EFFICIENT CHECKPOINTING ON MIMD ARCHITECTURES

James Steven Plank

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCEINCE

June 1993

Abstract

Presented here are efficient algorithms for checkpointing on MIMD architectures. These algorithms have been implemented on two representative machines: a shared-memory multiprocessor, and a message-passing multicomputer. The algorithms and implementations are evaluated according to three speed metrics: checkpoint time, overhead, and latency.

Checkpointing is important as a general means of software fault-tolerance. It is also the backbone of certain program control utilities, such as job-swapping, process migration, and playback debugging. We employ several techniques to minimize the invasiveness of the checkpointer on the target program. Such techniques are main memory checkpointing, copy-on-write, buffering, compression, and the elimination of bottlenecks and extra control messages.

The major result of this dissertation is that we can implement efficient checkpointing on MIMD architectures, thereby enhancing the usability of such machines.

Acknowledgements

I've been fortunate to work under the guidance of Kai Li. Kai has taught me many things about computer science and academia, and I have much to learn from his example of how to advise students. I am also indebted to Jeff Naughton for getting me interested in checkpointing, and being a collaborator in research.

I am indebted to several people for helping me finish this dissertation. I wish to thank Anne Rogers for being a reader, and Andrew Appel, David Dobkin and Pat Hanrahan for attending the defense. Melissa Lawson has been a lifesaver in administrivia. I'd also like to thank Jack Dongarra for providing me with space to work and access to machines, and Brad Vander Zanden for his generosity with his time.

Several people have donated their time and programs to help test checkpointing. I thank Michael Vose, Viktor Eijkhout, Jack Dongarra, Elmootazbellah Elnozahy, Willy Zwaenepoel, Susan Ostrochov, Al Geist and Steve Moulton for providing test programs for checkpointing. Karin Petersen, Matt Blumrich and Liviu Iftode have been invaluable in fixing and rebooting broken hypercubes while I'm 700 miles away.

There are many people who have made Princeton an enjoyable place to live and work. Adam Buchsbaum has provided me with more ways to waste time than I ever really needed. Norman Ramsey has shared with me a love of power tools, not-so-popular music, and writing non-thesis-oriented software. Adam, Mary Fernandez, Norbert Schlenker, Norah McCloy and Karin Petersen have provided me with many evenings of good food, card playing, and entertainment. Matt Blumrich and Mark Greenstreet have been solid and entertaining officemates for the duration of my stay in Princeton. Finally, I'd like to thank Stefanos Damianakis, Joe Studholme, Logan Fox, Ray Miller, Brad Vander Zanden, Bill Rosener, Gavin O'Brien, Derek Martin and especially David Dobkin and Geter Hicks for knowing the importance of devoting many hours of one's week to the playing of basketball.

I'd like to thank my parents and my sisters for providing me with love and emotional support all my life. Most of all, I thank my wife, Heather, for everything that has made my life happy since we've been together.

My last three years have been funded by an AT&T fellowship. This work has also been funded by NSF grants CCR-9020893 and CCR-8814265, DARPA and ONR contract N00014-91-J-4039, Intel Supercomputer Systems Division, and Digital Equipment Systems Research Center and External Research. I am grateful to Jack Dongarra, Al Demers and Xerox PARC for summer employment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, supercomputer systems with large numbers of processors, high-speed and scalable interconnects, and concurrent I/O systems have become commercially available. While the raw processing power of these machines is impressive, their general usability and scale is limited by the need for fault-tolerance. The combination of more and more components in a single system greatly reduces the system's mean time to failure. For example, if the mean time to failure of a microprocessor is F, then the mean time to failure of a system of n microprocessors is $\frac{F}{n}$. With such a large combination of components, hardware fault-tolerance is infeasible. Thus, supercomputers are greatly in need of software fault-tolerance.

This dissertation explores general software fault-tolerance on *Multiple-Instruction-Multiple-Data (MIMD)* architectures. Specifically, it presents algorithms for *checkpointing* on MIMD architectures, and evaluates implementations of these algorithms with respect to their speed and invasiveness. The main result of this dissertation is that we can implement efficient checkpointers on MIMD architectures, thereby adding fault-tolerance and job-swapping where there was none previously.

Checkpointing, also known as "backward error recovery," is an important means of software fault-tolerance as it is the only known mechanism that can tolerate unanticipated faults—faults that are not envisioned in the design of a system [AL81]. We present algorithms and optimizations for checkpointing the two types of MIMD architectures: Shared-memory *multiprocessors*, and message-passing *multicomputers*. All algorithms and optimizations are implemented on a representative multiprocessor and multicomputer. We discuss the implementations and their results in detail.

