, the processors in the virtual machine are frozen until all outstanding messages reach their destinations, after which the processors checkpoint and resume their activities. In this way, the SNS algorithm eliminates the need for a message log. The algorithm is easy to implement on most message-passing systems, and its performance is, for the most part, on par with that of more complex algorithms [40].

2.3 Checkpointers

There have been three coordinated checkpointers either released as public domain, or slated for release since 1993. These are Fail-Safe PVM [28], MIST [13], and CoCheck [51]. Each of these checkpointers has been implemented on PVM or MPI [6], and each is designed to run on networks of Unix workstations with no modification to the OS of the workstations. All of these checkpointers are nearly or completely transparent.
These coordinated checkpointers are generally used for three purposes: transparent coordinated checkpointing for process migration, coarse-grained job swapping (CGJS), and fault-tolerance. These are defined below.

- **Process Migration** — In process migration, a processor that is either too heavily loaded, or that is needed by its owner, checkpoints itself to a spare processor. That spare then resumes computation where the no longer usable processor left off.

- **CGJS** — This involves each processor checkpointing to a local disk, after which the parallel program is stopped and resumed on the same processors at a later time.

- **Fault-Tolerance** — This entails all processors checkpointing to a central disk. If one or more of the processors fails, the computation is halted, additional processors are added, and then the system is restarted using the checkpoints that were stored on the central file.

The checkpointer we used is one called mpvm, a derivative of the MIST checkpointer. This section first describes MIST — how it works, and the features it provides — and ends with a brief description of mpvm.
2.3.1 MIST

The Migration and Integrated Scheduling Tools (MIST) project is currently under development in the Department of Computer Science and Engineering at the Oregon Graduate Institute of Science and Technology. It is an enhancement of PVM, and aims to provide several functionalities. These are:

- intelligent resource management.

- tools for debugging, profiling, and monitoring parallel programming.

- tools for monitoring system utilization, resource availability, and network traffic.

The impetus behind the project was the unpredictability of shared networked computing environments. This unpredictability is a result of users running jobs of varying computing requirements, or machine owners allowing or disallowing the use of their machines. Also, machines being shut down for maintenance, or simply failing, affects the predictability of a network of computers. The variance caused by these activities often greatly affects the performance of applications running on the computers at the time, and in the case of machine shutdown or failure, results in their termination. Terminated applications result in the loss of all the results and time used up to the point of failure, and they have to be restarted from the beginning.
For these reasons, the MIST project added task migration and checkpoint/restart functionality to PVM. Task migration allows users to migrate from one machine to another as the machine owner or level of available machine resources dictates. Checkpoint/restart adds fault-tolerance and the ability to perform coarse-grained job swapping to PVM.

**How MIST works**

Central to the functioning of the MIST project are the MIST kernel, the scheduler, and the system load monitor components. In the following paragraphs, we detail each.

**The MIST Kernel** MMPVM [7] (Multi-user, Migratable PVM), is used for the MIST kernel. It is an enhanced version of PVM that supports transparent task migration, application checkpointing/restart, and multi-user application execution. MMPVM is implemented entirely at user level, using standard Unix system calls and libraries. Because it’s not necessary to modify the Unix OS, MMPVM is easily portable to the various Unix operating systems.

In our research, MIST’s checkpoint/restart ability was our main concern. However, because MIST uses its process migration mechanisms to accomplish checkpointing, we explain those mechanisms here. Process migration is the ability to suspend the execution of a process on one machine and have it resume execution.
on another. This is done by saving the state (as described previously) of a process on one host and sending that state to another machine. Once the state has reached the second host, it is used to resume execution of the suspended process on that machine.

In order to ensure the correct functioning of the PVM task after a migration, MIST uses task ID virtualization, message forwarding, and message sequencing. Through task ID virtualization, tasks can communicate with each other regardless of which host they are on. Task ID virtualization is achieved by MIST maintaining a task-to-host mapping that is updated whenever task migration takes place. Message forwarding is used to guarantee the delivery of messages. The possibility that a message may be sent to a task that migrates before the message reaches it makes message forwarding a necessity. Message forwarding, however, introduces another problem: multiple messages for a task may take different routes, causing the messages to reach the task out of order. As this situation is a violation of PVM message passing semantics, MIST uses a message sequencing mechanism to make sure that this does not take place. This mechanism is necessary only when a message is larger than the maximum transmission unit (MTU) and must be broken up. Message sequencing is accomplished by assigning a unique sequence number to each message fragment that is sent from one task to another. This way, the message can be rearranged in its correct order once all fragments of the message reach their destination.
The user-level implementation of MMPVM, as stated before, brings about ease of portability to different flavors of Unix. However, it also results in less than total transparency for task migration. For example, process IDs cannot be guaranteed to be the same on target hosts. They are not saved in the process state as a result. Other information (e.g., information about open files) is saved and transferred instead. This data can be used during restart to re-open the files and restore them to their previous state.

MIST's checkpointing and restart functionalities use the same mechanisms employed in task migration. The difference between the two is that in migration, the state is sent by the migrating task through a socket, whereas in checkpointing the state is stored on a disk. Upon restart, a skeleton process reads its stack and data from the disk.

**The MIST Scheduler** The MIST Global Scheduler (GS) is an enhanced version of the resource manager interface provided by PVM. It is responsible for determining which machines will give the best performance for which applications. The GS also decides what machines are best for the initial placement of application tasks, when and where to migrate a task, and when the checkpoints should be taken. The GS is not relevant to the discussion of our research and we therefore will not go into its details. A description of the implementation of the GS can be found in [13].
The MIST Load Monitor  Lastly, the system Load Monitor (LM) is implemented with a simple interface that allows users to toggle the availability of machines. The LM provides machine status information to the scheduler. It, like the scheduler, was not involved in any fashion in our research, and thus will not be discussed here. See [13] for details.

As was stated above, MIST uses MMPVM for its kernel, and it is through MMPVM that MIST gets its transparent process migration and checkpoint/restart capabilities. Our research centers around improving the overhead of CGJS and FT-checkpointing. These two functionalities are provided by MIST by way of its checkpoint/restart ability. Because of this, we will concentrate on MIST’s checkpoint/restart functionality for the remainder of our discussion on MIST.

