THEORY AND PRACTICE OF FAILURE TRANSPARENCY

by

David Ellis Lowell

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
1999

Doctoral Committee:

Associate Professor Peter M. Chen, Chair
Associate Professor Farnam Jahanian
Assistant Professor Brian Noble
Associate Professor Paul Resnick

For Marylou, David, and Ashleigh,
and all my family around the world

ACKNOWLEDGMENTS

This dissertation is the culmination of several year’s work, during which time many people have supported me and contributed to this document. I owe great thanks to my friend and advisor, Peter Chen. Many of the ideas in this dissertation emerged during vigorous, scintillating, and seemingly interminable conversations in his office. It is hard to imagine doing science any other way. I sincerely doubt I would have bothered to see this degree to completion were it not for the quality of my interaction with Pete, and I cannot thank him enough.

I would also like to thank my thesis committee. Their advice and comments have been tremendously helpful. A smarter bunch of folks one is not likely to find.

Thanks to my office mates for their company and daily trips out for lunch. Thanks to Wee Teck Ng especially for leading the way on the Rio project.

I have a wonderful bunch of friends who I love. To the gang here in Ann Arbor, in California, and around the rest of the world, thank you for all your support and encouragement. I must especially thank my friend René Bruckner. For the last several years, he and I have perniciously and unrepentantly emailed all throughout the workday, he at his computer in San Francisco, I at mine in Ann Arbor. Our correspondence has saved my sanity more than once.

Finally, I must thank my wonderful family. I cannot imagine two more wonderful parents, or a more wonderful sister. Thank you for your constant support, your enthusiasm for my endeavors, and your interest in my life. Thanks also to my extended family in the USA and abroad. I know you are cheering for me.

PREFACE

This is a study of failures in computer systems, and a guide for those systems that aim to survive their occurrence. Despite remarkable advances in performance and functionality, 40 years of computer science research has not managed to do away with failures. On the contrary, the increasing complexity of computer systems wrought by that research seems not to eliminate failures, but to guarantee them. Moreover, with computers insinuating themselves into the daily experience of more and more people on the planet, the argument could be made that the person•failure metric is spiraling out of control. With this dissertation we argue that if we have to live with computer failures, the least the computer could do is try to keep them a secret.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

PREFACE iv

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER 1

INTRODUCTION 1

1.1 ...... Motivation 1

1.2 ...... Dissertation Overview 3

1.3 ...... Scope of the Thesis 4

1.4 ...... Background: The Rio File Cache 5

CHAPTER 2

THEORY OF CONSISTENT RECOVERY 8

2.1 ...... Introduction 8

2.2 ...... Theory of Consistent Recovery 9

2.2.1 ... Computation model 10

2.2.2 ... General theory of consistent recovery 12

2.2.3 ... Assumptions for recovery 16

2.2.4 ... Special theory of consistent recovery 17

2.2.5 ... Upholding the theorem 31

2.2.6 ... Refining Assumptions 1 and 2 34

2.3 ...... Applying the Theory 36

2.3.1 ... Unifying existing recovery protocols 36

2.3.2 ... Revealing shortcomings in existing recovery research 41

2.3.3 ... Exploring interactions with the fail-stop model 43

2.3.4 ... Protocol space and other design variables 44

2.4 ...... Related Work 44

2.5 ...... Conclusion 45

CHAPTER 3

LIGHTWEIGHT TRANSACTIONS USING RELIABLE MEMORY 47

3.1 ...... Introduction 47
3.2 ...... Related Work 48
3.3 ...... The Rio File Cache 50
3.4 ...... The Vista Recoverable Memory 50
3.5 ...... Performance Evaluation 54
3.5.1 ... Benchmarks 54
3.5.2 ... Environment 56
3.5.3 ... Results 56
3.6 ...... Conclusions 61