1.1 MIMD Architectures

This dissertation studies software fault tolerance for two kinds of MIMD architectures: multiprocessors and multicomputers.

1.1.1 Shared-Memory Multiprocessors

Figure 1.1 shows the basic architecture of multiprocessors. There are n processors in such a machine, each of which is connected to a bus or interconnection network. All processors can access all memory modules in the architecture.

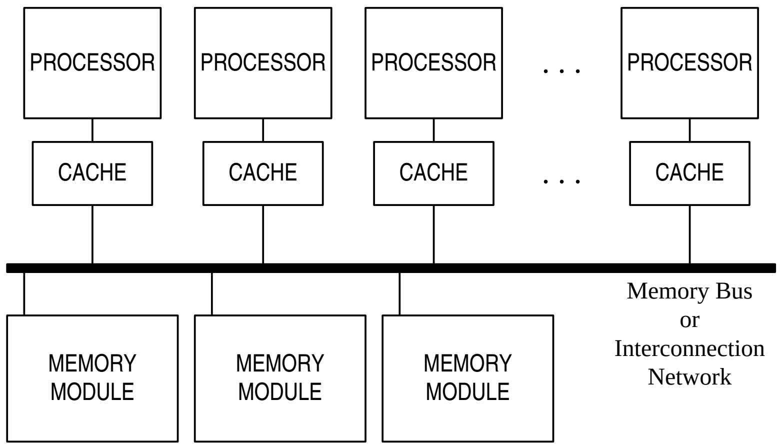

Figure 1.1: Basic Architecture of a Shared-Memory Multiprocessor.

In most multiprocessors, each processor has a local cache enabling it to make fast memory references. These caches have extra hardware between them and the bus or interconnection network that ensures that local cache entries do not become stale when a single memory location is present in several processors' caches. These are known as *snoopy caches*. The problem of keeping snoopy caches coherent is known as the *cache coherence* problem, and has a rich history in parallel architecture research [AB86].

Examples of multiprocessors are the Sequent Balance and Symmetry, the DEC-SRC Firefly, the SGI Powerstation, the BBN Butterfly, the Stanford DASH and MIT's Alewife. The first four have snoopy caches connected to a shared bus. Because the bus becomes a bottleneck, these multiprocessors are limited in the number of processors that they can handle. The machine with this design containing the largest number of processors is the Symmetry, with 32 processors.

On the BBN Butterfly, processors do not have caches. Instead, each processor is directly connected a local memory that is a small portion of the global address space. To access the rest of the global memory, a processor must issue a memory request over an interconnection network that forwards the request to another processor's local memory, which then services that request. This protocol allows for Butterflies to be built with a larger number of processors, the most being 256.

On the last two multiprocessors, instead of having a single, shared bus, there is an interconnection network which attaches the processors to the shared memory. Cache coherence is maintained by *directories*, which keep track of the memory locations contained in the processors' caches. The directories are responsible for sending out invalidation messages to other processors when possible coherence problems are encountered. Directory schemes are often combined with *relaxed consistency* models for building large scale multiprocessors [AH90, GLL$^+$90]. With relaxed consistency models, processors' caches are allowed to be inconsistent with memory and with other processors' caches until executing synchronizations or fences. This reduces the degree of false sharing and the number of invalidation messages.

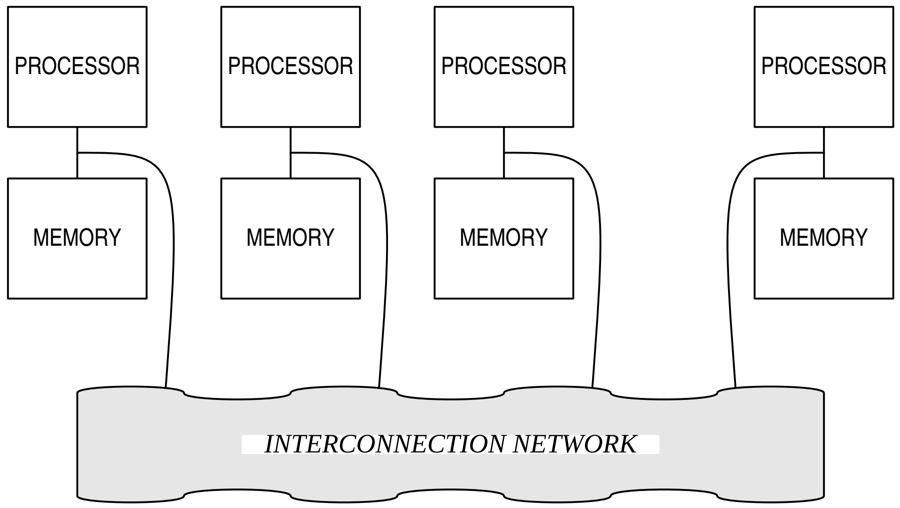

Figure 1.2: Basic Architecture of a Message-Passing Multicomputer.