To implement CGJS in MIST, each processor has its checkpoint sent to local disk. For fault-tolerance, the process checkpoints are sent to a central disk. MIST performs its checkpointing synchronously. In other words, all the tasks in the PVM application must stop executing and agree to checkpoint before a checkpoint can take place. This is the aforementioned sync-and-stop checkpointing method. Restart from a checkpoint is effected via a synchronization similar to that which occurs when a checkpoint is taken. In this synchronization, all participating tasks must agree that everyone has successfully restarted before continuing the process execution.

The cost of the synchronization involved in a MIST checkpoint/restart is non-
trivial, but the major cost in MIST's checkpointing is a result of saving the process states to disk [13]. MIST attempts to minimize the impact of this cost by allowing the PVM process to continue executing while a checkpoint is being taken. This is done by having the tasks execute a fork system call, after which the parent continues executing while the child saves its state to disk. This type of checkpointing is called "copy-on-write" [38] checkpointing, and will be referred to as such henceforth.

MPVM

mpvm is a derivative of the MIST checkpointer. As has been stated, MIST is still being developed. This being the case, it is still "buggy". This is why mpvm was developed. mpvm is a "bare-bones" version of MIST, i.e., it gets its design from, and contains code from the MIST checkpointer, but contains none of the MIST "amenities", such as the MIST scheduler and the system load monitor interface. mpvm is a checkpointing version of PVM 3.3.8, written by Michael Puening of the University of Tennessee [50]. It provides traditional checkpointing functionality and works as follows.

Applications running under mpvm are started by a host program called pvm-ckpt. This function detects failures and starts periodic checkpoints. It receives its directives (e.g., whether or not to checkpoint, how often to checkpoint, etc.) from a user specified file given on the command line when it is invoked. All PVM appli-
cations that will be checkpointed are required to have each participating task join 
a common group using the PVM group server, $\text{pvmgs}$. When a checkpoint is to be 
taken, $\text{pvmckpt}$ calls the function $\text{pvm\_groupckpt()}$. $\text{pvm\_groupckpt()}$, in 
turn, notifies the group server, which then sends a signal to all the group members 
to checkpoint. After completing the checkpoint, the members of the group inform 
the group server of the checkpoint completion, and of any errors that occurred 
during the checkpoint. The $\text{pvm\_groupckpt()}$ function returns any errors upon 
checkpoint completion.

Excluding the lack of amenities mentioned above, the main difference between 
$\text{mpvm}$ and MIST is that whereas MIST allows its checkpointing to be handled 
by the PVM daemon, $\text{mpvm}$ does this outside the PVM daemon. This leaves 
the PVM code in $\text{mpvm}$ relatively untouched. Overall, $\text{mpvm}$ is a reliable 
checkpointer akin to the MIST checkpointer, and fully satisfies the checkpointing 
needs we had for this research.
Chapter 3

RAID Techniques for Checkpointing

The advantages and conveniences that checkpointing provides are tempered by the undesirable qualities brought about by the accompanying checkpoint overhead. Part of our approach to reducing the impact of overhead involved the use of RAID techniques. In this chapter we will explain what RAID is and how we made use of RAID techniques [16, 41] in our checkpoint algorithms.

3.1 What RAID is

Error-correcting codes have been in existence for decades [8, 44, 49]. However, the technique of distributing data across multiple storage devices for the purpose of at-
taining high-bandwidth input and output, and using one or more error-correcting
devices for failure recovery is relatively new. This technique came to the fore with
the advent of RAID systems. RAID stands for Redundant Array of Inexpensive
Disks. RAID systems are made up of inexpensive drives grouped together as a
single logical device. The idea behind RAID is to provide equivalent or better disk
capacity, performance, and reliability for less than the price of one comparable-size
hard drive. There are six RAID configurations, RAID level 0 through RAID level
5. RAID level 0 is data striping. Levels 1 through 5 consist of data mirroring and
data parity techniques. SPFT and RS coding (described below) borrow directly
from the parity techniques employed in systems using RAID 5 (see [27, 47]).

RAID techniques make it possible to achieve high-bandwidth I/O by distribut-
ing data among several of the storage devices, and to regenerate the data in the
event of a disk failure. Besides disk arrays, these techniques have been employed
in the design of multicomputer and network file systems with high reliability and
bandwidth [19, 32]. Also, the technique is used to design fast checkpointing sys-
tems where extra processors, instead of disks, provide reliability [15, 22, 41]. All
such systems are called “RAID-like” systems.

Because the chance of a device failure becomes significant when data is stored
among n devices, a problem central to all RAID-like systems is the increased
likelihood of disk failure. More specifically, if the mean time to failure (MTTF)
of one disk is represented by F, then the chances of failure in a RAID system
containing $n$ such devices is $\frac{E}{n}$. Resultingly, fault-tolerance must be considered.

### 3.2 Trading Error Coverage for Performance

Traditionally, fault-tolerance has been accomplished by saving the state of each of the processors involved in a running application to disk. Thus, if there is an application running on twenty processors in PVM, twenty processor states will be sent to centrally located storage. Checkpointing in this way provides for the possibility of a failure ranging in size from one processor up to the entire system. However, because so many checkpoints are sent to disk simultaneously, this type of checkpointing results in the use of significant network and disk resources in all but the smallest programs.

Since the speed of writing the checkpoints to disk is the major source of overhead in traditional checkpointing, in this thesis, we explore algorithms that will allow us to lessen this overhead source. By writing fewer checkpoints to a centrally located disk, we reduce the overhead involved. However we correspondingly reduce the amount of fault coverage. We explore two algorithms, Single Processor Fault Tolerance (SPFT) and Reed-Solomon coding (RS), that send reduced amounts of data to a centrally located disk. Specifically, the SPFT algorithm allows for any single processor failure, while the RS algorithm provides for any user specified number of processor failures. In our tests of the RS algorithm, we allow
for double and triple processor failures. With the SPFT algorithm, the equivalent of one checkpoint is sent across the network, and in our tests of the RS algorithm, at most three checkpoints are sent across the network. Because the possibility of an unscheduled wholesale processor failure is unlikely, these algorithms will, in most cases, provide adequate fault coverage.

