Portable Fault-Tolerant File I/O

by

Igor B. Lyubashevskiy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author ....
Department of Electrical Engineering and Computer Science
June 1, 1998

Certified by ....

Volker Strumpen

Postdoctoral Associate

Thesis Supervisor

Certified by ....

Charles E. Leiserson

Professor

Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Portable Fault-Tolerant File I/O

by

Igor B. Lyubashevskiy

Submitted to the Department of Electrical Engineering and Computer Science
on June 1, 1998, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

The *ft*IO system provides portable and fault-tolerant file I/O by enhancing the functionality of the ANSI C file system without changing its application programmer interface and without depending on system-specific implementations of the standard file operations. The *ft*IO-system is an extension of the porch compiler and its runtime system. The porch compiler automatically generates code to save the internal state of a program in a portable checkpoint. These portable checkpoints can be recovered on a binary incompatible architecture. The *ft*IO system ensures that the state of stable storage is recovered along with the internal state of a program. The porch compiler automatically generates code to save bookkeeping information about *ft*IO's transactional file operations in portable checkpoints. We developed a new algorithm for supporting transactional file operations based on a *private-copy/copy-on-write* approach. In this thesis, we discuss design choices for the *ft*IO system, describe our new algorithm, provide experimental data for our prototype, and outline opportunities for future work with *ft*IO.

Thesis Supervisor: Volker Strumpen
Title: Postdoctoral Associate

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I would like to express my sincere gratitude to Dr. Volker Strumpen whose day-to-day involvement for the last two years made this project possible. He was involved in everything from showing me how to use LaTeX to countless brainstorming sessions. At the end, I made him suffer through all the drafts of this thesis.

I would also like to thank Prof. Charles Leiserson for his support, guidance, direction, and invaluable feedback. His sense of humor and Wednesday-night pizza provided the needed stimulus to keep working.

Likewise, I would like to note Edward Kogan, Dimitriy Katz, and Ilya Lisansky for proof reading an early version of this write-up and for their feedback.

I am greateful to Anne Hunter for her encouragement and motivation.

Last, but not least, I would like to express my appreciation to my girfriend, Marina Fedotova, and all my relatives for their care, support, and understanding.

This work has been funded in part by DARPA grant N00014-97-1-0985.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Much work has been done on the development of tools that allow programmers to quickly and effortlessly add fault tolerance to their existing code. Two general approaches have received the most attention: fault tolerance through redundancy and fault tolerance through checkpointing. Using hardware redundancy is the more costly approach, which can easily double (or triple) the cost of hardware required to complete the task. The benefit of redundant systems is that they are able to provide continuous availability. Checkpointing systems, on the other hand, do not require any additional hardware. They guarantee that, upon restart, a failed process can be recovered from the latest checkpoint and continue making progress with little work lost due to a failure. Checkpointing systems do not provide continuous availability, though.

There are two classes of checkpointing systems: system-level checkpointing and application-level checkpointing [13]. System-level checkpointing comprises hardware and/or OS support for taking periodic checkpoints of the running process. Most of those systems perform a periodic code dump to save the state of the memory and processor registers. Checkpoints of those systems, however, are often many times larger than the "live" state of the process. Also, the saved checkpoints are not useful in heterogeneous environments, because a failed process cannot be recovered on a binary-incompatible architecture.

With application-level checkpointing, a user process saves periodic checkpoints of itself. Because the application knows more about the data that need to be saved as well as their types, it can save only "live" data, as well as produce checkpoints that can be used to recover the process on any supported architecture. The porch compiler [18, 22] has been developed to automatically transform C programs into semantically equivalent C programs that are capable of saving and recovering from portable checkpoints. This technology can be exploited to provide fault tolerance in heterogeneous systems and may also be used for debugging or retrospective diagnostics [5]. To the latter end, a portable checkpoint can be used to restart or inspect a program on a binary-incompatible system architecture.

The problem of most checkpointing systems is that they only allow for checkpointing the internal state of a process. It is not sufficient, however, to recover the stack, heap, and data segments upon restarting the process. Most applications make use of stable storage, and the data in stable storage also need to be recovered. For example, suppose that a process reads a value from a file, increments it, and stores it back. Further, suppose that the process had saved a checkpoint before reading that value, and then failed some time after storing the incremented value. What will happen when the process is restarted, if only the internal state of the process was saved during the checkpoint? Upon recovery, the process will perform the same operations it executed after it saved the checkpoint—reading the value from the file, incrementing it, and storing the result back in the file. Clearly, this behavior is incorrect since the value is now incremented twice, and the state of the file is inconsistent with the state of the application. This example demonstrates the need for checkpointing systems that not only recover the internal state of a process but also revert the stable storage to a consistent state.

This thesis presents an extension to the porch compiler and its runtime system (**RTS** for short) to support portable fault-tolerant file I/O. My studies are based on my prototype implementation of the *ft*IO system. The *ft*IO system enhances the functionality of ANSI C file operations [10, 20] without changing its application programmer interface and without depending on a system-specific implementation of file operations. As the result, application programs using formatted file I/O can be made fault-tolerant by precompiling them with porch. Object code can be generated thereafter with a conventional C compiler. System-specific C library implementations can be used without modifications, because porch generates code based on the standard ANSI C function prototypes.

The problem we are tackling can be decomposed into three subproblems:

1. Providing transactional file operations for fault tolerance.
2. Checkpointing the state maintained by transactional files operations in a portable manner, allowing for recovery on a binary-incompatible machine.
3. Maintaining the data stored in a file in a machine-independent format.

When all three functionalities are present, we can save a checkpoint of a process, including the files it accesses, and recover the computation on a binary incompatible machine. The current implementation of the *ft*IO system provides the first two functionalities. Maintaining file data in a machine-independent fashion can be implemented by extending the porch compiler. The support for architecture-independent binary file I/O requires future research and is discussed in Section 7.

We provide transactional file operations to support *atomic transactions*. We consider an *atomic transaction* to be the execution of a program between two consecutive checkpoints. The program either *commits* its state during checkpointing or *aborts* at some point during the execution, in which case it can be recovered from the last checkpoint. Problems arise if a program *writes* to a file and aborts before the write has been committed. The reason is that the files are generally kept in nonvolatile storage. A value written before a crash is likely to remain in the file and will therefore erroneously be visible after recovery.

To avoid such incorrect behavior, I designed and integrated a new protocol with the porch compiler. This protocol is based on the *copy-on-write* approach [19], in which the entire file is copied upon the first write operation. Subsequent file operations are performed on the file replica. During checkpointing, the modifications are committed by simply replacing the original file with its replica. The protocol is based on a single global bit of information to ensure fault tolerance.

The *ft*IO system maintains bookkeeping information about the copy and the original file to preserve consistency with the remaining state of the application. This bookkeeping information comprises the protocol state of the file, e.g. clean or dirty, the file position, the mode in which the file was opened, an optional user buffer, the buffering policy (e.g. full, line, none), the `ungetch` state (up to 1 character, as required by ANSI C), and "meta information" such as the existence of the file and its name. It is important to checkpoint this information in a portable manner to facilitate recovery of an application on a binary incompatible machine. We utilize the porch compiler to automatically generate code for checkpointing the bookkeeping data in a machine-independent format by precompiling the *ft*IO runtime system itself with porch. This approach allows me to implement the runtime portion of *ft*IO as a shallow layer of wrapper routines for the complete set of file operations defined in the ANSI C standard.

The porch compiler and the *ft*IO system have been developed for the design of fault-tolerant systems based on the notion of portable checkpoints. This technology may be valuable for a variety of applications. Besides the most obvious use for providing fault tolerance in a network of binary incompatible machines, the technology can, for example, be used in the design of embedded systems. We envision the use of porch in embedded systems that use the C language for software development. Embedded systems are often built around commodity processors for which C (cross-) compilers and linkers exist or are relatively easy to port. The *ft*IO system requires that a subset of the ANSI C file operations is implemented, which is usually the case anyway.

Once a C compiler and I/O library are available, porch can be used in single-processor systems to capture the state of a computation, including the files it accesses, in a portable checkpoint. This checkpoint could be inspected on a binary incompatible machine, for example by using a debugger on a workstation. The capability of porch to automatically generate code for converting data representations can simplify the design of multiprocessor systems because machine-independent file data can be exchanged across binary incompatible processors without explicitly coding for portability. I believe that porch and *ft*IO could be a viable aid for software development and debugging as well as simplifying the software for fault-tolerant heterogeneous systems.

This thesis is organized as follows. I first give a brief review of the porch compiler technology in Chapter 2. In Chapter 3, I discuss design alternatives for the implementation of *ft*IO. In Chapter 4, I present the *ft*IO algorithm, which is in principle independent of the porch technology. Experimental results of the *ft*IO prototype are presented in Chapter 5, and more results available in Appendix A. I present an overview of related work in Chapter 6. Finally, I conclude with a discussion of the possibilities for future research in Chapter 7.

Chapter 2

The porch Compiler

The porch compiler [18, 22] is a source-to-source compiler that translates C programs into equivalent C programs capable of saving and recovering from portable checkpoints. *Portable checkpoints* capture the state of a computation in a machine-independent format, called *Universal Checkpoint Format*—**UCF**. The code for saving and recovering as well as converting the state to and from *UCF* is generated automatically by porch. This chapter presents an overview of the porch compiler and provides a brief summary of the techniques used to provide fault tolerance.

The porch compiler technology solves three key technical problems to making checkpoints portable.

Stack environment portability The stack environment is deeply embedded in a system, formed by hardware support, operating system and programming language design. A key design decision to implement porch as a source-to-source compiler has been the necessity to avoid coping with the system-specific state such as program counter or stack layout. It is not clear whether this low-level system state could be converted across binary incompatible machines. Instead, porch generates machine-independent source code to save and recover from checkpoints. At the C language level, variables can be accessed by their names without worrying about low-level details such as register allocation or stack layout done by the native compiler.

Data representation conversion Two issues of data representations are of concern: bit-level representations and data layout. Basic data types are stored in different formats at the bit level. The most prominant formats are *little endian* and *big endian*. Furthermore, different system designs require different memory alignments of basic data types. These determine the layout of complex data types such as structures. Consequently, all basic data types and the layout of complex data types are translated into a machine-independent format. The porch compiler generates code to facilitate the corresponding conversions automatically.

Pointer portability The portability of pointers is ensured by translating them into machine-independent offsets within the portable checkpoint. Since the target address of a pointer is not known in general at compile-time, porch is supported by its runtime system to perform the pointer translation during checkpointing and recovery.

To enable code generation, *potential checkpoint locations* are identified in a C program by inserting a call to the library function `checkpoint()`. For these potential checkpoint locations, porch generates code to save and recover the computation's state from portable checkpoints.

The porch compiler generates code for programs that compile and run on several target architectures without the programmer modifying the source code. This strategy does not prevent system-specific coding such as conditionally compiled or hardware-specific code fragments, but requires structuring a program appropriately. The means of hiding system-specific details from porch are structuring the program into multiple translation units and functions. The porch compiler employs interprocedural analysis to instrument only those functions that are on the call path to a potential checkpoint location. The functions that are not may safely be system specific. A reasonable convention would be to group all system-specific functions and data into a translation unit separate from the portable code. Only the portable portion of the code would be precompiled with porch. The nonportable portion could be different for any particular system, because it would not affect the state of the computation during checkpointing.