CHAPTER 4

DISTRIBUTED RECOVERY USING VISTAGRAMS 62

4.1 ...... Introduction 62
4.2 ...... Background 63
4.3 ...... Messages in Local Transactions Model 63
4.3.1 ... Protocol independence 65
4.4 ...... Design and implementation of Vistagrams 65
4.4.1 ... Proof that Vistagrams guarantees consistent recovery 66
4.4.2 ... Vistagrams implementation 67
4.4.3 ... Sample application 68
4.4.4 ... Comparison to existing systems 69
4.5 ...... Vistagrams performance 70
4.6 ...... Microbenchmark 70
4.6.1 ... Environment 72
4.6.2 ... Results 72
4.7 ...... Using MLT and Vistagrams 72
4.8 ...... Related Work 73
4.9 ...... Conclusion 74

CHAPTER 5

DISCOUNT CHECKING: FAST, USER-LEVEL, FULL-PROCESS CHECKPOINTS 75

5.1 ...... Introduction 75
5.2 ...... Design and Implementation of Discount Checking 76
5.2.1 ... Transactions and Checkpointing 76
5.2.2 ... Saving Process State 76

5.2.3 ... Minimizing Memory Copies ... 79
5.2.4 ... Vista-Specific Issues ... 80
5.3 ...... Using Discount Checking ... 81
5.4 ...... Performance Evaluation ... 82
5.4.1 ... Microbenchmark ... 82
5.4.2 ... Varying the Checkpoint Interval ... 83
5.5 ...... Related Work ... 87
5.6 ...... Conclusions ... 88

CHAPTER 6

FAILURE TRANSPARENCY FOR GENERAL APPLICATIONS USING FAST CHECKPOINTS ... 89

6.1 ...... Introduction ... 89
6.2 ...... Providing Failure Transparency ... 90
6.2.1 ... Discount Checking ... 90
6.2.2 ... Recovery Protocols ... 91
6.2.3 ... Applications ... 95
6.2.4 ... Results ... 97
6.3 ...... Related Work ... 104
6.4 ...... Conclusion ... 105

CHAPTER 7

CONCLUSION AND FUTURE WORK ... 106

7.1 ...... Conclusion ... 106
7.2 ...... Future Work ... 109

BIBLIOGRAPHY ... 111

LIST OF FIGURES

Figure 1.1: Using the Rio File Cache 6

Figure 2.1: Recovering an alphabet program 13

Figure 2.2: Comparison of two recoveries 15

Figure 2.3: Non-deterministic events 19

Figure 2.4: Single process inconsistency from non-determinism 20

Figure 2.5: Multi-process inconsistency from non-determinism 21

Figure 2.6: Universe partitioned into Fail and NoFail regions. 24

Figure 2.7: Fail partitioned into Redo and Deviate regions. 24

Figure 2.8: Deviate partitioned into Hidden and Exposed regions. 25

Figure 2.9: A problematic computation for liveness 28

Figure 2.10:Example of state machine in which Assumption 1 does not hold 35

Figure 2.11:Protocol space 40

Figure 2.12:Domino effect 42

Figure 3.1: Operations in a Typical Recoverable Memory System 49

Figure 3.2: Operations in Vista 51

Figure 3.3: Memory Allocation in Vista 53

Figure 3.4: Transaction overhead 57

Figure 3.5: Small transaction throughput 58

Figure 3.6: Large transaction throughput 59

Figure 4.1: A distributed computation 63

Figure 4.2: Distributed computation using MLT 64

Figure 4.3: Some of the routines in the Vista and Vistagrams API 66

Figure 4.4: A sample recoverable lock manager using Vistagrams 69

Figure 4.5: Vistagrams in the protocol space 71

Figure 4.6: Vistagrams performance 73

Figure 5.1: Equivalence of Checkpointing and Transactions 77

Figure 5.2: Overhead vs. checkpoint size. 84

Figure 5.3: Checkpoint Overhead for Varying Intervals: ijpeg 85
Figure 5.4: Checkpoint Overhead for Varying Intervals: m88ksim 86
Figure 6.1: Recovery protocols in the protocol space 94
Figure 6.2: Relation between all recovery protocols we study 96
Figure 6.3: Performance of recovery protocols for nvi 98
Figure 6.4: Performance of recovery protocols for oleo 99
Figure 6.5: Performance of recovery protocols for magic 100
Figure 6.6: Performance of recovery protocols for TreadMarks 101
Figure 6.7: Performance of recovery protocols for xpilot 102