3.3 The Single Processor Fault Tolerance Algorithm

Consider the following scenarios:

1. During the run of a program, a single processor fails and remains unavailable.

2. CGJS, in which the checkpoints are sent to local disk, is being used. When the user attempts to restart his/her program, one of the processors is unavailable.

3. A user is running a PVM application in an environment in which privately owned workstations are loaned out by the owners while they are away from their machines. They reclaim their machines when they return. While the user is executing his program, an owner returns and claims one of the machines being used.

These are examples of single processor failures. Although Vaidya has shown that single processor failures are much more common than wholesale failures [53], traditional checkpointers treat such failures as wholesale failures.
The SPFT algorithm provides for the occurrence of a single processor failure. Under this algorithm, each processor sends its checkpoint to local disk as in a CGJS checkpoint. Additionally, a checksum is calculated and sent across the network to central storage. It is from this checksum and the checkpoints of the non-failed processors that, in the event of any single processor failure, the lost checkpoint can be restored. The strengths of this method are:

- Much reduced checkpoint latency and recovery time from that of traditional checkpointing schemes.
- Lower overhead.
- Lower impact on central storage.
- Straightforward integration with CGJS.

This method, which we call “SPFT-checkpointing” works as follows. First, a CGJS-checkpoint (in which the checkpoints are sent to local disk) is taken. Next, the application processors coordinate to take the bitwise exclusive-or of all the checkpoints. This “parity checkpoint” is then sent to a centrally located disk.

If a single processor fails for any reason, it may be replaced by another processor. The checkpoint lost on the failed processor can then be regenerated by taking the bitwise exclusive-or of all the remaining checkpoints and the checksum. Finally, all the processors can roll back to the state of their checkpoints on local storage and restart.
To understand how SPFT-checkpointing is able to achieve its increases in performance over traditional checkpointers, consider the following. If the speed of sending data across a network to central disk is $R_{Central-disk}$, then, assuming synchronization and message startup costs to be negligible, the time it takes to send a checkpoint to disk across that network is roughly $T = \frac{S_{Checkpoint}}{R_{Central-disk}}$, where $S_{Checkpoint}$ is the size of the checkpoint. If we use traditional FT-checkpointing methods, and send $N$ checkpoints across the network to central storage, then the time is roughly $T_{FT} = N \frac{S_{Checkpoint}}{R_{Central-disk}}$. This is where SPFT reduces overhead costs.

By having each processor send its checkpoint to the disk local to it, there is no contention for network or disk resources. Therefore, if a fan-in algorithm is used to calculate the checksum, the time cost of SPFT checkpointing is roughly $T_{SPFT} = S_{Checkpoint} \left( \frac{1}{R_{Central-disk}} + \frac{1}{R_{Local-disk}} + \log N \left( \frac{1}{R_{Network}} + \frac{1}{R_{xor}} \right) \right)$, where $R_{Local-disk}$ is the speed of storing data to a local disk, $R_{Network}$ is the speed of sending data from processor to processor, and $R_{xor}$ is the speed of computing parity.

Thus, overhead is reduced by SPFT-checkpointing. This is due to the lack of competition for the central disk resources, and because $R_{Local-disk} >> R_{Central-disk}$.

### 3.4 The Reed-Solomon Algorithm

The RS algorithm extends the SPFT algorithm to tolerate multiple failures. SPFT is an algorithm based on the “$N + 1$-Parity” technique. Using this technique one
can store $Nk$ bytes of data on $N + 1$ devices, each of which holds $k$ bytes, in such a way that the failure of any one device is tolerated. The initial $N$ devices store the bytes themselves, while the final device stores the bitwise exclusive-or of those bytes. This final "parity-checkpoint" can be used to restore any one lost checkpoint. The advantage of this method is that it is simple. The disadvantage of this method is that recovery is possible from only one device failure.

The RS algorithm extends the $N + 1$-Parity method to recover from multiple failed devices. More specifically, the RS algorithm uses $N + m$ devices to hold $Nk$ bytes of data in such a way that any $m$ device failures may be tolerated.

Several other methods have been proposed to extend $N + 1$-Parity to recover from multiple device failures. These include "2d-parity", "$m$-dimensional parity", and the EVENODD algorithm [27], EVENODD being the most efficient algorithm known to tolerate two-device failures. However, none of these algorithms can effectively recover from any arbitrary $m$ failures, as the Reed-Solomon algorithm can [27]. Also, the RS algorithm requires the minimum number of additional devices needed to provide recovery.

The RS algorithm achieves the minimal number of extra devices by applying Reed-Solomon codes. Reed-Solomon codes are well-known in error-correcting coding theory, and have been used in correcting errors on noisy communication lines [44, 56]. They are also known as the default mechanism in providing $m$-device reliability with the addition of exactly $m$ checksum devices [2, 12, 29].
The costs for using the RS algorithm are small. The computational overhead required to compute a checksum consists of two table lookups, two additions, two conditionals, and a parity operation per data word. Thus, this algorithm, while not as efficient as parity-based schemes (which require simply one parity operation per word), is practical for many system applications. Full details of the RS algorithm may be found in [47]. We sketch the algorithm by way of an example below.

The algorithm is implemented as follows. A CGJS-checkpoint is taken to local disk, as in the SPFT algorithm. Next, the checksums are created. The number of checksums created is user-specified, and depends on the number of failures against which the user wants to be resilient. For the sake of example, assume we are running a PVM application on nine hosts (D_1 through D_9), and that we will calculate two checksums C_1 and C_2 that will be stored on the central disk. Thus we will be able to tolerate the failure of any two processors.

These coding functions operate on a word-by-word basis, where the size of each word is w bits, w being chosen by the coding algorithm. In our example, w = 4. This being the case, we can view our problem as consisting of 9 data words d_1, ..., d_9 and 2 checksum words c_1 and c_2, which are computed from the data words in such a way that the loss of any 2 words can be tolerated.