1.1.2 Message-Passing Multicomputers

Figure 1.2 shows the basic architecture of multicomputers. Multicomputers consist of n processors, each with a local memory that no other processor can reference. The processors communicate with one another by sending messages over an interconnection network.

One of the most important parts of a multicomputer is the interconnection network. In general, machine designers try to design multicomputers with interconnects that are scalable, high-bandwidth and low-latency. Popular interconnection networks include hypercubes, two and three-dimensional meshes, and fat trees. With these scalable interconnection networks, multicomputers can be built with many more processors than multiprocessors. Table 1.1 gives a listing of example multicomputers with their interconnection network topologies and maximum number of processors.

Machine	Topology	Maximum Number of Processors	
		Currently Built	Design Limit
Intel iPSC/2	Hypercube	128	128
Intel iPSC/860	Hypercube	128	128
Intel Delta	2-d Mesh	570	570
Intel Paragon	2-d Mesh	500	1000
MIT J-Machine	3-d Mesh	1024	4096
CalTech Mosaic	3-d Mesh	1024	4096
Thinking Machines CM-5	Fat Tree	1024	65536

Table 1.1: Example multicomputers

1.2 Checkpointing

To checkpoint a program for fault-tolerance, one must save enough information to stable storage that after a machine fails and its volatile state is lost, one can reconstruct it from the information in stable storage. By "stable storage," we mean a medium that does not lose its values when it loses power. In this dissertation, we assume it means magnetic disks. Other types of stable storage are magnetic tapes and drums, and non-volatile RAM.

As pictured in Figure 1.3, a typical checkpointer periodically interrupts the execution of a program (that we call the *target*), and checkpoints its state to disk. After a failure, the target program's state is restored from the most recent checkpoint. This attempts to minimize the amount of work lost due to the failure.

1.2.1 Uses of Checkpointing

Besides fault-tolerance, checkpointing is a useful tool for other tasks. Some operating systems, like Intel's NX/2, have no facilities for *job-swapping*, and thus are difficult to share, especially with respect to long-running jobs. With a checkpointing tool, one can swap a program off a machine by checkpointing

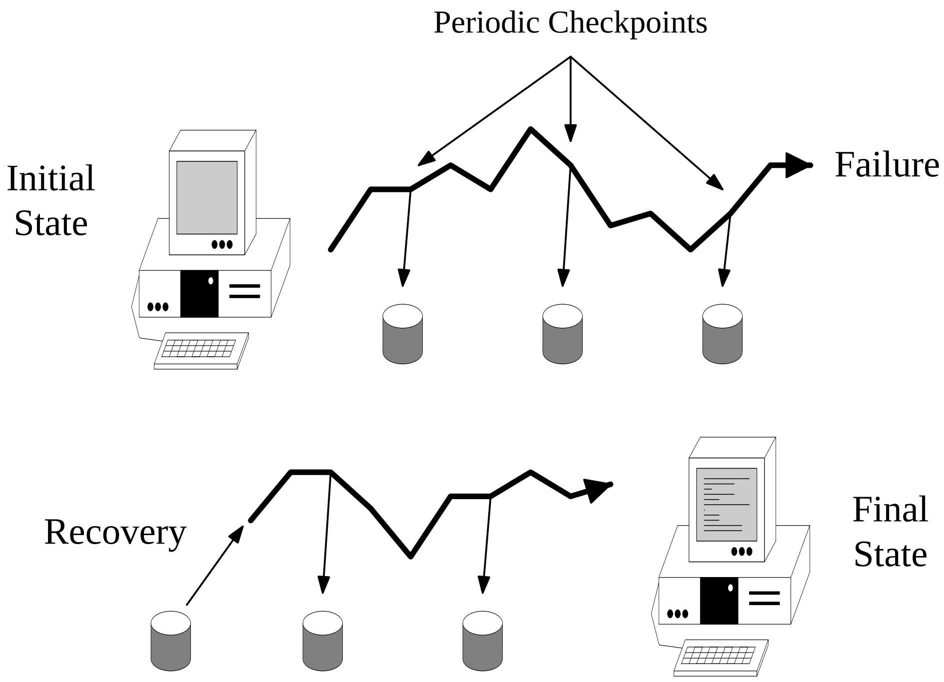

Figure 1.3: Periodic Checkpoints for Fault-Tolerance.

the program, and then killing its process. Swapping on is simply a matter of recovering.