Chapter 3

The ftIO System Design Alternatives

In this chapter we discuss alternative implementations of transactional file operations in the context of checkpointed applications. We study the performance of *ft*IO by estimating the space and runtime overheads of these implementations. As the result of this study, we conclude that the most reasonable implementation for the *ft*IO system is the *copy-on-write* implementation of the *private-copy* approach.

The task of the *ft*IO system is analogous to the task of any transaction system. Indeed, we can think of all file operations between two checkpoints as belonging to one large transaction. Hence, an implementation of *ft*IO is in fact an implementation of a transaction system [6], where a transaction consists of the computation executed between subsequent checkpoints. Transactions are atomic operations. Either all operations within a transaction are executed, or none are. Instead of implementing a general transaction system, however, several factors allow us to argue for a simplified version. The two obvious ones are the absence of nested transactions and the availability of the internal state of the system during recovery (due to porch [18, 22]).

Transaction systems are generally classified as *undo-log* systems or *private-copy* systems [23]; the latter also called *side-file* [6] systems. Undo-log systems maintain a log of undo records, with an undo record for each operation within a transaction. If a failure occurs, the log is used to undo the modifications of an unfinished transaction. Private-copy systems maintain a private copy of all the data accessed by the operations of a transaction and update the original data only when the transaction commits. There are several implementations to optimize the private-copy approach. They differ in what is copied and when. Usually the trade-off is between performance, storage overhead, and complexity of the implementation. The design choice usually depends on the size of the transactions, the amount of data involved in the transaction, the performance and space requirements for the program (e.g., real-time response, limited stable storage), and the stable storage technology.

In the design of the *ft*IO system there were three important goals:

1. The support of heterogeneous computing environments.

2. The ability to generate fault-tolerant code automatically.

3. The efficiency (speed) of programs using the *ft*IO system.

I ensure platform independence of *ft*IO by implementing it as a set of shallow wrappers around the I/O functions from the standard C library. Since the *ft*IO runtime code is written entirely in the ANSI C language without any extensions (even without UNIX extensions), it can be used without modifications on any architecture.

Due to a rich set of analysis and transformation capabilities of the porch compiler, I am able to automate the translation of ANSI C programs for use with the *ft*IO system. That translation is completely transparent to the programmer and does not require any code to be changed manually.

In the following sections we identify the most efficient implementation of the *ft*IO system. To do so, we discuss the *undo-log* approach, the *private-copy* approach, and three possible implementations of the latter one, namely the *copy-on-write*, the *shadow-block*, and the *twin-diff* implementations. We conclude this chapter with a comparison of the performance and space overheads of different implementations.

3.1 Undo-Log Approach

This section describes the undo-log approach, a popular approach for implementing transaction systems. The assumption behind this approach is that the system fails rarely, and most transactions commit normally. Therefore, undo-log systems make all changes directly to the files that the changes affect and keep a separate log to undo the changes. If a transaction commits normally, the undo-log is simply discarded. In this section we discuss the performance and implementation of the undo-log approach in the context of checkpointed applications and conclude that this approach is not well-suited for the *ft*IO system.

The undo-log approach works well for small transactions. Large transactions, however, impose a serious time and space penalty. Since the undo-log approach requires logging of undo operations for every write operation, the cost of maintaining the undo-log is proportional to the number of write operations between subsequent checkpoints. For n writes between checkpoints, the runtime overhead is $\Theta(n)$ for committing the associated undo records. The overhead includes reading the original value at the write location and writing the undo record to the undo-log. The space overhead due to n writes is also of order $\Theta(n)$. Therefore, the runtime and space overheads are not bound by the size of the files. Instead, they depend on the time interval between checkpoints.

The other problem with the undo-log approach is that one must ensure that all undo records are committed (flushed) to stable storage before the corresponding file operations are committed to the disk files. If this is not the case, a system failure after a file operation is committed to the disk, but before a corresponding undo operation is written, can make recovery impossible. Buffering file opeartions in memory is a very effective file I/O optimization implemented by most ANSI C libraries. However, to implement the undo-log approach in ANSI C, one would have to sacrifice the benefits of buffering file operations by flushing every undo record to disk before performing the actual write operation. Undo records cannot be buffered in memory, since there is no guarantee that they will be flushed to disk before the corresponding write operations.

It is conceivable that one could reimplement ANSI C file buffering in *ft*IO to ensure that buffers with undo records are flushed before the corresponding buffers with writes. However, very often memory buffering of file I/O is also implemented on the level of the operating system. Ensuring that an operating system flushes buffers with undo records before it flushes the buffers with writes would require functionality beyond that provided by ANSI C.

The last potential problem with the undo-log approach is that support for renaming and removing files is not straightforward. Because all file operations are performed on the original files, remove and rename operations would remove and rename the actual files. Hence, undoing rename and remove operations is not straightforward with the undo-log approach.

3.2 Private-Copy Approach

This section discusses the private-copy approach, an alternative to the undo-log approach. Private-copy systems, unlike undo-log systems, do not modify files until a transaction is committed. Instead, they keep a separate *private* copy of the data and perform all operations on that copy. During the commit phase of the transaction, the original is reconciled with the copy. In this section, we discuss implementations and system trade-offs that affect the performance of the private-copy approach in practice. Namely, we will discuss three implementations of the private-copy approach: *copy-on-write*, *shadow-block*, and *twin-diff*. We will find that *copy-on-write* is the preferred implementation, since it attains the best worst-case performance and space overhead.

The space overhead of a private copy is bound by the size of the original file N plus the potential bookkeeping overhead. The runtime overhead of the private-copy approach is bound by $O(N)$. The constant factor hidden by O depends on the individual implementation. Note that both space and runtime overheads depend on the file size N in the private-copy approach.

Since the amount of data that must be copied is at most the total size of all files affected by the changes, private-copy systems are likely to perform better than undo-log systems in the context of relatively large transactions and relatively small files. Additionally, since the private copy is an ordinary file, ANSI C file buffering can be fully utilized during the execution of the program.

Another advantage of private-copy systems is that it is easier to maintain the transactional properties of operations such as renaming, removing, and creating new files. Indeed, since private-copy systems do not modify the original files, undoing renames, removals, and file creations involves no more than discarding private copies of the affected files.

Copy-on-write Implementation

Copy-on-write was first developed for virtual memory management in the Mach operating system [19]. It can be applied in the context of file I/O, where it may be viewed as an implementation technique of the private-copy approach. In fact, our *ft*IO system is based on a copy-on-write implementation.

Copy-on-write is an implementation of the private-copy approach where a private copy of a file is made upon the first `write` operation after the last checkpoint or after the program begins execution. All subsequent operations are performed on the replica until the next checkpoint is saved, when the original file is replaced with the replica. Copy-on-write is an optimization of the private-copy in the sense that a copy is only made when the first `write` occurs, as opposed to copying at the very first access, which may be an `open` or `read`.

The copy-on-write implementation suffers runtime overhead only during the first `write` operation after a checkpoint. All other file operations incur virtually no overhead, with the exception of *ft*IO-related bookkeeping. During the commit or recovery phase, the original file is replaced by the replica via a single `rename` operation. An atomic `rename` operation is used to this end, as provided by POSIX-compliant operating systems [16].

Copy-on-write incurs no overhead if a file is accessed by `read` operations only. If there are `write` accesses, copy-on-write performs best for environments with relatively infrequent checkpoints in the presence of a large number of write operations whose modifying accesses are scattered across the entire address range of a file. In this case, write accesses are expected to amortize the copy overhead, because infrequent block-copying of files utilizes disk technology and memory bandwidth very efficiently.

During the commit phase, the original is simply replaced with the replica via a single rename operation. Also, the files that were marked to be removed during the transaction are actually removed during commit.

Shadow-block Implementation

Replicating files may be considered prohibitive if either runtime is critical, such as in real time applications, or if the capacity of non-volatile storage is limited, as is often the case in embedded systems design. Moreover, one can argue that if a program exhibits spatial locality in file I/O, only parts of the file would be modified during a transaction. Hence, a straightforward algorithm might create replicas of the modified blocks only, instead of copying the entire file. This block-based algorithm is known as the *shadow-block* implementation [23], or *shadow-page* algorithm in [6]. It is an optimization of the copy-on-write implementation that is suited for use in operating systems. Files on the disk are typically fragmented into disk blocks, and the file system maintains bookkeeping information about the fragmentation. In such an environment, rather than copying an entire file upon the first write, only the block accessed by a write operation is copied. Each block of a file can be copied, committed, and recovered independently.

The shadow-block implementation is not well suited for use in *ft*IO, however, because *ft*IO is implemented at user level for portability reasons. At user level, the fragmentation of files is invisible. Besides adding another level of fragmentation at user level, a shadow-block implementation in *ft*IO would cause additional overhead during the commit phase, as we discuss now.

Fragmenting a file at the user level into several independent blocks prevents us from using a single rename operation to commit the changes. Instead, replicas of the changed blocks must be copied back into the original file (master file). Hence, during the course of a single transaction, each updated block is copied twice—first when the replica of the block is made, and second when the changes to the block are committed. In the worst case, when all blocks of a file must be copied during a single transaction, the performance penalty of the shadow-block implementation doubles the penalty of the copy-on-write implementation and suffers some overhead for the bookkeeping and added complexity.

Like the undo-log approach, the shadow-block implementation works best for small transactions, involving programs that exhibit strong spatial locality and use large files, because only few blocks are likely to be updated during a single transaction. In a larger transaction, however, more blocks are likely to be modified, and the performance is more likely to resemble the worst-case performance.

Twin-Diff Implementation

Another alternative for implementing *ft*IO is the twin-diff technique, popular for building distributed shared memory systems [26]. There, memory consistency must be provided at the smaller unit of basic data types rather than at the block level or, in case of distributed shared memory systems, at the level of virtual-memory pages.

A typical twin-diff implementation maintains replicas of pages (*twins*) in memory. During the commit phase, the number of copies is minimized by reconciling only the *diff*erences—the modified data—between the original page and the replica.

The twin-diff implementation seems appealing, because it allows for copying only blocks modified and minimizes the number of copies during the commit phase, which has been identified as the crucial disadvantage of the shadow-block implementation. Computing the diffs, however, requires reading both the original page and the replica. Then, relatively small, generally noncontiguous data items (the *diffs*) are written back into the file.

Due to the performance characteristics of the modern I/O systems, the twin-diff implementation is not well-suited for *ft*IO. First of all, during the commit operation, both the original file and the replica need to be read to compute the diff. Due to delayed disk writes, however, reads on modern architectures are more expensive than writes. Additionally, due to the block-oriented nature of disk I/O, there is usually no performance gain in writing only a small portion of a block instead of the whole block. Finally, since the commit phase of the twin-diff approach is not atomic, and the diffs are kept in memory, ensuring a correct recovery from crashes during commit operation is hard.