To compute a checksum word c_i for checksum device C_i, we apply the function F_i to the corresponding data word d_i in all nine hosts:
Recovery is handled as follows. First, a function to restore the words $d_j$ from the words for the non-failed devices is constructed for each failed device $D_j$. Then, if there are any lost checksums $C_l$, they can be recomputed with $F_i$. For lower level details of this algorithm, see [47].

In the SPFT algorithm, the computation of checksums, and recovery from a failure using those checksums, uses only XOR operations. The RS algorithm requires more complex calculations. Specifically, the Vandermonde matrix is used in the calculation of the checksums, Gaussian Elimination is used in recovery from failures, and Galois Fields are used for all arithmetic operations. The part each plays in the RS algorithm is detailed below.

### 3.4.1 The Vandermonde matrix

In the RS algorithm, a Vandermonde matrix is used to calculate the checksums. If we let $F$ be an $m \times n$ Vandermonde matrix, and represent the data and checksum words as vectors $D$ and $C$ respectively, then the checksums are calculated using the following formula:

\[ FD = C. \]
The Vandermonde matrix is defined as: \( f_{i,j} = j^{i-1} \). Thus, the above equation becomes:

\[
\begin{bmatrix}
1^0 & 2^0 & \cdots & n^0 \\
1^1 & 2^1 & \cdots & n^1 \\
\vdots & \vdots & \ddots & \vdots \\
1^{m-1} & 2^{m-2} & \cdots & n^{m-1}
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_2 \\
\vdots \\
d_n
\end{bmatrix}
= 
\begin{bmatrix}
c_1 \\
c_2 \\
\vdots \\
c_m
\end{bmatrix}
\] (3.1)

The calculation of the checksums, therefore, can be done by simple arithmetic (although it is a special kind of arithmetic, as explained below).

### 3.4.2 Gaussian Elimination in RS recovery

Recovery in the RS algorithm is effected via the use of Gaussian Elimination. To explain how this is done, we define the matrix \( A \) and the vector \( E \) as follows:

\[
A = \begin{bmatrix} I \\ F \end{bmatrix}, \text{ and } E = \begin{bmatrix} D \\ C \end{bmatrix},
\]

where \( I \) is the identity matrix, \( F \) is the Vandermonde matrix, \( D \) is a vector of the data devices, and \( C \) is a vector of the checksums. From this, we get the equation \((AD = E)\), or:
Each device can be viewed as having a corresponding row of the matrix $A$ and of the vector $E$. When a device fails, the failure is reflected by deleting a row from $A$ and $E$, which forms the new equation:

$$A'D = E'$$

So, if $m$ devices fail, then $A'$ is a $n \times n$ matrix. Because matrix $F$ is a Vandermonde matrix, every subset of $n$ rows of $A$ is guaranteed to be linearly independent. Therefore, $A'$ is non-singular, and the values of $D$ may be calculated from $A'D = E'$ using Gaussian Elimination. Hence, all checkpoints can be recovered.
3.4.3 Galois Fields

When computing checksums in the RS algorithm, the domain and range of those computations must be of a fixed length. Thus, the computations must be done using a special kind of arithmetic. While traditional algebra is guaranteed to give correct results when all the elements are infinite precision real numbers, we must make sure that correct results are achieved for fixed-size words. For this reason, addition and multiplication must be performed over a Galois field with more than \( n + m \) elements [49].

We, therefore, use Galois Fields for any addition, subtraction, multiplication, or division, when performing computations for the RS algorithm. Galois Fields are fields with \( 2^w \) elements (denoted \( GF(2^w) \)), and are a fundamental topic in algebra (e.g., [30, 44, 54]). In this section, we show how arithmetic operations are performed using Galois Fields without fully explaining Galois Fields in general.

Addition and subtraction of elements of \( GF(2^w) \) are simple. They are simply the XOR operation. For example, in \( GF(16) \):

\[
11 + 7 = 1011 \oplus 0111 = 1100 = 12.
\]

\[
11 - 7 = 1011 \oplus 0111 = 1100 = 12.
\]

Multiplication and division are not so trivial. They require two mapping tables, each of length \( 2^w \), which are analogous to logarithm tables for real numbers:
• \textbf{gflog[ NW ]}: A table that maps an integer to its logarithm in the Galois Field. \((NW = 2^w.)\)

• \textbf{gfilog[ NW ]}: An inverse table that maps an integer to its inverse logarithm in the Galois Field.

With these two tables, we can multiply two elements of \(GF(2^w)\) by adding their logs and then taking the inverse log, which yields the product. To divide two numbers, we instead subtract the logs. As with regular logarithms, zero is treated as a special case. [47] gives complete details for generating \textbf{gflog} and \textbf{gfilog}, and for performing multiplication and division. The exact requirements of a multiplication or division are one conditional, three table lookups (two logarithm table lookups and one inverse table lookup), an addition or subtraction, and a modulo operation.

3.4.4 RS Algorithm Summary

In summary, given \(N\) data devices and \(m\) checksum devices, the following steps are followed in the RS algorithm:

1. Choose a value of \(w\) such that \(2^w \geq N + m\).

2. Set up the tables \textbf{gflog} and \textbf{gfilog}.

3. Set up the Vandermonde matrix \(F\), where \(f_{i,j} = j^{i-1}\) (for \(1 \leq i \leq m, 1 \leq j \leq N\)), and multiplication is performed over \(GF(2^w)\).
4. Use the Vandermonde matrix to calculate the checksums.

5. If any number of devices up to \( m \) fail, construct the matrix \( A' \) and the vector \( E' \). Then use Gaussian Elimination to solve for \( D \) in the equation \( A'D = E' \).
We believe that the performance of the SPFT and RS algorithms in comparison to that of traditional checkpointing algorithms warrants their consideration as checkpointing alternatives. In this chapter, we detail the implementation and performance of both algorithms. First, we cover how we implemented SPFT. This is followed by an explanation of our implementation of the RS algorithm. Next we discuss each of the applications used in our tests. Finally, we show the performance of the SPFT and RS algorithms in comparison to that of the standard checkpointing algorithm.