Few operating systems provide generic *process migration*. With a checkpointer, one can migrate a process from one machine to another of the same type by checkpointing on the first machine, and then recovering on the second machine. In fact, checkpointing for fault-tolerance and job-swapping both fall under the category of process-migration: They migrate to the same machine at a different point in time.

An extension of processes migration is *program migration*. The difference is that a program might consist of multiple processes on the same or on different machines. With a checkpointing tool, one can checkpoint all the processes in such a program, and thus migrate the program to a different set of machines, or to a different point in time.

Finally, checkpointing can be useful as a tool for debugging. The information contained in a checkpoint is the same as a typical core file. If several checkpoints are maintained instead of just the most recent one, users can replay their programs back to any point in time to regenerate and discover error conditions. This is called *playback debugging.* Playback debugging is especially useful in parallel and distributed programming, as these programs contain more non-determinism than sequential programs, and thus error conditions may not be regenerated by simply restarting a program from the beginning. Playback debugging is also useful for debugging long-running programs, as users may not have ample time to restart their program and regenerate bugs.

1.2.2 How Checkpoints Are Taken

On a uniprocessor, checkpointing is straightforward. The point at which the target program is interrupted for checkpointing is called the *recovery point.* It is to this point that the program is restored upon recovery. The volatile state of a standard uniprocessor consists of the state of memory, and the state of the machine's registers. Upon checkpointing, enough information must be saved to disk so that upon recovery, the state of the registers and memory can be reconstructed, and the target program can proceed from the recovery point.

Figure 1.4 shows what information is saved in a typical uniprocessor checkpoint. It consists of the registers, and all of a process's address space except for the code. This is because on most machines the code is read-only, and is already saved on disk as part of the target's executable file. None of the operating system's address space is checkpointed: To do so, the checkpointer would have to be part of the operating system—otherwise it would not have permission to read or write the kernel address space. Instead, the checkpointer must save what information it can extract from the kernel's state, and attempt to reconstruct it upon recovery.

For example, on a Unix-like operating system, one can almost make the file system fully recoverable in the following way: At checkpoint time, for each open file, flush the file to disk, and save the current value of its `seek` pointer,

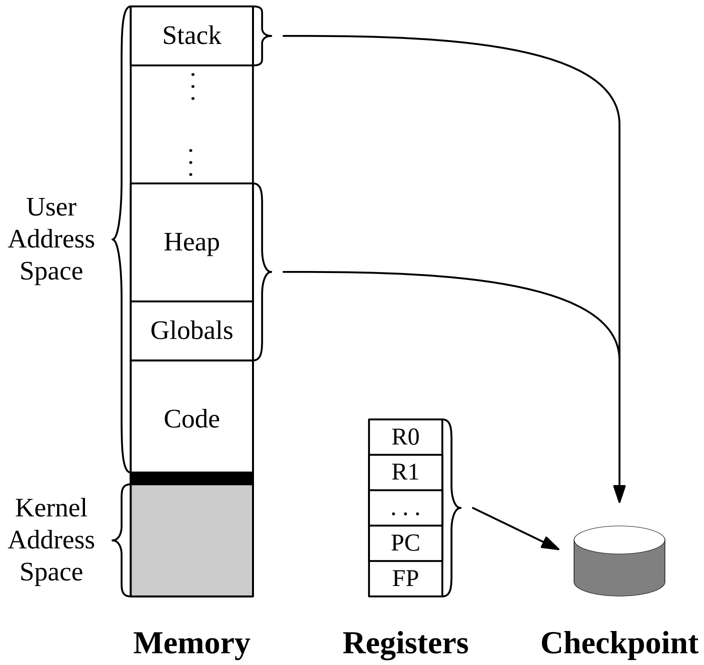

Figure 1.4: A Typical Uniprocessor Checkpoint.

as well as enough information to reopen the file later. Upon recovery, reopen each file and reset its `seek` pointer to the saved value. If the files are not deleted between checkpointing and recovery, and each file is either read-only, write-only, or sequentially written read-write, then all the files are guaranteed to be recoverable. Only randomly-written read-write files are non-recoverable. Figures 1.5 and 1.6 illustrate how this works.