3.3 Comparison of Implementations

This section compares space and runtime overheads of the *undo-log*, *copy-on-write*, and *shadow-block* implementations discussed above. We are primarily interested in the worst-case analysis, since it provides upper bounds on the overheads. We will also show a comparative analysis of the *copy-on-write* and *shadow-block* implementations. Finally, we will show that for the modern disk I/O systems that are optimized for block transfer of data, the *copy-on-write* implementation is almost always preferable.

Below are the definitions of the variables used in the following discussion:

n	The number of write operations performed on a file
N	The number of bytes in the file
b	The number of blocks in the file
f	The ratio of the modified blocks to the number of blocks in the file
t_s	Startup time required for reading or writing data (moving disk head)
d	Incremental time (beyond startup time) required to read or write one byte
D	Incremental time required to read or write the entire file ($D = Nd$)
S	Space overhead
T	Time overhead

For simplicity, we assume that read and write times (t_s, d, D) are the same. Variables b and f apply only to the shadow-block implementation of copy-on-write approach. Since f is a fraction of the modified blocks, we have $1/b \leq f \leq 1$, given that at least one write operation has been performed on the file, and therefore at least one block is modified.

The overheads for n `write` operations for the different implementations are presented in Table 3.1. For the *undo-log* implementation, the space overhead is proportional to the number of `write` operations, since each `write` generates a new undo record. The time overhead is $n(t_s + \Theta(d))$ since each `write` involves writing a separate undo record. The space overhead of *copy-on-write* is N for the copy of a file, and the time overhead is $2(t_s + D)$ to copy the whole file upon the first `write`. The space overhead of the *shadow-block* implementation is proportional to the number of blocks modified, and the time overhead is the overhead for copying each modified block twice, once to generate a replica and once to commit the relpica.

Implementation	Space overhead S	Time overhead T
ungo-log	$\Theta(n)$	$n(t_s + \Theta(d))$
copy-on-write	N	$2(t_s + D)$
shadow-block	fN	$4f(b \cdot t_s + D)$

Table 3.1: Time and space overheads for *undo-log*, *copy-on-write*, and *shadow-block* implementations.

Table 3.2 presents the worst-case space and time overheads for the three implementations. The $\Theta(n)$ space overhead of the *undo-log* implementation is not bound by the file size N but by the number of file operations between checkpoints. In particular, $n \gg N$ represents a possible scenario. In contrast, the space overhead of *copy-on-write* is always $S = N$. Also, the runtime overhead $T = n(t_s + \Theta(d))$ for committing n undo records could be substantially larger than $T = 2(t_s + D)$ for the *copy-on-write* implementation. The worst case senario, for which $f = 1$ when all blocks are modified, yields a higher runtime overhead for the *shadow-block* implementation than for the *copy-on-write* implementation: $4(b \cdot t_s + D) > 2(t_s + D)$, while both implementations imply $S = N$.

Besides the worst-case analysis, it is instructive to compare the overheads of the *copy-on-write* and *shadow-block* implementations, as there exists a ratio f^* of modified blocks, where both implementations incur the same runtime overhead. Comparing f^* to f can help choose the best implementation for a particular problem. Specifically,

Implementation	Space overhead S	Time overhead T
ungo-log	$\Theta(n)$	$n(t_s + \Theta(d))$
copy-on-write	N	$2(t_s + D)$
shadow-block when $f = 1$	N	$4(b \cdot t_s + D)$

Table 3.2: Worst case time and space overheads for *undo-log*, *copy-on-write*, and *shadow-block* implementations.

for $f > f^*$, the *copy-on-write* implementation is faster, whereas for $f < f^*$, the *shadow-block* implementation is faster. We find that

$$T = 4f^*(bt_s + D) = 2(t_s + D) \quad \text{for} \quad f^* = \frac{1}{2}\frac{t_s + D}{bt_s + D}.$$

Depending on the disk technology used and the file size, there are two boundary cases:

1. If the startup time t_s is an insignificant part of the time to write one block of data $t_s + D/b$, then $b \cdot t_s \ll D$, and both implementations incur the same runtime overhead for $f^* = 1/2$. In this case, the *copy-on-write* implementation would be preferred only if at least half of the blocks are modified during the time between two checkpoints.

2. If the startup time is large, or the file is small, then $t_s \gg D$ and thus $f^* = 1/(2b)$. Since $f \geq 1/b$ by definition, then $f^* < f$ for all problems. Therefore, *copy-on-write* always incures less runtime overhead.

An important observation is that the larger the time between checkpoints is, the larger f becomes, and the larger the overhead the *shadow-block* implementation incurs. It is not clear, however, what the optimal number of blocks b for a particular problem is. On one hand, the runtime overhead $T = 2f(b \cdot t_s + D)$ for the *shadow-blok* implementation grows as b increases, but the fraction of modified blocks f may decrease with larger b, thereby reducing T. According to the experimental data (see Section 5), however, the difference in performance between a user-level *shadow-block* and *copy-on-write* implementations is likely to be insignificant. As a result of this discussion, we used the *copy-on-write* implementation in the *ft*IO system, which is also substantially simpler than the other alternatives.

Chapter 4

The ftIO Algorithm

This chapter describes the algorithm underlying the *ft*IO system. In Section 4.1, we introduce the *ft*IO finite automaton and describe its operation. We also discuss the *ft*IO commit operation and the *ft*IO recovery phase. In Section 4.2, we conclude by presenting an effective optimization to the *ft*IO's *copy-on-write* implementation for the case when files are opened in the append mode.

The *ft*IO algorithm is a private-copy/copy-on-write design for checkpointing systems. Checkpointing systems are characterized by alternating phases of *normal execution* and *checkpointing* unless a failure occurs. The *ft*IO algorithm is based on a finite automaton built upon a set of *ft*IO states for each file accessed by the application. State transitions occur if certain file operations are executed. All *ft*IO file operations are implemented as wrappers around the standard ANSI C file operations. These wrapper functions maintain for each file a data structure that contains its state. The state maintained for each file includes the name of the file and its replica, the *ft*IO state (see below), the mode in which the file is opened, the buffering policy for the file, the location and size of an optional user's buffer, a push-back character, `eof` and `error` flags, and the current position in the file.

*ft*IO States

Each file is associated with three orthogonal *ft*IO states. A file may be *dirty* or *clean*, it may be *open* or *closed*, and it may be *live* or *dead*.

clean/dirty A file is *clean* if it has not been modified since the last checkpoint (or since the beginning of the execution, before the first checkpoint); otherwise, it is *dirty*. By this definition, files that do not exist (and have not been removed since the last checkpoint) are clean. In a private-copy implementation, no replicas exist for clean files.

open/closed A file can be either *open* or *closed*. By default, all files are considered closed, unless they are explicitly opened. In particular, files that do not exist are closed by this definition.

live/dead A file can be scheduled for removal, in which case it is dead. Otherwise, it is alive.

Besides the file-specific states, a global boolean *ft*IO state is maintained.

FIN (for *finished*) is a bit used to record the success of the *ft*IO commit operation. It will be explained in detail in Section 4.1.

*ft*IO File Operations

We distinguish ANSI C file operations from *ft*IO file operations. The latter define the input alphabet of the *ft*IO finite automaton as listed below:

create creates and opens a file. If the file exists already, it is erased before being created again. This *ft*IO file operation corresponds to the ANSI C file operation `fopen` with opening mode `"w"`, or `"a"` if the file does not already exist.

createTmp creates and opens a temporary file, which is to be removed automatically when closed. By definition, this file has a unique name. The corresponding ANSI C operation is `tmpfile`.

open opens an existing file. The corresponding ANSI C file operation is `fopen` with opening mode `"r"`, or `"a"` if the file exists.

write modifies the contents of an open file, including appending data to that file. The corresponding ANSI C file operations are `fprintf`, `vfprintf`, `fputc`, `fputs`, `putc`, and `fwrite`.

close closes an open file. This operation corresponds to the ANSI C file operation `fclose`.

remove removes a closed file. This operation corresponds to the ANSI C file operation `remove`.

renameSrc (to B) moves an existing and closed file to location B. This operation corresponds to a part of ANSI C operation `rename` and is applied to the file that is the *source* of rename operation. The definition of the input alphabet for *ft*IO's finite automaton requires splitting the ANSI C file operation `rename` into two parts, renameSrc and renameDst, because a `rename` affects the source and the destination files differently.

renameDst (from A) replaces a closed (possibly nonexisting) file with a file from location A. If this file exists, it is overwritten with A. This operation corresponds to that part of the ANSI C operation `rename`, which affetcs the *destination* file of the `rename` operation.

A note concerning portability: ANSI C does not define the behavior of `rename` if the destination file exists. According to the UNIX specification, however, the `rename` operation atomically replaces the destination file with the source file [16]. The *ft*IO system implements the UNIX-like behavior for `rename` and requires the UNIX-like behavior from the native C library.

Note that ANSI C operations that read data, check for an error or the end of file, change the current position in a file, and change the buffering mode do not have an equivalent *ft*IO operation. Those ANSI C operations do not change the state of a file and therefore do not cause a state transition of the *ft*IO finite automaton.

4.1 The *ft*IO Finite Automaton

In this section we explore the *ft*IO finite automaton. We start by describing the the operation of the *ft*IO system for live files. Then, we present the complete finite automaton by broadening the context to include the dead files and the support for temporary files as well as file operations `remove` and `rename`. During the discussion of the *ft*IO commit and recovery phases, we introduce the global FIN bit and investigate the idempotency of the commit operation. We conclude with an informal argument about the correctness of the *ft*IO algorithm.

Figure 4-1 shows the transition diagram of the *ft*IO finite automaton based on a subset of the *ft*IO states and file operations that affect live files only. This subset does not contain file operations remove, renameSrc, renameDst, and createTmp. The start state of the automaton is clean & closed; the accepting states after a commit operation are shown in black.

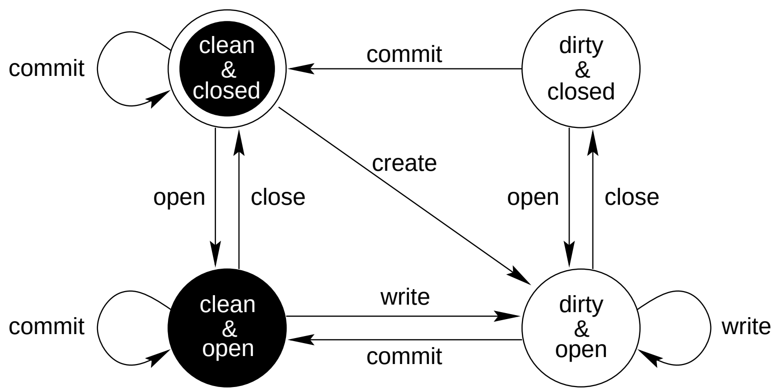

Figure 4-1: The "live" subset of of transition diagram for the *ft*IO finite automaton.

Opening and closing files does not affect the initial clean/dirty state. Open files can be written, causing the file to become dirty. In our *copy-on-write* implementation of *ft*IO, a clean file is copied upon a write, and the write, as well as all future writes, are performed on the replica. Upon file creation the dirty state is set.