4.1 Machine Configuration

In all of our tests, we used the mpvm checkpointer, which we augmented to perform the RS and SPFT algorithms. The tests were conducted on a network of
30 SUN IPX workstations at the University of Tennessee. Each workstation runs SunOS 4.1.3 and contains 16 megabytes of memory. The network is connected by 10 megabit per second ethernet.

Each workstation contains 424 Mb of local disk space, and is connected via Sun NFS to a central disk with 1.95 gigabytes of storage. We ran each of our tests on 25 processors during a time in which the machines and the network (but not the central file system) were allocated exclusively for our use. Also, we used copy-on-write checkpointing in every case. The implementation details follow.

4.2 SPFT Implementation

mpvm, as has been stated, has standard disk-based checkpoint/restart ability in which the checkpoints are sent to some central location. In the results below, we will abbreviate this as FT. We implemented SPFT in mpvm by adding its functionality on top of that already contained in mpvm. This was accomplished in two steps. The first was adding CGJS functionality to mpvm, and the second was adding SPFT functionality to that. We present here a high-level description of how we did this.

4.2.1 Implementation of CGJS in mpvm

To implement CGJS in mpvm we first added code to the pvmckpt program that enabled it to recognize a new flag, spft, in the “directives” that it reads
upon startup.

In the unaltered version of mpvm, only the group server created a checkpoint directory if one did not exist. We added code that had each task, in addition to the group server, create a checkpoint directory if the spft flag was assigned a value of 1 in the directive file. The logic behind this is as follows. If the flag has a positive value, a checkpoint directory local to each task in the PVM is created. If the flag is given a value of 0, only the group server creates a checkpoint directory. It is assumed that if the group server creates the sole checkpoint directory, the directory is on a centrally located disk. If, instead, each task creates a checkpoint directory, then the directory is one local to each processor. Thus, if the spft flag is set, the checkpoints will be sent to a directory local to each processor. The PVM tasks can then be restarted from these local checkpoints in the event of a task failure, or if a temporary halt and subsequent restart of the PVM tasks is desired. In this way, CGJS functionality was added to mpvm.

4.2.2 The Parity Process

To add SPFT-checkpointing to this, we created a separate running process to write the parity checkpoints needed for the SPFT algorithm. This process is called the “parity_process”, and runs on every processor that is involved in running the current PVM application. After the parity processes are started on each host, they are sent an array that contains the task i.d. (tid) of every parity process spawned.
From this array, each parity process computes its “parity process i.d.”, where the parity process whose tid is in array slot 0 gets i.d. 0, the parity process whose tid is in array slot 1 gets i.d. 1, and so on. This information (the list of tids and the individual parity process i.d.’s) is used as a means of coordinating communication between the parity processes when the parity checkpoint is created. How this is done will be explained forthwith.

4.2.3 Parity Checkpoint Creation

The parity checkpoints are created as follows. When the spft flag is set, the parity processes are notified each time a checkpoint is completed. Each parity process then reads the checkpoint stored on local disk, and works in concert with the other parity processes to create an XOR of each checkpoint associated with the current PVM application. The processes communicate via a fan-in algorithm. The parity process i.d.’s mentioned above make implementing this algorithm straightforward. For example, if we ran a PVM process on seven hosts, the i.d.’s would range from 0 to 6. They would pair up in three waves of XOR-ing. First, 0 would XOR with 1, 2 with 3, and 4 with 5. Next, 0 with 2 and 4 with 6. Lastly, 0 would XOR with 4, and 0 would write the result to central storage. In this way, in all but the last wave of XOR-ing, at least two pairs of parity processes are XOR-ing their data simultaneously. Efficiency is thereby enhanced.

As explained above, the parity processes use the fan-in algorithm to create the
parity checkpoint in an efficient manner. They also create the parity checkpoint one page at a time so as to minimize the amount of memory consumed. In other words, when XOR-ing the checkpoints, the parity processes do so in a loop in which one page of data is read, XOR-ed, and then written to central disk. The parity process with the lowest i.d. (which is always the parity process with i.d. 0) is the one responsible for writing.

4.2.4 SPFT Recovery

SPFT-recovery was effected by adding a recovery function, fix_spft(), to pvmckpt. This function is called if one or more parity processes fails, if any number of the PVM tasks fails, or if there is a single processor failure (if there is a multi-processor failure, the PVM application is halted). In the case of a parity process failing, fix_spft() simply spawns another parity_process process on the host on which the failure occurred. The new parity process is then given the i.d. of the one that failed. If a PVM task has failed, all tasks are halted and restarted using the CGJS-checkpoints stored on local disk. Lastly, if a processor has failed, all tasks in the PVM application are halted, as occurs in the aforementioned task failure recovery scenario. Restart, at this point, is impossible, as one of the CGJS-checkpoints is stored locally to the dead processor, and is thus unavailable. This being the case, a new processor is selected on which a PVM daemon is started via the pvm_addhosts() function call. Once this takes place, a parity process
is spawned on the new host. This parity process assumes the i.d. of 0 so that it will be the one to write the result of any parity checkpoints taken (the reasoning behind this will be explained shortly). All parity processes are notified of the i.d. change, and the parity process that was originally parity process 0 is given the i.d. of the parity process that was lost when the processor faulted.

The processes then proceed to take a parity checkpoint. This parity checkpoint is different however. When the new parity process 0 is spawned, a flag is set within the process that causes it to read the parity checkpoint from central disk as its checkpoint. The flag also causes parity process 0 to write the results of the next checkpoint XOR to its local disk. In this way, when the parity processes coordinate to produce the parity checkpoint, the bitwise exclusive-or of the surviving processors' checkpoints and the parity checkpoint on central storage is calculated and then written by parity process 0 to its local disk, effectively replacing the lost checkpoint. After this is completed, all processes roll back to the checkpoints on local storage.

4.3 RS Implementation

We now cover the implementation of the RS algorithm in mpvm. It, like the SPFT algorithm, was added to the functionality of mpvm. The RS algorithm, also very much like the SPFT algorithm, is initiated by setting a flag — in this
case, the RS flag — in the directive file. Once this flag is set, mpvm runs the RS algorithm.