LIST OF TABLES

Table 3.1: Disk Parameters 56
Table 5.1: Experimental Platform 83
Table 6.1: Experimental Platform 97

CHAPTER 1

INTRODUCTION

In building systems that aim to survive failures, systems designers must tackle many thorny issues. For example, there is the issue of which party handles the failure, the application or the operating system? What class of failures can the system survive? Does failure-free performance suffer? Can the user tell the failure occurred even if it gets handled correctly? What exactly must a system do to handle a failure anyway? By methodically building a theory and systems that aim to let applications survive failures, we endeavor with this study to illuminate many of these issues.

1.1 Motivation

People don't complain about their slow computers much any more. The success of hardware designers' mobilization for megahertz has been that all but the most performance obsessed have enough cycles. During a visit to an office or computer lab these days, one is far more likely to hear complaints resulting from system crashes, buggy software, configuration nightmares, lost work, and system reinstalls.

These complaints all stem from the unreliability of current computer systems. Indeed, the most common operating system in use today on desktop computers is reported in technical literature to be downright "unstable", crashing frequently [Costa97]. Furthermore, failures are not going away. They seem to be a necessary side effect of the complexity of modern computers and software.

Recognizing that failures are a fact of life, many researchers have examined the problem of enabling systems to tolerate their occurrence. Fault-tolerance research has focused historically on providing reliability for computer systems that must survive hardware faults. However, researchers have found that over 90% of failures are caused by faulty software [Gray90]. Furthermore, traditional fault tolerance has been targeted for high-end business machines, factory control, scientific computation, and mission critical systems. Fault tolerance for mainstream, commodity computer systems has been largely overlooked.

We conclude that there is a glaring need for systems that provide reliability in the face of software failures, and do so for real, interactive applications running on general-purpose hardware—precisely the applications and systems employed by the vast majority of computer users. It is into this void we step with this dissertation.

Our goal with this thesis is to investigate the issues behind masking failures from users. The challenges in doing so are many. First of all, recovering failed applications typically involves saving recovery information on stable storage, sometimes very frequently. These writes must be done quickly. Masking failures also necessarily involves recovering applications that have no internal support for recovery. Finally, systems that endeavor to recover applications consisting of many coordinating processes must ensure that the processes all continue to execute in a mutually consistent fashion after recovery. And all of these requirements must be met in a manner that is invisible to the user.

During the years of fault-tolerance research mentioned above, researchers have studied the problem of recovering systems after they fail. Their labors have born fruit in the form of myriad consistent recovery protocols such as message logging, distributed checkpointing, and two-phase commit. Surprisingly however, there are no fundamental guidelines for guaranteeing consistent recovery that are relevant independent of protocol.

In the first stage of our work, we will construct a theory of recovery that unifies all existing recovery protocols and defines the fundamental rules for providing consistent recovery. The theory will then serve as the backbone for the remainder of our investigation: using the theory, we will systematically explore many of the remaining challenges of masking failures from users.

In the end, we will argue that operating systems can and should provide the fundamental abstraction that systems do not fail, an abstraction we call *failure transparency*. We find that systems can provide failure transparency for real, interactive applications on general-purpose hardware with overhead of less than 2%.