If a file is clean, no replica of the file exists, and the file has not been modified since the last checkpoint. Therefore, all operations are performed on the actual file until a write operation is performed on that file. Upon the first write operation, a private copy of the file is generated, and the file becomes dirty. Henceforth, all operations are performed on the replica of the file until the next checkpoint is saved, when the changes to the file are committed. During the commit operation, the original file is replaced by its replica, and the file becomes clean again.

Temporary Files, Removal, and Renaming

ANSI C supports renaming and removing files. Additionally, it defines temporary files, which are removed automatically when closed. The live/dead state is introduced in *ft*IO to support these features. The dead state indicates that a file, including its replica, must be removed during the commit phase, provided the file is in a closed state. If the file is open during the commit phase, then the file is treated as if it were alive.

Figure 4-2 shows the complete transition diagram of *ft*IO's finite automaton. It incorporates three additional states besides the four "live" states to support the *ft*IO operations remove, renameSrc, renameDst, and createTmp.

The states "clean & open & dead" and "dirty & open & dead" are used for temporary files (created with createTmp) while they are open. All files that are removed, renamed, or were created as temporary and are closed are in the "closed & dead" state. Files in state "closed & dead" can be clean or dirty.

When a temporary file is created, that file is created as dead. Consequently, when the temporary file is closed, it will be removed during the commit operation. While it is open, the live/dead state is ignored, and the temporary file is treated as an ordinary file.

The live/dead state enables checkpointing removed and renamed files. If a remove operation is applied to a file (it must be closed to be removed), the file's state becomes dead. When a file is marked dead, the `fopen` and `freopen` file operations treat that file as if it does not exist, because the file is effectively removed. The file will be actually removed, however, only during the commit phase.

If a rename operation is invoked (both source and destination files must be closed

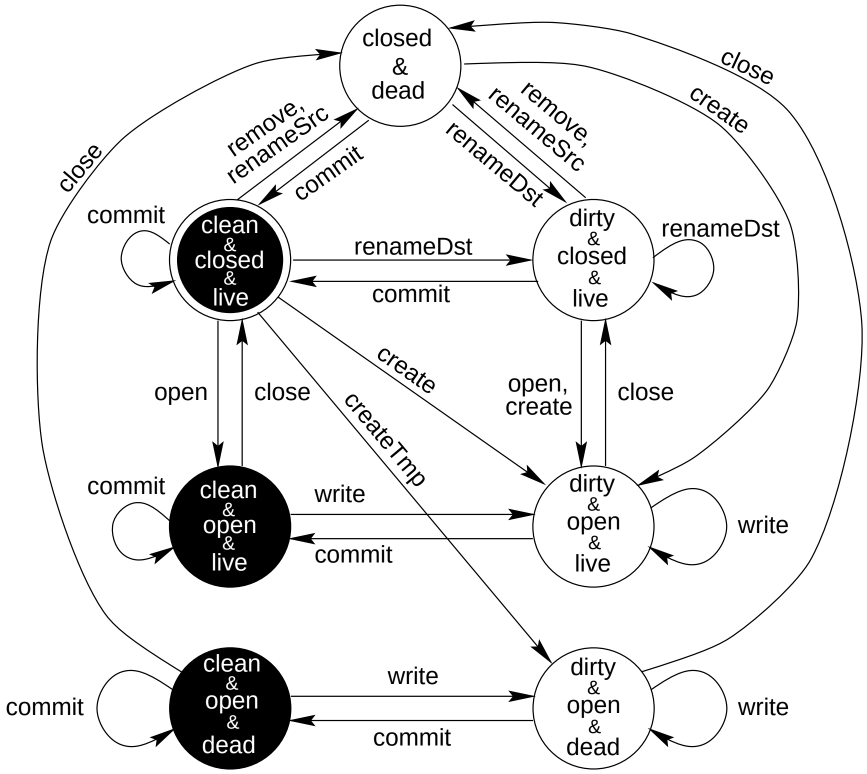

Figure 4-2: Complete transition diagram of the *ft*IO finite automaton.

for that), the source file's state becomes dead. We also need to ensure that the replica of the destination file contains the data from the source file. Therefore, we either rename the replica of the source file to be the replica of the destination file, if the source file is dirty, or copy the source file to become the replica of the destination file, if the source file was clean. Also, the destination file is marked dirty because it has been modified, and the destination file is marked live, if it is dead before.

Checkpointing

We separate the checkpointing process into two phases. First, the porch RTS saves the volatile state, including register and memory values. Second, the *ft*IO system commits the changes made to the files. The two phases are accompanied by a third action, the maintenance of the FIN bit. Figure 4-3 shows the sequence of the checkpointing phases.

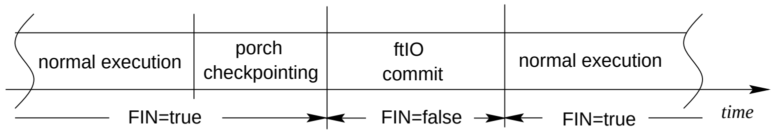

Figure 4-3: Execution phases.

Upon checkpointing, the volatile state of a process is saved by the porch RTS in *UCF* format in a temporary checkpoint file. This checkpoint file includes the FIN bit, explained in detail below. When the volatile state is gathered in the temporary checkpoint file, it replaces the previous checkpoint file by means of the atomic `rename` operation of the ANSI Standard C library [16], thereby committing the checkpoint.

After the porch RTS has saved the checkpoint, the *ft*IO system performs commit operations for all files. The **commit operation** affects files in the following way:

> For each dirty file, the original file is replaced with its replica, and the file state transitions to clean. If a file is dead and closed, however, both the actual file and the replica (if one exists) are removed irrespective of the clean/dirty state.

The FIN bit ensures the atomicity of the *ft*IO commit operation. It occupies a bit in the checkpoint. The porch RTS initializes a new temporary checkpoint file with the FIN bit value set to *false*. At the end of the *ft*IO *commit* phase, the FIN bit is set to *true* in the checkpoint file. This happens in the actual checkpoint file, which was created during the porch checkpointing phase and contains the internal state of the program. In Figure 4-3 the FIN bit represents a global state, which is defined as *the value of the* FIN *bit in the last checkpoint file*. Therefore, the FIN bit becomes *false* in Figure 4-3 only after the porch checkpointing phase, when the new checkpoint file with the FIN bit set to *false* is committed by replacing the previous checkpoint. Also note that writing the FIN bit does not require an atomic disk-write operation. Instead, we only must ensure that the FIN bit has been written to disk before the first write operation after checkpointing.

Failure Cases and Recovery

Figure 4-3 above shows the two checkpointing phases embedded in periods of normal execution. Failures may strike during any of these phases. There are, however, only two distinguished failure cases, which simplifies reasoning about the correctness of the *ft*IO algorithm.

1. If the application aborts due to a failure during the *normal execution* or during the porch *checkpointing* phase, recovery is based on the previous checkpoint. Since all modifying file operations have been performed on private copies, the original files are untouched. Because none of the file modifications have been committed yet, recovery from the previous checkpoint involves only discarding the replicas.

2. If the application aborts during the ft*IO-commit* phase, only a subset of the files may have been committed before the failure. In this case, the previous checkpoint has been replaced with the new checkpoint already, and recovery will restore the computation from the new checkpoint. Files are recovered simply by executing the *ft*IO commit operations during the *ft*IO recovery phase before returning control to the application. Since committing files involves no more than replacing the original files with their replicas, those files that have not been committed due to a failure can be committed before the application continues. Files that have been committed already remain untouched. The *ft*IO commit code does not even have to be changed because `rename` and `remove` operations are idempotent. These operations are idempotent because only the first call to `rename` or `remove` will succeed, and the following calls fail quietly. When the recovery phase is finished, the FIN bit is set to *true* in the checkpoint file, just as it would be set after the commit phase.

Recovery, analogously to the checkpointing process, is split into two phases:

1. During the porch recovery, the volatile state of a computation is restored, and the FIN bit is read from the checkpoint file, which determines the mode of the

*ft*IO recovery.

2. Files are recovered during the *ft*IO recovery phase.

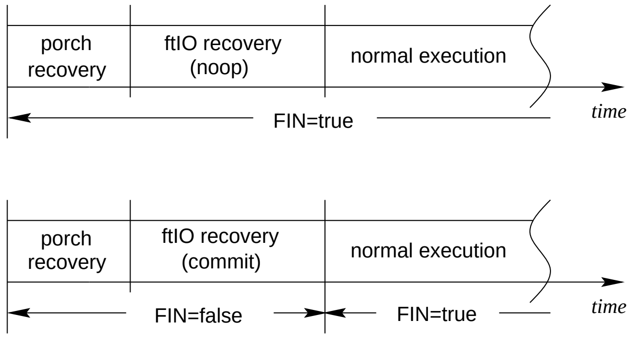

Figure 4-4: Recovery phases if failure occurred during normal execution or during porch checkpointing (top), and if failure occurred during *ft*IO commit (bottom).

The FIN bit serves to distinguish the two modes of recovery. Figure 4-4 shows the values of the FIN bit for both modes; cf. Figure 4-3. In the following, we argue informally why the *ft*IO algorithm works due to the FIN bit. Recall from the description above that the FIN bit is initialized to *false* within a new temporary checkpoint file. It is only changed to *true* when the *ft*IO commit phase is finished.

The value of FIN in the checkpoint is *true* upon recovery, if the checkpointing process succeeded. In this case the *ft*IO recovery phase is a `noop`, as shown at the top of Figure 4-4. Only if a failure occurs during the *ft*IO commit phase, does the FIN bit in the checkpoint remain *false*. In this case, files are committed during the *ft*IO recovery phase by executing the commit operation (see page 33). Since `rename` and `remove` are idempotent, the subset of files that have been committed before the crash will not be affected during recovery.

The only failure case remaining to be discussed is a failure during recovery, cf. Figure 4-4. No special treatment is required for this case. Since during recovery either the idempotent *ft*IO commit operation is executed or a `noop`, the recovery phase can safely be executed again to recover from a failure during recovery. No distinction is necessary as to whether the failure occurs during the porch recovery phase or the *ft*IO recovery phase.

To ensure that all files are committed correctly after the program has ended, porch saves the very last checkpoint after the `main` routine in the application code returns. That last checkpoint does not contain any state from the application, but only the state of *ft*IO. Hence, if the process fails during the *ft*IO commit operation, the program commits all files after recovery and terminates.

4.2 Append Optimization

According to [17], many long-running scientific applications store data by appending them to an output file. The analysis in Section 3.3 shows that the append scenario presents the best case for the *shadow-block* implementation, since only a few blocks at the end of the file are modified during the time between checkpoints. It is straightforward, however, to optimize the *copy-on-write* implementation for this case. I implemented the append optimization for *ft*IO that avoids copying a file upon the first write and performs therefore at least as well as the *shadow-block* implementation in the append scenario. The experimental results, presented in Section 5, confirm the claim for a significant performance improvement due to the append optimization.

The append optimization is based on the idea that instead of generating a replica of the original file upon the first write, a temporary file is created, and all writes before the next checkpoint are appended to this temporary file. During the *ft*IO commit operation, this temporary file is appended to the original file. Since the original file is not modified until the next checkpoint, discarding the temporary file is sufficient to recover from failures that occurred during execution.