Upon seeing that the RS flag is set in the directive file, pvmckpt looks in the same file for the number of checksums to create, where to store the checksums, and where to store the checkpoints. After starting the PVM application, pvmckpt spawns a copy of the parity process on each host that is involved in running the application. Again, as in the SPFT algorithm, each of the parity processes is sent an array of tids from which each parity process derives its i.d. The i.d.'s help in the creation of the checksums (as will be explained in the following paragraphs). The parity processes are then sent a signal that causes each to set up the gflog and gfilog tables, and to create a Vandermonde matrix. Thus, each parity process has in its address space all the information it needs to calculate checksums, and to assist in recovery.

4.3.1 RS Checksum Creation

A new set of checksums is created after each checkpoint. As in the SPFT algorithm, each parity process reads the checkpoint on local disk one page at a time, and coordinates with the other parity processes using the fan-in algorithm to XOR all of the checkpoints together. However, unlike the SPFT algorithm, before the parity processes send a page of checkpoint data to be XOR-ed, they use Galois Field multiplication to multiply that page with the correct element in
the Vandermonde matrix. This product along with the subsequent XOR-ing is
the equation \( FD = C \), which we discussed in the explanation of the RS algo-


rithm. Performing this action in a loop, one loop for each checksum, creates the
user specified number of checksums. Each parity process uses its i.d. along with
the loop index to determine the correct Vandermonde matrix element to use for
multiplication. This is best explained by an example. Assuming \( F[a][b] \) signifies
a Vandermonde matrix element, in loop number 1, parity process 1 (pp1) would
use element \( F[1][1] \) for multiplication, pp2 would use \( F[1][2] \), pp3 — \( F[1][3] \), and
so on. In loop 2, pp1 would use \( F[2][1] \), pp2 — \( F[2][2] \), pp3 — \( F[2][3] \), etc. This
is simply the process of using the loop number to denote the row number and
the parity process i.d. to denote the column number. This continues, one loop for
each desired checksum. In this way, the correct Vandermonde element is used for
multiplication, and the user specified number of checksums is computed. Parity
process 0 is responsible for writing the checksums to a central disk location.

4.3.2 RS Recovery

If any number of tasks fail, whether they be PVM tasks or parity processor tasks,
the event is handled exactly as in the SPFT algorithm. If some number of proces-
sors, say \( n \) (where \( n \leq m \), and \( m \) is a user defined maximum number of failures
to be tolerated) fails, the following occurs. The function \textbf{fix\_RS()} is called. This
function halts all remaining PVM tasks, selects \( n \) processors and spawns PVM
daemons followed by parity processes on each. \texttt{fix\_RS()} next sends a signal to the parity processes which causes each to build the matrix $A'$. $A'$ is then inverted by the parity processes and multiplied with the correct rows of the vector $E'$, which is formed from the remaining checkpoints and the necessary number of checksums. In this way, the equation $D = (A')^{-1}E'$ is solved for, where $D$ represents the lost checkpoints.

4.4 Experiments

We used our enhanced version of \texttt{mpvm} to test FT-checkpointing, SPFT-checkpointing, and RS-checkpointing on two applications. These were the CELL program and the PSTSWM program. These applications were chosen because they are examples of long running scientific parallel programs that can benefit from fault-tolerance. In each test, we measured the performance of the checkpointing algorithms mentioned above.

There are three important metrics by which checkpointing algorithms are typically judged. These are:

- Checkpoint Time: This is the time it takes for a checkpoint to run from start to finish.
- Checkpoint Overhead: This is the amount of time added to the run time of a program due to checkpointing.

41
- Checkpoint size: This is the size of the checkpoint.

Checkpoint overhead is the most important of the three. This is because it is the metric that the user will notice most. If the overhead is too high, the user will probably opt not to use checkpointing. Thus it is desirable to reduce this metric as much as possible. The checkpoint time derives its importance from the fact that it affects the number of times a checkpoint can be taken during the execution of a program. Lastly, the checkpoint size matters because there must be enough disk space to hold the checkpoint. For these reasons, it is desirable to keep each metric to a minimum.

As our base case (BASE), we ran each application without taking any checkpoints. We then executed the applications with FT-checkpointing. This was done in order that we could compare the FT-checkpointing algorithm against those we have been advocating in this paper. Finally, we concluded with SPFT and two RS-checkpointing tests — one to provide resilience against 2 processor failures, and the other for 3 processor failures. All checkpointing was performed in copy-on-write fashion, and in all tests, only one checkpoint was taken. We did not run any tests using synchronous (i.e., without copy-on-write) checkpointing because of the overhead it would impose given the size of the checkpoints in our tests.
4.4.1 The CELL Program

The CELL program is a C program that simulates cellular automata over an initialized grid for some specified number of generations. In this program, two grids are used. The first is a representation of the \( i \)th generation, and it is used to calculate the second grid, which represents generation \( i + 1 \). After this calculation, the grids switch identities.

We ran three series of tests on the CELL program. In the first, we calculated 50 generations on a 12,550 by 12,550 grid. The average size of the checkpoints was 13.0 megabytes per processor. Table 4.1 below details the results we obtained in the base case, and when using each of FT, SPFT, and RS-checkpointing.

<table>
<thead>
<tr>
<th>CASE</th>
<th>NRuns</th>
<th>AvgTime (min:sec)</th>
<th>AveTime (sec)</th>
<th>StdDev</th>
<th>NCkpts</th>
<th>CkptSize (Mb)</th>
<th>AvgCkptTime (min:sec)</th>
<th>OvHd (min:sec)</th>
<th>NCksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>26:59</td>
<td>1619.8</td>
<td>7.4</td>
<td>1619.8</td>
<td>7.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td>41:35</td>
<td>2495.9</td>
<td>10.8</td>
<td>1</td>
<td>13.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPFT</td>
<td>29:19</td>
<td>1759.0</td>
<td>19.7</td>
<td>1</td>
<td>13.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td>30:43</td>
<td>1843.3</td>
<td>16.4</td>
<td>1</td>
<td>13.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td>32:00</td>
<td>1920.7</td>
<td>25.0</td>
<td>1</td>
<td>13.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the second series of CELL tests, we used a 9300 by 9300 grid, and calculated 325 generations. The results are shown in table 4.2.
In the third series of CELL tests, we tested only the SPFT and RS-algorithms. We used the same parameters that were used in the second CELL test, to wit, a 9300 by 9300 grid and 325 generations. We sent the checkpoints to the /tmp directory (swap), as this would keep the checkpoints in memory. The purpose of these tests was to ascertain whether or not an even further reduction in checkpoint overhead and time would result if the checkpoints were kept in memory. The times produced in these tests were compared with the BASE case of the second CELL test. The results are presented in table 4.3.