Although failure transparency and consistent recovery are closely related, they are not the same thing. Consistent recovery defines the conditions for the correct handling of application state after a failure. This notion of consistency leaves open the possibility that users will discern failures, and that programmers will be the parties responsible for adding recovery abilities to their applications. In contrast, systems that provide failure transparency endeavor to mask all signs of failure from the user, and take responsibility for recovering applications consistently. In other words, providing failure transparency involves achieving consistent recovery on behalf of applications, and constrains that it be done transparently to the user and programmer.

Most of the systems we build during the course of our investigation use reliable memory provided by the Rio File Cache as stable storage. As a result, a secondary theme of this work is examining the use of reliable memory for recovery. We find that reliable memory is profoundly useful in this arena. It simplifies and accelerates the fundamental tools of recovery, and allows recovery in ways that are not possible with traditional forms of stable storage such as disk.

1.2 Dissertation Overview

In this dissertation, we gradually assemble the pieces needed to build systems that provide failure transparency.

In Chapter 2, we construct a model of computation and use that model to formally define the general problem of failures by computer systems. We then develop our theory of consistent recovery which provides the ground rules for recovering transparently from failures. We then show how existing recovery protocols all fit into the theory we have developed, and discuss several new protocols that fall out of it.

One conclusion from Chapter 2 is that faster commits allow for simpler recovery protocols. In Chapter 3, we develop Vista, a lightweight transaction system that provides an extremely fast commit. Vista is built upon Rio's reliable memory. We compare Vista's performance and complexity against a lightweight transaction system called RVM that is designed to use disk as stable storage. We find that running RVM on Rio provides a significant but expected 20 times speedup. More surprisingly, we find that custom designing Vista for Rio provides and additional factor of 100 speed up, for a factor of 2000 overall.

In Chapter 4, we take a first stab at providing consistent recovery for distributed applications. We build a system called Vistagrams in which messages are persistent, and sent and received within local Vista transactions. We show that distributed applications can use Vistagrams to get consistent recovery with almost no overhead. However, programmers are the parties responsible for guaranteeing consistent recovery using Vistagrams, rather than the system. As a result, Vistagrams cannot be said to provide failure transparency as we have defined it.

In Chapter 5, we build a system called Discount Checking that provides another form of fast commit, this time a lightweight full process checkpoint. Discount Checking uses Rio's reliable memory and Vista's fast transactions. It preserves most kernel state during every user-level checkpoint by duplicating that kernel state at user level and storing it in persistent memory provided by Vista. We show that Discount Checking can perform a full process checkpoint in as little as 50 µs.

Discount Checking's checkpoints are useful for providing true failure transparency. In Chapter 6, we investigate doing so for a representative set of real, interactive applications using a variety of recovery protocols revealed by our theory of consistent recovery. We find that we can provide failure transparency for real applications with overhead of less than 2%. Furthermore, we find that no one protocol performs the best for all applications.

Chapter 7 concludes.

1.3 Scope of the Thesis

This dissertation details our investigation of the following thesis: there exists a theory of consistent recovery that unifies all existing recovery protocols and lights the path to providing failure transparency. Furthermore, it is feasible to provide failure transparency as a fundamental abstraction of the operating system with low overhead.

The path we take in investigating this thesis cuts a broad swath through traditional systems and fault tolerance research. We begin by constructing the theory itself and showing the relationship between our theory and existing recovery research. We then construct and evaluate a series of systems that each bring us closer to the goal of providing failure transparency. Along this path, we make a series of contributions, including:

- creating a theory of consistent recovery that unifies the many separate consistent recovery protocols in existence and shows them to all be variations on the theme of a single invariant.
- building a system that provides very lightweight transactions and exposes the price systems builders pay in complexity and overhead for disk's slow performance.
- showing that it is possible to perform full-process checkpoints at user level for a large class of complex, real applications.
- developing a number of new recovery protocols.
- revealing that it is possible to provide failure transparency for real, interactive applications with overhead of less than 2%.
- highlighting the usefulness of Rio's reliable memory in recoverable systems.