Unlike renaming or removing replicas, as necessary to commit in the unoptimized cases, appending the temporary file to the original is not an idempotent operation. The *ft*IO recovery described in Section 4.1, however, requires the commit operation to be idempotent. The *ft*IO system stores the size of the original file in the checkpoint and truncates the original file to the recorded size before appending the temporary file to ensure idempotency. Hence, if a failure occurs while appending the temporary file, recovery involves first undoing the aborted append and then reexecuting the commit operation for the file.

If a file, opened in `append` mode, is closed and then reopened in a different mode, a replica is generated by concatenating the original file with the temporary file containing the appended data.

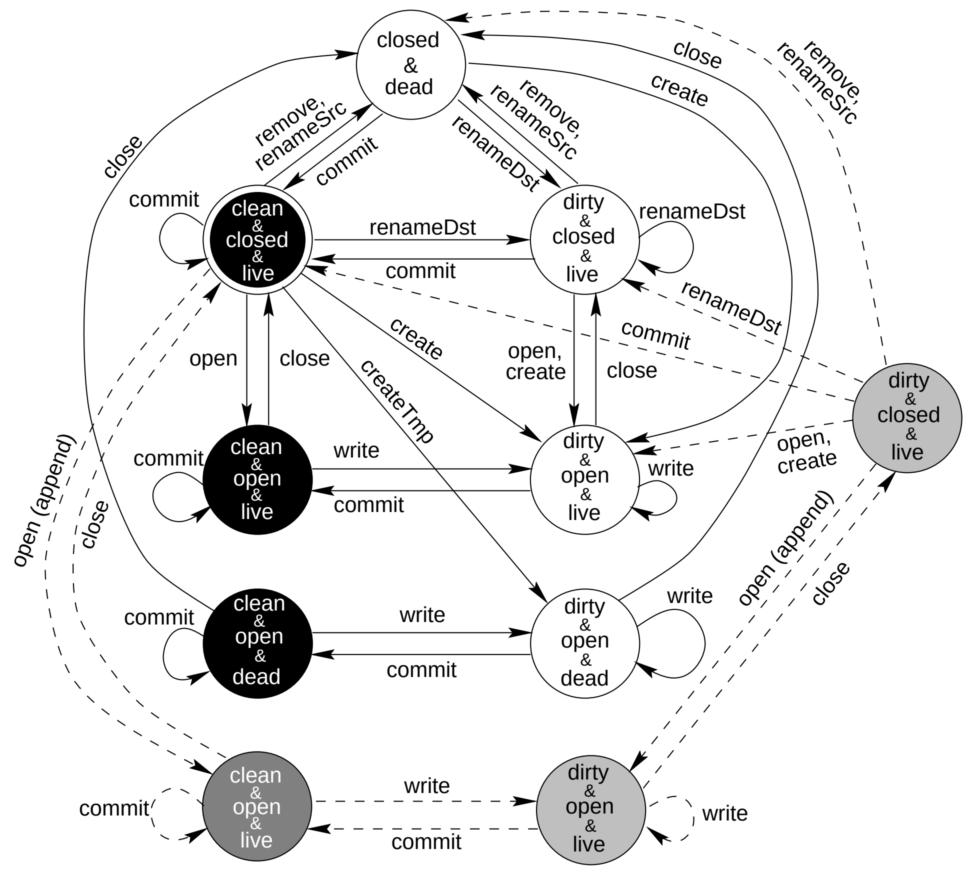

Figure 4-5: Transition diagram of the *ft*IO finite automaton including append optimization.

Figure 4-5 shows the transition diagram for *ft*IO'a finite automaton that includes the append optimization. The grey states and dashed transitions are added to the finite automaton presented in Figure 4-2. No other changes are necessary.

Chapter 5

Experimental Results

We designed three synthetic benchmarks to exhibit the performance characteristics of our *copy-on-write* implementation of *ft*IO: a read benchmark, a write benchmark, and a random write-access with spatial locality. We also present the performance of a molecular dynamics code to show that the overhead of checkpointing a scientific application that writes a large amount of simulation results to a file is reasonably small. All runtimes presented in this chapter were gathered on a Sun Sparcstation10 with checkpointing to local disk. Appendix A presents data for other machines.

Sequential Access: read and write

Figure 5-1 presents accumulated runtimes of the read benchmark and the write benchmark in the presence of checkpointing compared to the original, not porchified code. The read benchmark *reads* consecutive bytes from a file. The write benchmark *writes* (appends) bytes to a file. For both benchmarks, the interval between checkpoints is 2 seconds to exaggerate the checkpointing overhead. The runtimes reported are accumulated over the bytes read or written. Therefore, the runtime corresponding to a particular number of reads or writes includes the number of checkpoints saved during the period of executing that particular number of file operations.

Figure 5-1 shows that the overhead of the checkpointed version of the *read*-benchmark is due to two factors. First, checkpointing the program's state introduces

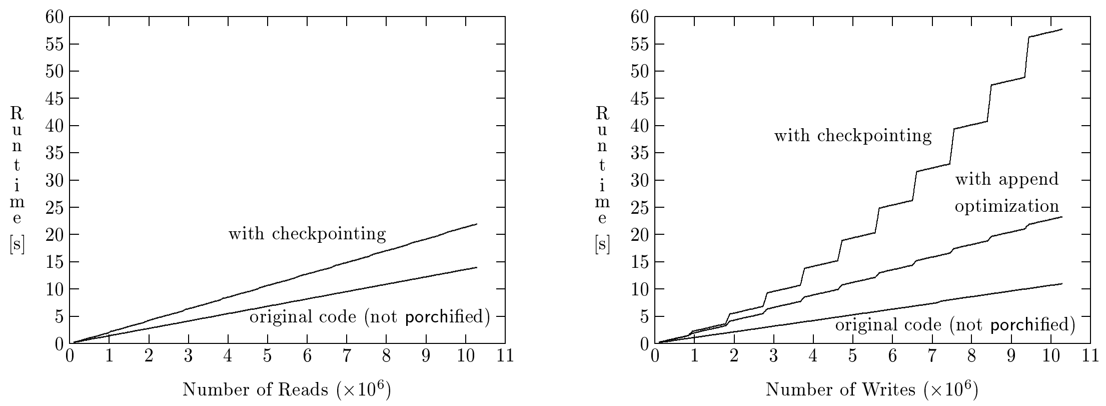

Figure 5-1: Performance of reading (left) and appending (right) 10 Mbytes of data from and to a file. Checkpoints are taken after about 10^6 file operations in the experiments marked "with checkpointing" and those marked "with append optimization."

small jumps when saving the internal state. Second, the overhead of the files operation wrappers leads to a slightly larger slope of the runtime curves between checkpoints. Since no write operations are performed in the read benchmark, no cost is incurred due to replicating the file.

The checkpointed version of the *write* benchmark, however, exhibits the expected overhead due to file replication upon the first write after a checkpoint has been saved. Each step in that curve corresponds to the overhead of file replication. The monotonically increasing step size of the checkpointed version without append optimization is a result of the growing file size. Clearly, file replication dominates the checkpointing overhead. The curve of the checkpointed version with the *append optimization* exhibits a constant step size, which corresponds to checkpointing the program state and copying the appended data from a temporary file to the original file. The step size is constant, because the same amount of bytes is appended between subsequent checkpoints, taken at equal intervals.

Random Access

Figure 5-2 presents runtimes of our third benchmark. This benchmark analyzes the performance of *ft*IO with file operations randomly distributed over the address range of a file, and where file accesses exhibit spatial locality. Spatial locality is incorporated by computing the location of a file access to be normally distributed around the location of the previous file access with a standard deviation of 10.24 KBytes. This value corresponds to 0.1 % of the total file size of 10 MBytes. Our benchmark performs 10^6 file operations, 90 % of which are `read` operations, and 10 % are `write` operations. Ten checkpoints are taken at equal intervals during the execution of the benchmark.

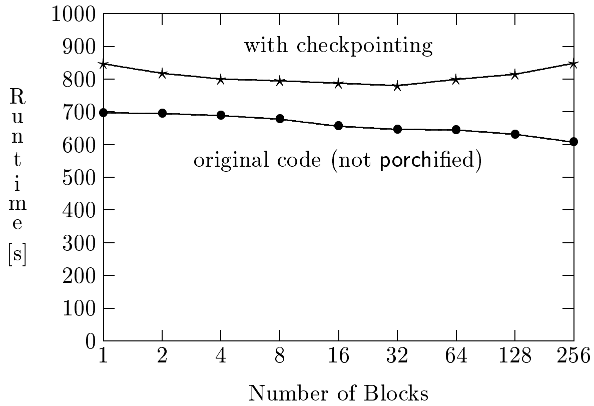

Figure 5-2: Runtimes of random file accesses with spatial locality. The variation of the number of blocks simulates the behavior of a shadow-block implementation with different block sizes.

Figure 5-2 includes runtimes of the *copy-on-write* implementation for different numbers of blocks. We simulate a *shadow-block* implementation by splitting the 10 MByte file into files (blocks) of equal size and accessing these blocks with our random strategy as if they occupied a contiguous address space. This simulation models a lower bound for the *shadow-block* implementation. By physically partitioning the data set into separate files we avoid copying of blocks during the commit operation. Instead, each modified file (block) is committed by renaming.

The runtimes in Figure 5-2 indicate that the *copy-on-write* implementation is more reasonable than the *shadow-block* implementation. As the number of blocks increases from 1 to 32, performance improves by 8 % only. For larger numbers of blocks, performance degrades, primarily due to the cost of the `rename` operations.

Another observation involves a comparison of the runtimes in Figure 5-2 with those in Figure 5-1. Accessing a file randomly (10^6 accesses) is almost three orders of magnitude more expensive than contiguous read or write accesses, even with the spatial locality of the accesses. This allows us to argue that modern disk I/O systems have a high startup time for disk reads and writes.

Finally, the performance of the benchmark without checkpointing in Figure 5-2 improves as the number of files (blocks) increases. We attribute this behavior to the file caching mechanisms in the Solaris operating system, which seems to be optimized for smaller files.

Molecular Dynamics Code

Figure 5-3 presents the cumulative runtime for a 2D short-range molecular dynamics code, written by Greg Johnson, Rich Brower, and Volker Strumpen. The code outputs the simulation results, including potential, kinetic, and total energy of the particle system (a total of 53 bytes) every 10 iterations to a file. The period between checkpoints is set to 20 minutes. The program generates approximately 400 bytes of data between checkpoints, and the total size of the output file after 60, 000 iterations is 150 Kbytes.

Presented are the accumulated runtimes for the original code and the checkpointed code with the append optimization. We observe that the overhead of checkpointing, including the *ft*IO overhead, is less then 6 % for any number of iterations.