### Table 4.2: Test results of CELL with 7.3 megabyte checkpoints.

<table>
<thead>
<tr>
<th>CASE</th>
<th>NRuns</th>
<th>AvgTime (min:sec)</th>
<th>AvgTime (sec)</th>
<th>StdDev</th>
<th>NCkpts</th>
<th>CkptSize (Mb)</th>
<th>AvgCkptTime (min:sec)</th>
<th>OvHd (min:sec)</th>
<th>NCksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>6</td>
<td>30:36</td>
<td>1836.0</td>
<td>3.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td>1</td>
<td>45:28</td>
<td>2728.4</td>
<td>0.00</td>
<td>1</td>
<td>7.3</td>
<td>22:43</td>
<td>14:52</td>
<td></td>
</tr>
<tr>
<td>SPFT</td>
<td>6</td>
<td>31:45</td>
<td>1906.9</td>
<td>9.50</td>
<td>1</td>
<td>7.3</td>
<td>6:05</td>
<td>1:11</td>
<td>1</td>
</tr>
<tr>
<td>RS</td>
<td>5</td>
<td>33:29</td>
<td>2009.3</td>
<td>9.33</td>
<td>1</td>
<td>7.3</td>
<td>11:31</td>
<td>2:53</td>
<td>2</td>
</tr>
<tr>
<td>RS</td>
<td>7</td>
<td>34:35</td>
<td>2075.7</td>
<td>9.53</td>
<td>1</td>
<td>7.3</td>
<td>17:07</td>
<td>3:59</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 4.3: Test results of CELL with 7.3 megabyte checkpoints sent to /tmp.

<table>
<thead>
<tr>
<th>CASE</th>
<th>NRuns</th>
<th>AvgTime (min:sec)</th>
<th>AvgTime (sec)</th>
<th>StdDev</th>
<th>NCkpts</th>
<th>CkptSize (Mb)</th>
<th>AvgCkptTime (min:sec)</th>
<th>OvHd (min:sec)</th>
<th>NCksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>6</td>
<td>30:36</td>
<td>1836.0</td>
<td>3.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPFT</td>
<td>4</td>
<td>31:44</td>
<td>1904.8</td>
<td>5.03</td>
<td>1</td>
<td>7.3</td>
<td>6:08</td>
<td>1:08</td>
<td>1</td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>33:55</td>
<td>2035.0</td>
<td>5.51</td>
<td>1</td>
<td>7.3</td>
<td>11:48</td>
<td>3:19</td>
<td>2</td>
</tr>
<tr>
<td>RS</td>
<td>3</td>
<td>34:44</td>
<td>2084.7</td>
<td>9.56</td>
<td>1</td>
<td>7.3</td>
<td>17:19</td>
<td>4:08</td>
<td>3</td>
</tr>
</tbody>
</table>
4.4.2 Shallow Water Model (PSTSWM)

The PSTSWM program comes from the "ParkBench" suite of parallel benchmark programs [21]. This program, written in FORTRAN, is a shallow water model based on the spectral transform method [31]. We ran test case #2, "Steady state nonlinear geostrophic flow", on a 5 by 5 processor grid. The results are shown in table 4.4.

<table>
<thead>
<tr>
<th>CASE</th>
<th>NRuns</th>
<th>AvgTime (min:sec)</th>
<th>AvgTime (sec)</th>
<th>StdDev</th>
<th>NCkpts</th>
<th>CkptSize (Mb)</th>
<th>AvgCkptTime (min:sec)</th>
<th>OvHd (min:sec)</th>
<th>NCksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>5</td>
<td>45:31</td>
<td>2731.10</td>
<td>33.14</td>
<td>1</td>
<td></td>
<td>50:29</td>
<td>12:59</td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td>5</td>
<td>58:30</td>
<td>3510.20</td>
<td>2.84</td>
<td>1</td>
<td></td>
<td>14.4</td>
<td>50:29</td>
<td>12:59</td>
</tr>
<tr>
<td>SPFT</td>
<td>4</td>
<td>48:44</td>
<td>2924.20</td>
<td>17.43</td>
<td>1</td>
<td></td>
<td>14.4</td>
<td>12:40</td>
<td>3:13</td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>49:36</td>
<td>2986.70</td>
<td>14.47</td>
<td>1</td>
<td></td>
<td>14.4</td>
<td>23:06</td>
<td>4:05</td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>49:30</td>
<td>2970.10</td>
<td>23.49</td>
<td>1</td>
<td></td>
<td>14.4</td>
<td>36:09</td>
<td>3:59</td>
</tr>
</tbody>
</table>

4.4.3 Discussion of Results

Conclusions concerning the SPFT and RS-algorithms can be drawn based on their performance in the three important metrics: checkpoint time, checkpoint overhead, and checkpoint size. We here discuss the performance of both algorithms in comparison to that of FT-checkpointing, using these three metrics as the basis of comparison.

With regard to the checkpoint time metric, both the SPFT and RS-algorithms
outperform FT-checkpointing. This is due to the fact that although checksums must be created in the SPFT and RS algorithms, they both use CGJS-checkpointing to store their checkpoints. The speed with which the checkpoints are stored is nominal, and the bulk of the checkpoint time is taken in calculating checksums. The checkpoint times of the SPFT and RS algorithms ranged from roughly 31 percent up to about 91 percent those of the FT-algorithm, depending on which algorithm was used. This information is presented graphically in Figure 4.1.