In Figure 1.5, there are two files: File A is write-only, and File B is read-only. When the processor checkpoints, the `seek` pointers are saved with the checkpoint. When the processor fails, the `seek` pointers have been updated, as have the contents of File A. When the processor recovers, the `seek` pointers of Files A and B are reset to their values at the time of the checkpoint. As File B is read-only, its values are unchanged from the time of checkpointing. File A's values have changed, but since the file is write-only, the processor will eventually repeat the writes that have gotten it to its current state. Thus, the files are recoverable.

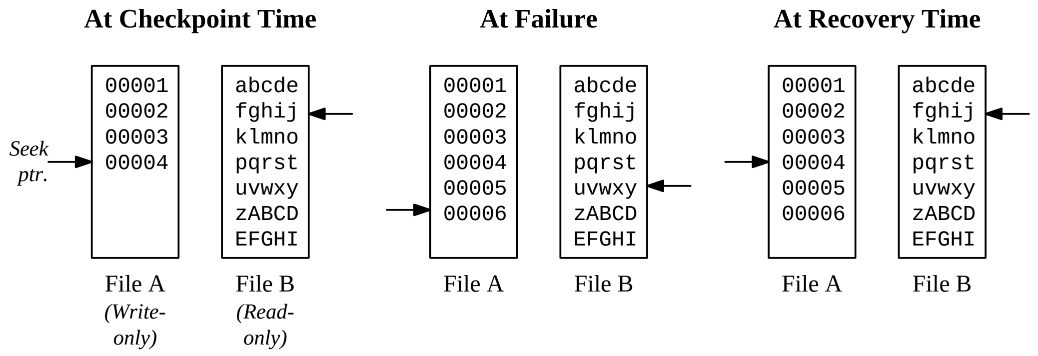

Figure 1.5: Example of Recoverable Files.

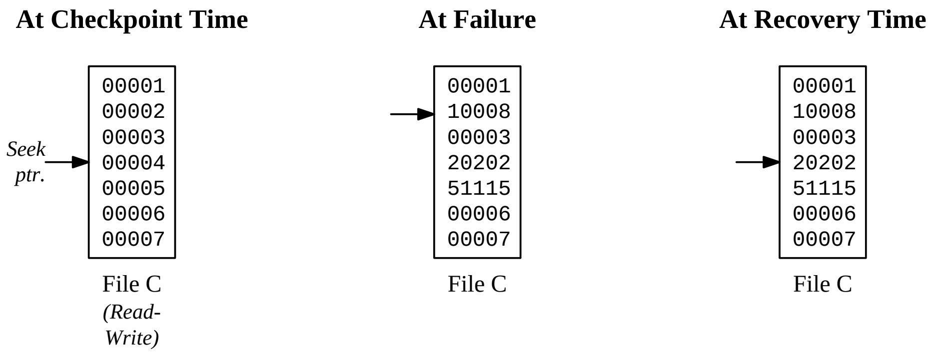

Figure 1.6: Example of Non-recoverable Files.

Figure 1.6 shows an example of a non-recoverable, randomly-written read-write file. After checkpointing and saving the value of the `seek` pointer, the processor skips around in the file, and performs reads and writes. Upon recovery, the file is reopened and the `seek` pointer is reset, but the contents of the file at the `seek` pointer have changed. If the processor reads this value, then it will receive `20202` instead of `00004`. Files of this sort cannot be recoverable by this kind of mechanism.

Other operating systems constructs are also not recoverable. For example, the value returned by the *getpid()* system call might change over time in the presence of checkpointing and recovery. Moreover, sockets and pipes are non-recoverable. For programs to be checkpointable without making changes to the operating system, they must restrict themselves to be bereft of any reliance on non-recoverable kernel information. We consider such restrictions to be reasonable. Other checkpointers [LS92, EJZ92] have also made these assumptions.

1.3 Speed of Checkpointing

1.3.1 Speed Metrics

One of the most important properties of a checkpointer is its speed: Users are not likely to employ a tool that significantly slows down their programs, even if doing so renders them vulnerable to failures. Thus, when designing a checkpointer, emphasis must be placed on the speed of the checkpointing algorithms. For checkpointing, the measurement of speed can be divided into three metrics: checkpoint time, checkpoint overhead, and latency. Each is explained below.