The small steps in the runtime curve "with checkpointing" correspond to the checkpointing overhead, which is approximately 60 seconds each on Sun Sparcstation10. Hence, even as the file size increases, the append optimization keeps the commit cost constant. The runtime overhead of each checkpoint is 75 % due to checkpointing the internal state of the program by the porch RTS [22]. The difference of the slopes of the two curves in the regions between checkpoints corresponds to the

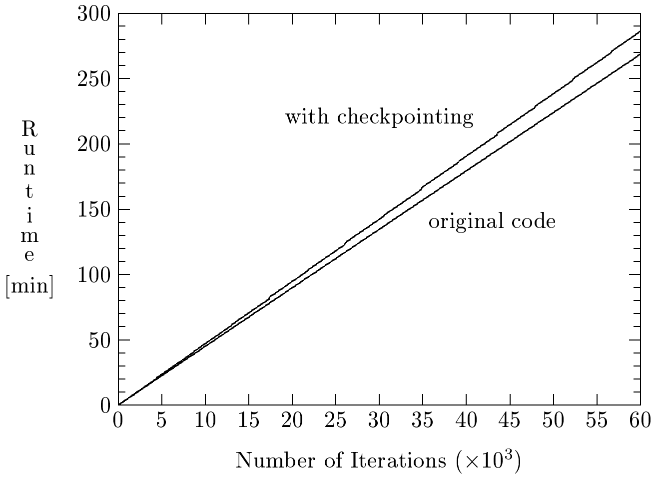

Figure 5-3: Runtimes of 2D short-range molecular dynamics code for 5,000 particles. The checkpointing interval is 20 minutes.

runtime overhead of the *ft*IO bookkeeping. The *ft*IO overhead per file operation is insignificant, because the difference of the slopes is marginal.

Chapter 6

Related Work

I are not aware of any checkpointing system that implements portable, fault-tolerant file I/O. According to Gray [6, p. 723], "a large variety of methods exists for moving data back and forth between main memory and external storage." To my knowledge, the *ft*IO algorithm is new in that it uses only an atomic rename operation and a single state bit to implement transactional file operations.

The libfcp library [24] provides support for transactional file operations. This library implements the *undo-log* approach, using a two-phase commit protocol to commit a transaction. Besides requiring explicit denotation of a program to mark the start and end of a transaction, libfcp does not provide portability. In combination with libckp [24, 25] and libft [8, 24], libfcp can be used to checkpoint the state of persistent storage in the context of checkpointed applications in homogoneous environments.

Paxton [14] presents a system that uses a combination of the *shadow-block* and *undo-log* techniques to support transactional file I/O. He uses the *shadow-block* technique, implementated at the level of the operating system, to store the data modified during a transaction. An undo-log (intentions log in [14]) is kept to facilitate bookkeeping of the *shadow-block* data structures. This system is not designed for checkpointing and is not portable, since it requires modification of the operating system.

Eden is a transaction-based file system [9], which is not, however, designed for fault tolerance. It supports concurrent transactions by maintaining multiple versions of a file. In the presence of only one process, Eden's algorithm is similar to *ft*IO's *copy-on-write* implementation since it creates a replica for all modified files during a transaction. The implementation of Eden is by no means portable, since it is tightly integrated with the operating system.

A number of checkpointing systems exist for homogeneous environments, such as libckpt [15], a supplement of the Condor system [11, 12], and the CLIP system for Intel Paragon [4]. Not only are these systems tied to a particular machine architecture, they also do not support transactional file operations. By saving only the name of the file and the current file position in their checkpoints, these systems limit the applications to read-only or write-only file I/O.

There are two approaches to migration across binary incompatible machines, the Tui system [21] and the work on Dynamic Reconfiguration [7]. Neither of these systems supports transactional file operations. The Dome system [1] provides architecture-independent user-level checkpointing. However, it does not support transactional file operations either.

The work on I/O benchmarking by Chen and Patterson [2, 3] inspired the development of our synthetic benchmarks for transactional file-I/O. Likewise, the survey of I/O intensive scientific applications [17] increased our understanding of the trade-offs in the *ft*IO design.

Chapter 7

Conclusion and Future Research

I have presented the *ft*IO system, which extends the porch compiler by providing portable, fault-tolerant file I/O. Due to the *ft*IO system, C programs that use formatted file I/O can be rendered fault-tolerant in a transparent fashion by precompilation with porch. Furthermore, no changes are necessary to the system-specific C library.

I have analyzed performance characteristics of the *undo-log* approach and the *private-copy* approach, including three implementations of the latter: *copy-on-write*, *shadow-block*, and *twin-diff*. As the result of the analysis I found that a *copy-on-write* implementation of the *private-copy* approach is the most reasonable choice for the *ft*IO system. I have introduced the *ft*IO algorithm, which provides transactional file operations using only a single state bit. The *ft*IO runtime system is written entirely in ANSI C, and the *ft*IO code itself is compiled with porch, thereby ensuring that a checkpoint contains all *ft*IO-state information in a machine-independent format.

I provided experimental results measuring the performance overhead of the *ft*IO system under sequential and random access workloads and comparing it with the performance of the *shadow-block* implementation of the *private-copy* approach. The experiments show that the *copy-on-write* implementation is the most reasonable choice for the *ft*IO system. Moreover, the performance of the molecular dynamics code indicates that the *ft*IO system incures relatively low runtime overhead for this and similar applications.

Future research can extend the *ft*IO system in several directions. It can be modified to support other file operations besides the ones from the ANSI C library, to incorporate the support for architecture-independent binary file I/O, and to provide new functionalities such as fault-tolerant socket I/O.

Architecture-independent Binary File I/O

To allow true architecture-independent checkpointing of the program state, data writte to a file on one machine should be readable from the file on a binary-incompatible machine after recovery. In the current *ft*IO prototype the only way to ensure that data can be read on a different architecture is to use formatted file I/O [20], which implies that all data must be stored in ASCII format. However, ANSI C supports binary file I/O via the `fread` and `fwrite` operations, which allow a program to read and write blocks of binary data from and to the main memory.

The problem of architecture-independent binary file I/O resembles the problem of storing and recovering from architecture-independent checkpoints, since in both cases the data must be converted to and from an architecture-independent intermediate format. The two problems differ, however, in that the data type—which is required from a proper conversion—is unknown when the binary data is read from a file. I envision two ways to approach the problem. First, one can store the data together with its type for binary file I/O. Second, with the assistance of a compiler, one can attempt to infer the type of the data read. I believe that the second approach is preferable. It avoid the overhead of storing the type information with the data.

Supporting Multiple Streams to a File

ANSI C [20, 10] and POSIX [16] specifications distinguish the notion of a *stream* and a *file*. A *file* is a data object stored on a disk, whereas a *stream*, according to [10], "is a source or destination of data that may be associated with a disk or other peripheral." Hence, the ANSI C operation `fopen` opens a *stream* to a specified *file*.

The current implementation of *ft*IO does not support multiple open streams to a single file. Adding this support, however, is not difficult. The *ft*IO finite automaton operates on files, not on streams. Hence, the open/closed state marks whether there are *any* open streams associated with a file independent of the number of streams. Adding support for multiple streams would require reorganizing the *ft*IO bookkeeping to separate the information about the state of a stream from the information about the state of a file.

Supporting UNIX File I/O

The algorithm developed for *ft*IO is not limited to support only standard ANSI C file operations. With small changes, I believe, *ft*IO can support any set of existing file operations. Implementing UNIX file I/O, for example, requires only changing the way that bookkeeping information is stored since UNIX file operations are very similar to the ANSI C ones.

Appendix A

More Experimental Results

Appendix A presents experimental performance data of our benchmarks on a variety of machines. The data show that the overhead of the *ft*IO system, measured as the percentage of the runtime of the program, stays almost constant for machines of very different computational power.

The figures are organized in the order of increasing computational power. Note, however, that some of the less powerful machines, like Sparc4, have apparently superior disk I/O characteristics.

Sequential Access: read and write

The figures below present the performance of reading (left) and appending (right) 10 Mbytes of data from and to a file. Ten checkpoints are taken after about 10^6 file operations (less then every 2 seconds for all machines) in the experiments marked "with checkpointing" and those marked "with append optimization."