This reduction is a significant one in that not only is the time to checkpoint much lower in SPFT and RS-checkpointing than in FT-checkpointing, but the central disk is not saturated to the point of unusability as it is while checkpoints are taken in FT-checkpointing. As was stated in our explanation of the SPFT and RS-algorithms, CGJS checkpointing out-performs FT-checkpointing because the speed of sending a checkpoint to local disk is much greater than that of sending a checkpoint across NFS to a central disk. Also, as only the equivalent of one checkpoint is sent to disk in the SPFT-algorithm, and only two or three in the RS-algorithm, the central location is not flooded with data as it is in the FT-checkpointing algorithm. Thus the improvement in the time to checkpoint.

Similar, albeit less dramatic, improvement can be seen in checkpoint overhead. This is illustrated in Figure 4.2. The figure shows the percentage reduction in checkpoint overhead we experienced in each of the program tests we ran.
Figure 4.1: Percentage reduction in ckpt time, per application, over FT-ckpting
Figure 4.2: Percentage reduction in ckpt overhead for each program we tested.
In our tests, the checkpoint overhead was a result of four factors:

1. The time needed to process page faults, and to copy pages during the copy-on-write checkpoints.

2. The time needed to fault pages to and from disk when physical memory is depleted during checkpointing.

3. Contention on the network due to the heavy network traffic that results when checkpoints are sent across the network during checkpointing.

4. The time to perform XOR’s (in SPFT), or to perform the calculations necessary to create a checksum device (in the RS-algorithm).

It would be expected that the SPFT and RS-algorithms would both improve upon the FT-checkpointing algorithm’s performance the first two sources listed above. This is because the extra copy of the PVM process is only exists during the time that the CGJS checkpoint is being performed. The results bear out the truth of this supposition.

The individual checkpoint sizes of the SPFT and RS-algorithms is exactly the same as that of the FT-algorithm. The SPFT and RS-algorithms use less central disk space than the FT-algorithm, but more local disk space. This is not a problem as long as there is sufficient local disk space to handle a single checkpoint. The size of the entire checkpoint (the aggregate of the sizes of all of the checkpoints taken by each processor involved) however, is less in the FT-algorithm than in either
of the SPFT or RS algorithms. Because a parity checkpoint is created in the SPFT-algorithm, the entire checkpoint in this algorithm is the equivalent of one processor checkpoint greater than the entire checkpoint of the same data under the FT-algorithm. In the RS-algorithm, the size difference is either two or three checkpoints greater, depending on whether or not two or three checksums are calculated. Because the individual checkpoints are distributed among the local disks of the processors involved in the checkpoint, as long as there is sufficient local disk space, the greater size of the checkpoints under the SPFT and RS algorithms is not “felt” by the user. This is because of the greater speed of CGJS checkpointing versus FT-checkpointing.
Chapter 5

Related Work and Summary

Related Work  Checkpointing in distributed systems has had a rich history. The survey paper by Elnozahy, Wang and Johnson provides an excellent history and background of the field [23]. Papers that are specifically relevant to this thesis are as follows.

Diskless checkpointing [41] was the original use of Raid Level 5 parity in checkpointing systems. Here, checkpoints are stored in memory, and extra processors rather than disks provide the fault tolerant medium. Since the original diskless checkpointing paper appeared, subsequent research projects have been completed which:

- implemented diskless checkpointing on a SIMD multicomputer [15],

- implemented diskless checkpointing on PVM [50],
• inserted diskless checkpointing into distributed matrix operations [22], and
• explored 1-dimensional parity in distributed matrix operations [35].

The original paper exploring the concept of trading off failure coverage for performance was written by Vaidya [53]. His algorithm for single-processor fault tolerance was for each processor to store its checkpoint locally, and on the local disk of another processor. This consumes more space than SPFT checkpointing, but has no reliance on central storage.

There are two papers upon which this thesis is based. These are [46] and [47]. This thesis extends the implementation in [46] to a real checkpointing system, complete with working recovery.

Summary As has been discussed, in today’s scientific computing environment, programs that require a long time to run are commonplace. These applications are often run on networks of workstations in order to take advantage of the computing power of multiple processors working in parallel. Due to the inherent risk of processor failure present in such multi-processor virtual machines, fault-tolerance is necessary. Traditional checkpointing algorithms (FT-checkpointing), however, provide their advantages with the disadvantage of saturated disk locations and high checkpoint overhead. For these reasons, we’ve explored two alternative checkpointing algorithms. These two algorithms, the SPFT-algorithm and the RS-algorithm, are based on RAID techniques, and provide fault-tolerance...
with reduced fault coverage. What RAID is, as well as an explanation of the
two algorithms, is covered in chapter 3. Details of the implementation of both
algorithms, and results of tests run with the two checkpoints are presented in
chapter 4. The results, when compared with the performance of FT-checkpointing
in similar tests, provide evidence of the validity of SPFT and RS-checkpointing
as legitimate checkpointing alternatives.

Because both the SPFT and RS-algorithms use CGJS-checkpointing rather
than sending their checkpoints to a central disk location, their checkpoint times
are considerably less. Also, checkpoint overhead in SPFT and RS-checkpointing
is much improved over that of FT-checkpointing. Lastly, the SPFT and RS-
algorithms do not result in saturated central disk locations. If there is a dis-
advantage to using these two algorithms, it is that they do not provide for a
wholesale failure. However, a checkpointing system can combine one of SPFT or
RS-checkpointing with a traditional checkpoint to cover the rare case of whole-
sale failure. Thus, the SPFT and RS-algorithms are attractive alternatives.
Bibliography


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.


Vita

Darryl V. Pace was born in Raleigh, North Carolina on July 19, 1966. He attended public schools in the Knox County School District, and graduated from Fulton High School in June, 1984. In the fall of the same year, he entered Vanderbilt University, where in May of 1989, he received his Bachelor of Science degree in Engineering Science. He entered the Computer Science Master’s program at The University of Tennessee in Knoxville in August of 1992. His Master of Science degree was received in December of 1997. He plans to live in Atlanta, Georgia after graduation.