Checkpoint Time

Checkpoint time is the total time that it takes for the checkpointer to take a checkpoint. It is a measure of how long it takes the checkpointer to run, and also determines how many checkpoints can be taken over a period of time. Checkpoint time is limited by the speed of writing to stable storage. For example, if a checkpoint file is four megabytes long, then the checkpoint time must be at least as long as it takes to write four megabytes to disk. As a rule of thumb, fast disk drives write at a speed of one or two megabytes per second; therefore checkpoints of programs with large writable address spaces must take at least a few seconds to complete.

Checkpoint Overhead

Checkpoint overhead is the total time added to the application program as a result of the checkpoint. Note that this can be less than the checkpoint time if, for example, the writing to stable storage can be overlapped with the execution of the target program. Checkpoint overhead is a measure of how invasive the checkpointer is, and thus is related to the amount of concurrency that a checkpointing algorithm has. Specifically, we define concurrency as:

$$concurrency = \left(1 - \frac{overhead}{checkpoint\ time}\right) * 100\%.$$

One of the goals of our checkpointing algorithms is to maximize concurrency, and therefore decrease the checkpoint overhead as much as possible.

Latency

Latency is a measure of real-time behavior. We define it to be the maximum amount of time that the target program is interrupted as a result of the checkpointer. Decreasing latency is important, especially if the target program is interactive. A checkpointer will be successful if users do not perceive it as freezing their program for large chunks of time. Another goal of our algorithms, especially those concerning multiprocessor workstations where a user is more likely to be at a console watching his or her program, is that the latency be small—the checkpointer should only interrupt the target for small, fixed periods of time, such as a tenth of a second.

1.3.2 Incremental vs. Non-incremental Checkpoints

The best way for a checkpointer to reduce its checkpoint time is to reduce the amount of information saved in a checkpoint. One way to reduce this amount was suggested by Feldman and Brown for playback debugging [FB89]. Instead of taking a complete checkpoint at periodic intervals, they take *incremental checkpoints*. That is, only the state that has changed since the previous checkpoint is recorded. To recover from an incremental checkpoint, one must recreate the recovery point from a number of incremental checkpoints, rather than just the most recent one. The assumption made with incremental checkpointing is that programs tend to exhibit locality of reference, and that the state changed between checkpoints is a small fraction of the total volatile state of the program.

In our algorithms, we take full instead of incremental checkpoints. The reason is that we want to test the worst-case behavior of our checkpointing algorithms. The work involved in taking an incremental checkpoint is nearly a subset of the work involved in taking a full checkpoint (some extra work must be performed to figure out which pages to checkpoint). Therefore if the algorithms perform well according to the speed metrics with non-incremental checkpointing, then they should perform even better with incremental checkpointing. The incremental checkpointer by Elnozahy, Johnson and Zwaenepoel confirms this claim [EJZ92].

1.4 Related Work

1.4.1 Database and Uniprocessor Checkpointing

The bulk of work on and implementations of checkpointing for fault tolerance has been in database and transaction-processing systems [DKO+84, Pu85, Hag86, SGM87, LN88]. Although these are a major class of applications, the fact that database computations can be viewed as atomic transactions greatly simplifies the act of checkpointing. We concentrate on general purpose programs, so no such computational model can be assumed.

General-purpose checkpointing has been studied and implemented on uniprocessors. Proposals and overviews have been described [Ran75, AL81, Lam81], and implementations for playback debugging have been provided [FB89, PL88, Wit89]. As mentioned above, the debugger Igor [FB89] performs incremental checkpointing. It uses virtual memory management to detect changed pages and reconstructs an execution state from a number of snapshots.

Of special note is a system called *Condor* [LS92], which is a checkpointing facility that runs on numerous commercial uniprocessors. Condor's focus is on process-migration and job-swapping for batch computing, rather than on fault-tolerance. Its mechanism for checkpointing is to take a core-dump at each recovery point. This way their code is easily portable to a number of different machines (although the checkpoints are not). Condor runs completely in the user's address space, and imposes the same restrictions on checkpointable programs as outlined in Section 1.2.2 above.

There has been research in making file and operating systems recoverable. Taylor and Wright describe a backward recovery implementation that focuses on a file system for Unix that is fully recoverable [TW86]. This removes the restrictions on randomly written read-write files described in Section 1.2.2. Some operating systems, such as Sprite [OCD+88], Accent [FR86], and the V System [TLC85] include the ability to migrate processes as one of their major design goals. In such a system, checkpointing should be fully supportable with no restrictions, as checkpointing is a process migration to stable storage.