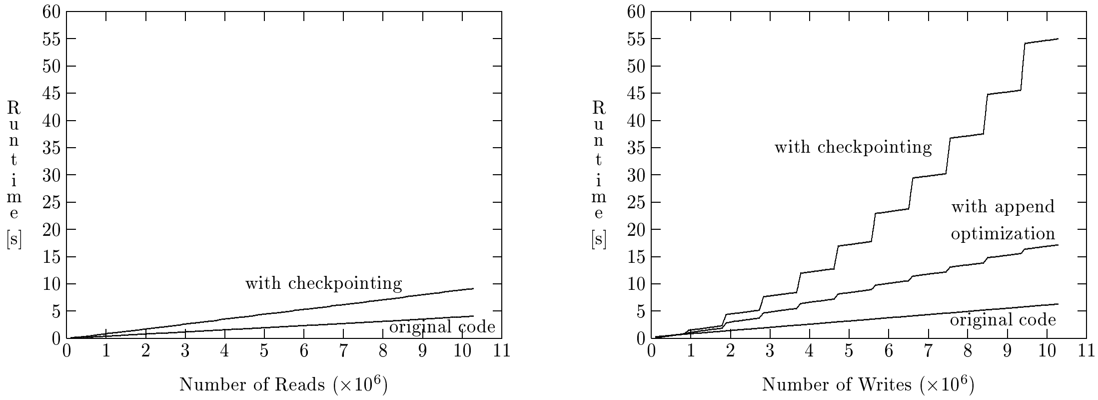

Figure A-1: Sun Sparcstation4. Sequential read/write

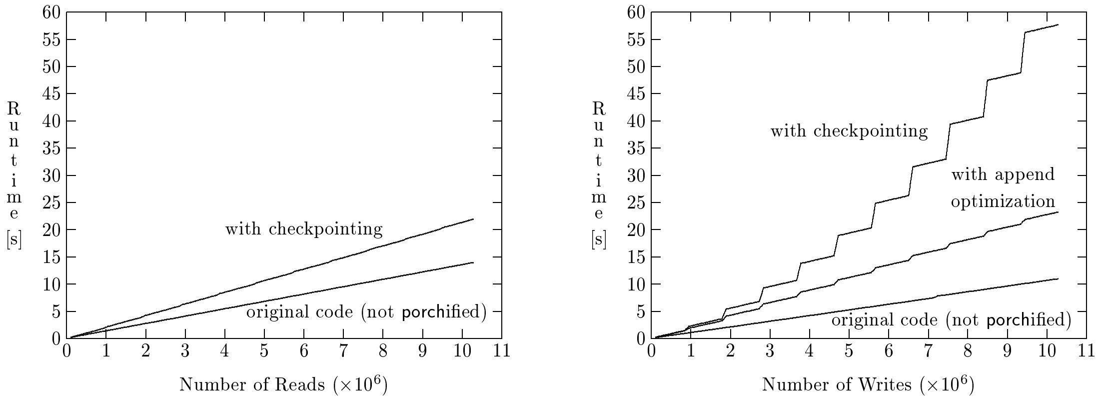

Figure A-2: Sun Sparcstation10. Sequential read/write

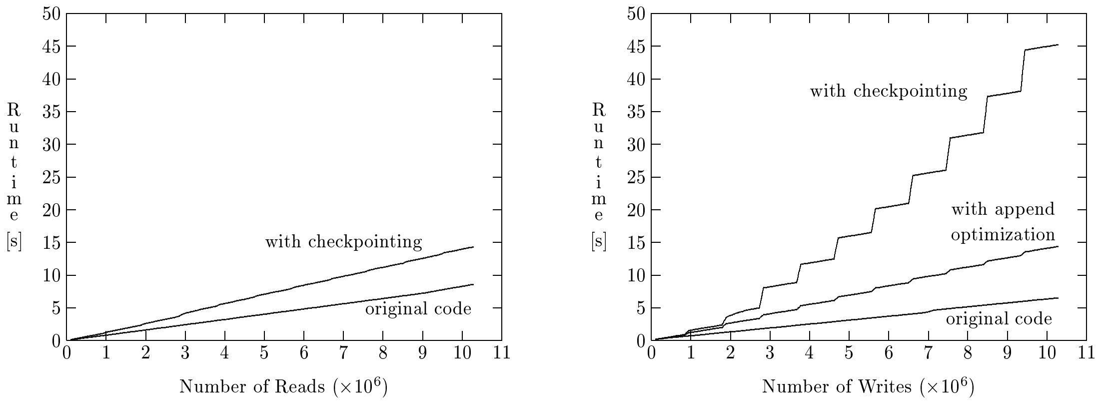

Figure A-3: Sun Sparcstation20. Sequential read/write

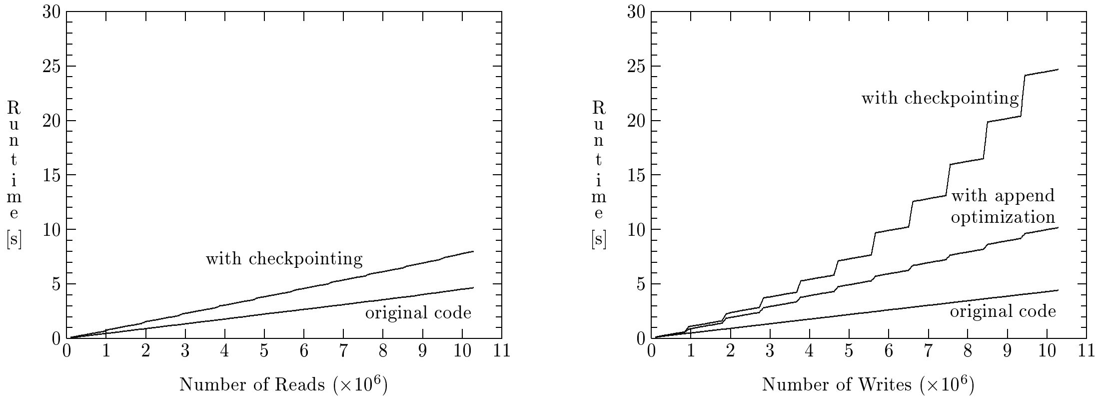

Figure A-4: Sun UltraSparc1. Sequential read/write

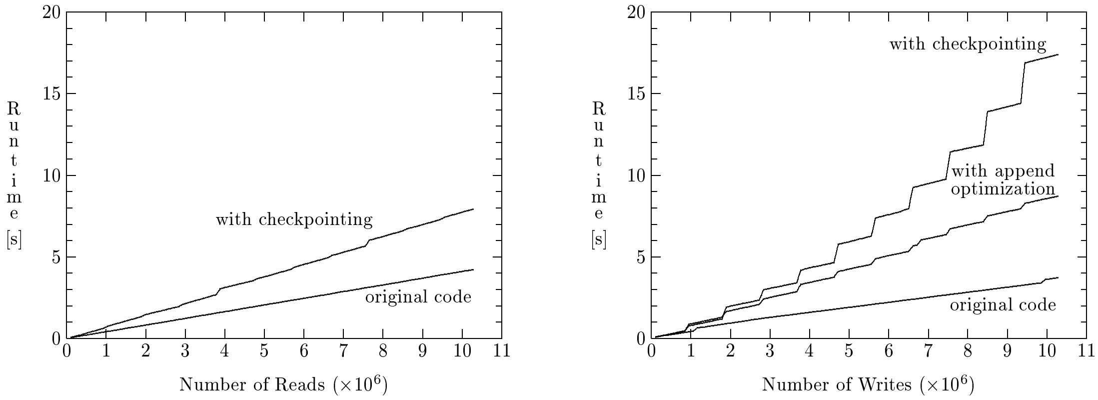

Figure A-5: Sun UltraSparc2. Sequential read/write

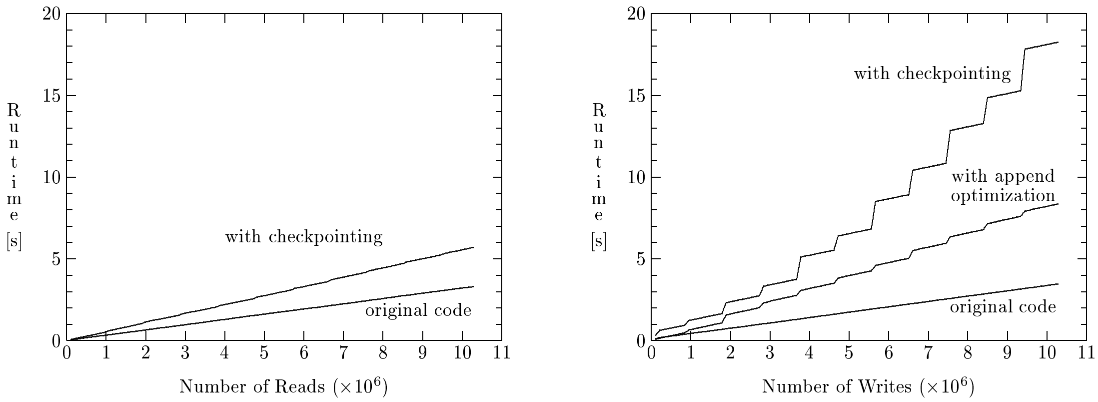

Figure A-6: Sun Ultra-Enterprise. Sequential read/write

Random Access

The figures below present the runtimes of random file accesses with spatial locality. The variation of the number of blocks simulates the behavior of a shadow-block implementation with different block sizes. This benchmark analyzes the performance of *ft*IO with file operations randomly distributed over the address range of a file, and where file accesses exhibit spatial locality. Spatial locality is incorporated by computing the location of a file access to be normally distributed around the location of the previous file access with a standard deviation of 10.24 KBytes. This value corresponds to 0.1 % of the total file size of 10 MBytes. Our benchmark performs 10^6 file operations, 90 % of which are read operations, and 10 % are write operations. Ten checkpoints are taken at equal intervals (less then every 1.5 minutes for all cases) during the execution of the benchmark.

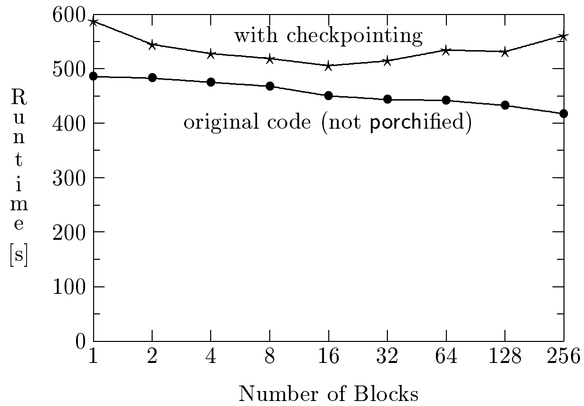

Figure A-7: Sun Sparcstation4. Random access

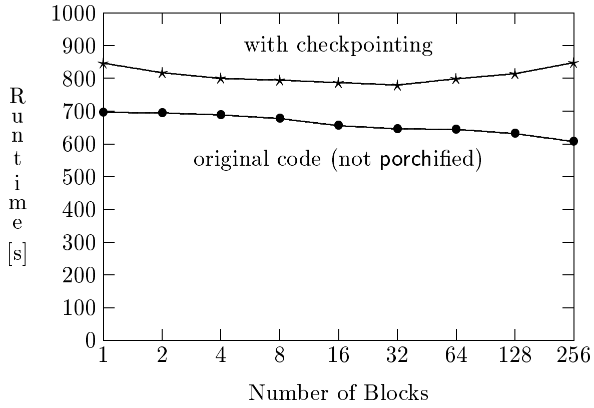

Figure A-8: Sun Sparcstation10. Random access

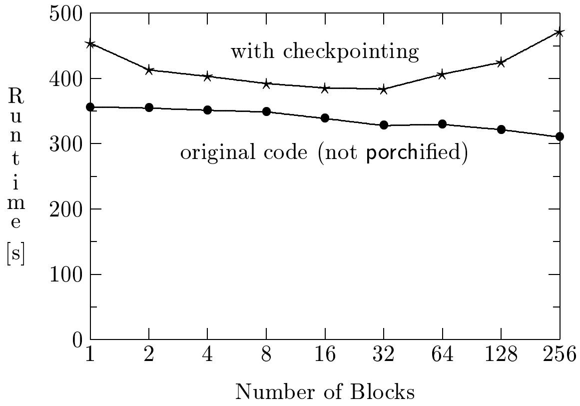

Figure A-9: Sun Sparcstation20. Random access

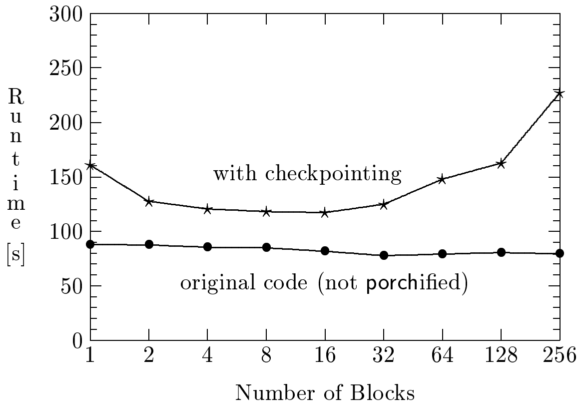

Figure A-10: Sun UltraSparc1. Random access

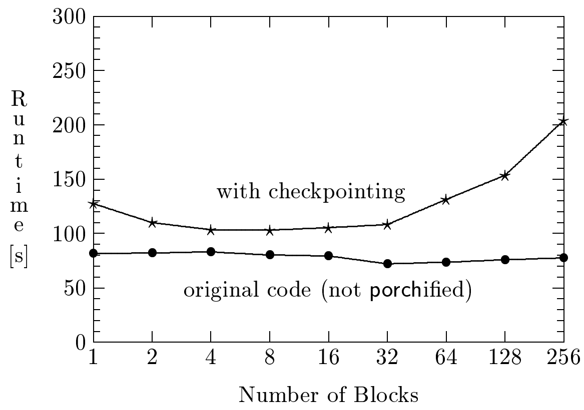

Figure A-11: Sun UltraSparc2. Random access

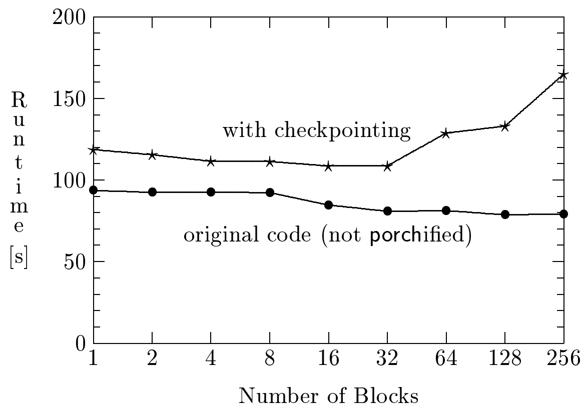

Figure A-12: Sun Ultra-Enterprise. Random access

Molecular Dynamic Code

The figures below present the cumulative runtimes for a 2D short-range molecular dynamics code for $5,000$ particles. The code outputs the simulation results (53 bytes) every 10 iterations to a file, and checkpoints are taken every 20 minutes. The program generates approximately 400 bytes of data between checkpoints, and the total size of the output file after $60,000$ iterations is 150 Kbytes. From the data below, we can see that the performance overhead of the *ft*IO system is relatively low. In fact, it ranges from approximately 8 % on Sun Sparkstation4 to as low as 2 % on Sun UltraSparc1.