Finally, there have been some hardware designs that would facilitate checkpointing. Staknis proposed a new memory design called sheaved memory [Sta89] for supporting checkpointing in paged systems. In a sheaved memory, physical page frames can be bundled together so that data written to one frame in the bundle is simultaneously written to all frames in the bundle. If the entire address space consists of bundles of more than two pages each, then one may checkpoint the address space by removing one frame from each bundle.

The price of non-volatile memory is decreasing to the point that within the next ten years, it might be affordably incorporated into workstation architectures [BAD+92]. Such memories can greatly improve checkpointing speeds as they do not have the large write latency of disks. However, like sheaved memory, non-volatile memories are currently expensive to build, and not yet widespread enough to warrant further consideration.

1.4.2 Multiprocessor Checkpointing

There has been little explicit work published on checkpointing multiprocessors. As we will explain in Chapters 2 and 3, multiprocessor checkpointing is very similar to uniprocessor checkpointing, except that there is more parallelism that can be used to decrease the overhead of checkpointing. Using a multiprocessor in this way is suggested by Feldman and Brown [FB89] in terms of checkpointing for playback debugging. Moreover, the hardware designs for checkpointing mentioned above would also greatly facilitate multiprocessor checkpointing.

1.4.3 Distributed Checkpointing Algorithms

Multicomputers bear a great deal of similarity to distributed systems, and thus the vast amount of research on distributed checkpointing algorithms is relevant to checkpointing multicomputers. Checkpointing algorithms for distributed computing fall into four general categories. First are algorithms for distributed database checkpointing. As stated above, these rely on special properties of database computations, and are of little use for general-purpose computations. Second are *pessimistic* algorithms. These are algorithms that assume frequent failures of the system, and thus take strict precautions to ensure the reliability of computations. Pessimistic algorithms tend to impose heavy penalties on failure-free performance for the sake of reliability. *Optimistic* checkpointing algorithms are the opposite. They assume that failures occur relatively infrequently, and therefore attempt to push most of the overhead of checkpointing into the recovery process. The last type of checkpointing algorithms are called *consistent* checkpoints. Consistent checkpointers try to strike a balance between optimistic and pessimistic checkpointers. In other words, they attempt to provide good failure-free performance, yet simple recovery. Consistent checkpointers tend to be best suited for multicomputer checkpointing.

Pessimistic Checkpointing Algorithms

In a pessimistic checkpointing scheme, each process is backed up by a *mirror* process. All messages in the system are sent to three locations: the specified destination process, the destination's mirror, and the sender's mirror. The system must guarantee that this operation be atomic. In other words, if a process receives a message m_1 before receiving m_2, then its mirror should receive those two messages in that order as well. If a process in the distributed system fails, then its mirror takes over (starting another mirror process to back itself up). It first must catch up to the state at which the failing processor failed, which it does using messages it has saved. Then it continues as normal. Implementations of such schemes have been written [BBG83, PP83, BBG+89].

One of the problems with this simple form of checkpointing is that the mirror process has to restart its computation from the beginning until it catches up with the others. While this can be tolerated in a distributed environment, where the processes that are waiting for the mirror to catch up can be swapped off their machines, in a multicomputer environment it is much more wasteful. For example, on the Intel iPSC/860, users must allocate a portion of the processors for exclusive use. If $n-1$ of them wait while one catches up, then $n-1$ processors end up doing nothing for that period of time. The recovery takes just as long as doing the entire computation from the beginning, minus the message traffic. In the TARGON/32 system, the nodes periodically checkpoint themselves to prevent this problem of having one node restart from the beginning [BBG+89].

Another problem with this kind of checkpointing on some multicomputers is that with no job-swapping or multiprogramming, entire processors must be assigned as mirrors. This effectively cuts the size of the machine in half. Finally, the requirement that messages be transmitted to three places atomically is unreasonable for some machines without rewriting the operating system. In TARGON/32, extra hardware was added to facilitate this atomic transmission [BBG+89].

Optimistic Checkpointing Algorithms

In optimistic checkpointing algorithms [SY85, JZ89, Joh89], processors independently checkpoint themselves and maintain a log of the messages that they have received. They also maintain some information about the state of the processors that send them messages. This information is usually "piggybacked" on top of the messages themselves. When a processor fails, then all processors cooperate to decide to which state the failed processor should restore itself, and to which state they should restore themselves. This is decided by a complex distributed algorithm, after which processors may have to restore themselves to saved checkpoints, and then replay their computations using messages from their message logs. This is all to get the system to a plausible state that either did occur previously, or that could have occurred previously according to the information saved in the message logs.