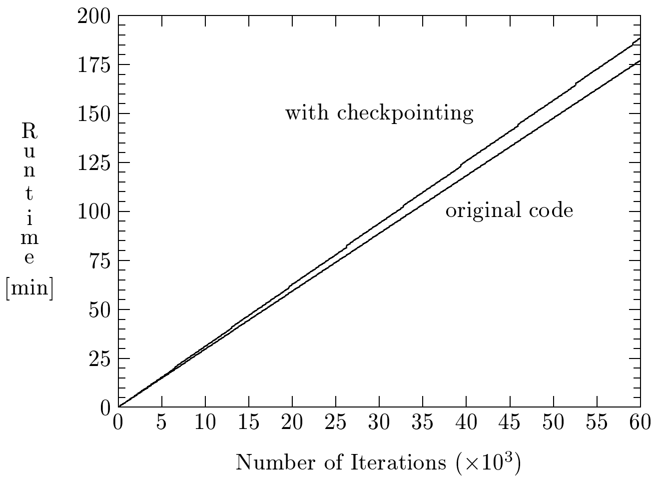

Figure A-13: Sun Sparcstation4. Molecular dynamics

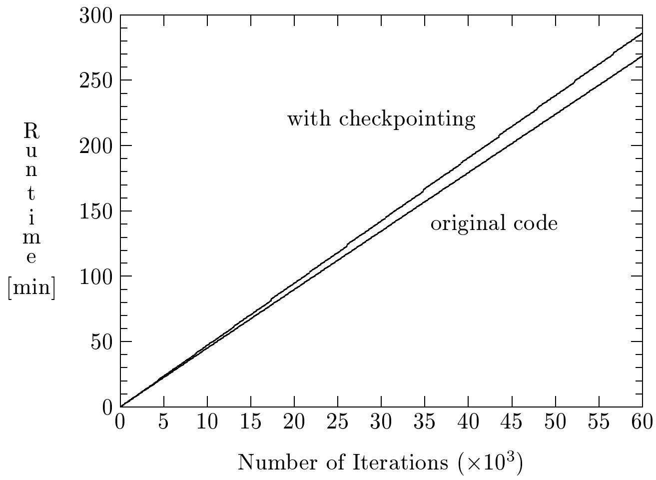

Figure A-14: Sun Sparcstation10. Molecular dynamics

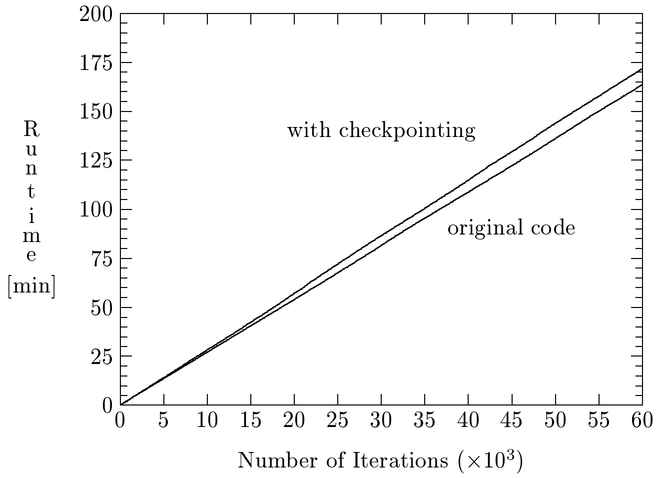

Figure A-15: Sun Sparcstation20. Molecular dynamics

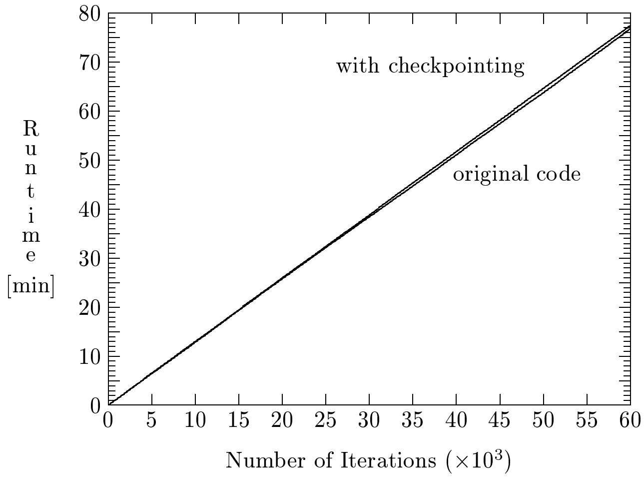

Figure A-16: Sun UltraSparc1. Molecular dynamics

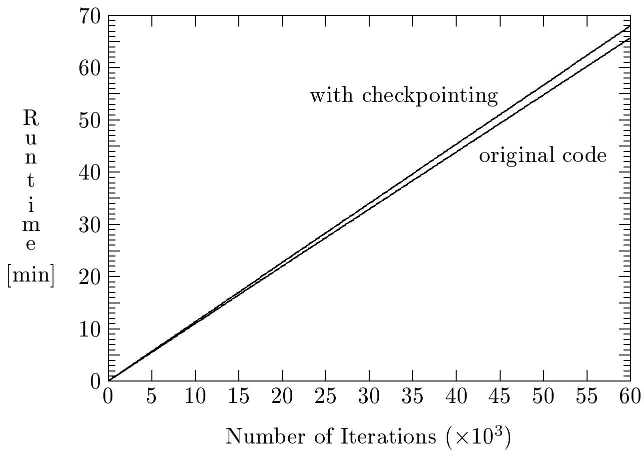

Figure A-17: Sun UltraSparc2. Molecular dynamics

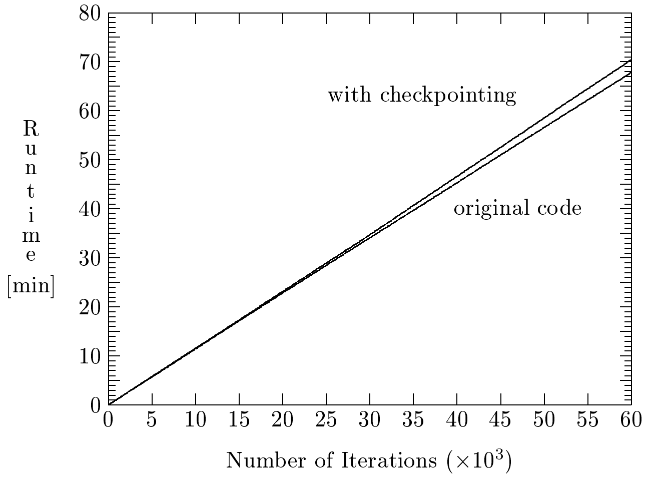

Figure A-18: Sun Ultra-Enterprise. Molecular dynamics

Bibliography

[1] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[2] P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. *ACM Transactions on Computer Systems*, 12, 4:309–339, 1994.

[3] Peter M. Chen and David A. Patterson. Storage Performance—Metrics and Benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, August 1993.

[4] Yuqun Chen, James S. Plank, and Kai Li. CLIP: A checkpointing tool for message-passing parallel programs. Technical Report TR-543-97, Princeton University, Computer Science Department, May 1997.

[5] Stuart I. Feldman and Channing B. Brown. Igor: A System for Program Debugging via Reversible Execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[6] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[7] Christine Hofmeister. *Dynamic Reconfiguration.* PhD thesis, Computer Science Department, University of Maryland, College Park, 1993.

[8] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In Jean-Claude Laprie, editor, *Digest of Papers—23rd Inter-*

national Symposium on Fault-Tolerant Computing, pages 2–9, Toulouse, France, June 1993.

[9] W. H. Jessop, J. D. Noe, D. M. Jacobson, J. L. Baer, and C. Pu. The EDEN transaction-based file system. In *IEEE Symp. on Reliability in Distributed Software and Database Systems, IEEE CS 2, Wiederhold(ed), Pittsburgh PA*, pages 163–169, July 1982.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition.* Prentice-Hall, 1988.

[11] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pages 283–290, Berkeley, CA, USA, Winter 1992. USENIX.

[12] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report CS-TR-97-1346, University of Wisconsin, Madison, April 1997.

[13] A. Nangia and D. Finkel. Transaction-based fault-tolerant computing in distributed systems. In Jha Niraj and Donald S. Fussell, editors, *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 92–97, Amherst, MA, July 1992. IEEE Computer Society Press.

[14] W. H. Paxton. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–23, 1979.

[15] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *USENIX Winter 1995 Technical Conference*, pages 213–233, New Orleans, Louisiana, January 1995.

[16] Peter J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, 1992.

[17] James T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[18] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogeneous Architectures. In *Digest of Papers—27th International Symposium on Fault-Tolerant Computing*, pages 58–67, Seattle, Washington, June 1997. IEEE Computer Society.

[19] Richard Rashid, Avadis Tevanian Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[20] Herbert Schildt. *The Annotated ANSI C Standard*. McGraw-Hill, 1990.

[21] Peter W. Smith. *The Possibilities and Limitations of Heterogeneous Process Migration*. PhD thesis, Department of Computer Sience, University of British Columbia, October 1997. (`http://www.cs.ubc.ca/spider/psmith/tui.html`).

[22] Volker Strumpen. Compiler Technology for Portable Checkpoints. submitted for publication (`http://theory.lcs. mit.edu/~strumpen/porch.ps.gz`), 1998.

[23] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[24] Y. M. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating Checkpointing with Transaction Processing. In *Digest of Papers—27th International Symposium on Fault-Tolerant Computing*, pages 304–308, Seattle, Washington, June 1997. IEEE Computer Society.

[25] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *Digest of Papers—25th International Symposium on*

Fault-Tolerant Computing, pages 22–32, Los Alamitos, June 1995. IEEE Computer Society.

[26] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for a Distributed Shared Memory. In *1st Symposium on Operating Systems Design and Implementation*, pages 87–100, Monterey, CA, November 1994. USENIX.