One of the strengths of the optimistic algorithms, especially in widely dispersed distributed systems, is that during failure-free operation, they allow a large degree of autonomy. That is, processors can checkpoint and log messages whenever they want: They are not affected by the checkpointing and logging of other processors. This autonomy does not come for free—it is a source of much of the complexity in the recovery algorithm. On a multicomputer, autonomy is not quite as important as in a distributed system. For example, in a distributed system, processors may be separated by thousands of miles. Single-processor failures and partitions are likely to occur and do not affect the processing ability of the other processors. In a multicomputer, all the nodes are physically close, and more often than not, a single-processor or power failure brings down the entire machine. It makes sense for the processors to give up autonomy and cooperate more tightly than in optimistic checkpointing.

Finally, the recovery mechanisms for optimistic checkpointing are not useful for job-swapping. Rolling back and replaying can be time-consuming activities. Existing optimistic recovery methods are not appropriate for fast job swapping on multicomputers.

Consistent Checkpointing Algorithms

All consistent checkpointing algorithms stem from a paper by Chandy and Lamport [CL85]. In these algorithms, all processors cooperate at checkpoint time to define a *consistent state*, and write it to disk. Recovery is simple—all processors read their states from disk, and the computation may proceed. Consistent checkpointing seems to be the best for multicomputer checkpointing, as it does not overly penalize failure-free performance, and recovery is simple and fast enough to be used for general-purpose job-swapping. The main drawback of consistent checkpointing is that it requires all processors to cooperate at checkpoint time. This is not a big problem for multicomputers, where the processors are more tightly coupled than in general distributed systems.

There has been a great deal of research on consistent checkpointing algorithms [SK86, LY87, KT87, Ahu89, CT90, CJ91, KMBT91, LNP92]. Much of this work is discussed in Chapter 2, where we explain consistent checkpointing in detail, outline algorithms from the literature that are relevant to multicomputer checkpointing, and then describe the "Network Sweeping" algorithm, which is a new consistent checkpointing algorithm specifically designed for multicomputers.

1.4.4 Other Related Work

Many of the algorithms and optimizations described in the next two chapters stem from work in areas other than fault-tolerance and checkpointing. The Sync-and-Stop algorithm described below is an application of the two phase commit protocol [KS86] to distributed checkpointing. The latency-reducing optimizations of Chapter 3 were motivated by copy-on-write [TLC85, FR86], as well as shared virtual memory [LH89] and real-time, concurrent garbage collection [AEL88]. The compression algorithms mentioned in Chapter 3 are modified versions of those employed in a study of compressed log-structured file systems [BJLM92].

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes basic algorithms for checkpointing. One algorithm is given for multiprocessors, and three are described for multicomputers. Two of these three are new algorithms. Chapter 3 presents optimizations for the algorithms to improve the speed metrics of checkpoint time, overhead and latency. Chapter 4 presents the results of two checkpointing implementations, one on a multiprocessor, and the other on a multicomputer. All algorithms from Chapter 2 and all optimizations from Chapter 3 have been implemented and the results are discussed. Conclusions and future work are presented in Chapter 5.

Parts of this dissertation have been published previously. The initial portions of Chapters 2 through 4 were presented in [LNP90]. The rest of Chapter 2 has been published in [LNP91] and [LNP92].

Chapter 2

Checkpointing Algorithms

This chapter presents basic algorithms for checkpointing and restarting parallel programs on MIMD architectures. Combined with the optimizations in Chapter 3 they give a spectrum of efficient algorithms for checkpointing these machines. First, we describe an algorithm for checkpointing multiprocessors called "sequential checkpointing." Next, we describe consistent checkpointing in detail and present three consistent checkpointing algorithms for multicomputers. Two of these three algorithms are new. The last algorithm, called Network Sweeping, uses properties of wormhole routing networks present in all of today's multicomputers, to achieve superior performance. This algorithm is discussed in detail.

2.1 Checkpointing Multiprocessors

The only difference between checkpointing a uniprocessor and checkpointing a multiprocessor is that there are several CPU states to save to disk instead of just one. To establish a recovery point for a multiprocessor, the CPU states saved in the checkpoint should all be valid with respect to the state of memory saved in the checkpoint. That is, the processors all need to synchronize with each other at the memory boundary. Otherwise the checkpointer runs the risk of saving an invalid recovery point to disk.