As a second order effect, we contribute by staying focused throughout this work on providing fault tolerance for real, interactive applications running on low-end systems, a vast but traditionally overlooked target domain.

1.4 Background: The Rio File Cache

All systems that provide recovery use stable storage copiously. As mentioned in Section 1.1, most of the systems we build use as stable storage reliable memory provided by the Rio File Cache. However, all our systems will work with any stable storage technology that lets applications map persistent memory into an application's address space.

Like most file caches, Rio buffers file data in main memory to accelerate future accesses. Unlike traditional caches however, Rio seeks to protect this area of memory from its two common modes of failure: power loss and system crashes [Chen96, Chen99, Ng99]. Protecting against power outages is as simple as using a $100 uninterruptible power supply [APC96]. Software errors are tricker however. Rio uses virtual memory protection to prevent operating system errors, such as wild stores, from corrupting the file cache during a system crash. This protection does not significantly affect performance. During a crash, Rio writes the file cache data in memory to disk, a process called *safe sync*.

To verify Rio's reliability against software crashes, Chen et al. crashed the operating system several thousand times using a wide variety of randomly chosen programming bugs. Their results showed that Rio is even more reliable than a file system that writes data immediately through to disk. To further establish Rio's ability to make memory as safe as disk, the Rio team stores their home directories, source trees, and mail on Rio. In three years and many system crashes, the team has not lost any data.

There are two main ways systems can use the reliable memory Rio provides: `write` and `mmap` (Figure 1.1). Applications commonly modify data in the file cache using the `write` system call. This explicit I/O operation copies data from a user buffer to the file cache. In Rio, each `write` also changes twice the protection on the modified page. To guarantee immediate permanence, applications running on non-Rio systems must use `fsync` or other synchronous I/O to force new data to disk, or they must bypass the file cache and write directly to the raw disk. On Rio, the data being written is permanent as soon as it is copied into the file cache.

In contrast to `write`, `mmap` maps a portion of the file cache directly into the application's address space. Once this mapping is established, applications can use store instructions to manipulate the file cache data directly—no copying is needed. Applications using `mmap` on non-Rio systems must use `msync` to ensure that newly written data is stored on disk. On Rio, however, data from each individual store instruction is persistent immediately without invoking `msync`.

Each method of modifying the file cache has advantages and disadvantages. Mapping a portion of the file cache has significantly lower overheads associated with it, especially on Rio, as

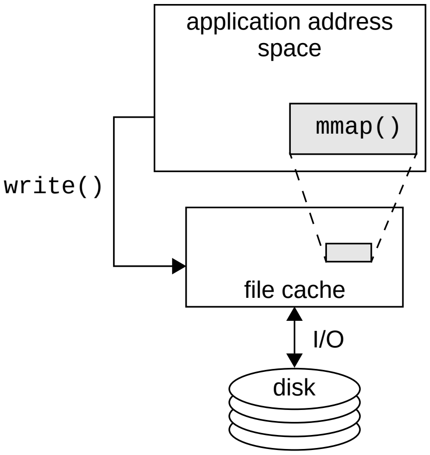


Figure 1.1: Using the Rio File Cache

Applications can modify persistent data in the Rio file cache using `write` and `mmap`. Most applications use `write`, which copies data from an application buffer to the file cache. `mmap` maps a portion of the file cache directly into the application's address space. Once this mapping is established, applications can use store instructions to manipulate file cache data directly; no copying is needed.

stores to a mapped region incur no system calls. Modifying file cache data with `write` however requires one system call. In addition, `write` requires an extra memory-to-memory copy, which can lead to keeping each datum in memory twice.

The main advantage of `write` is that it may isolate the file cache from application errors. The protection mechanism in Rio focuses only on kernel errors. Users are still responsible for their own errors and can corrupt their file data through erroneous `write` calls or store instructions. Because store instructions are much more common than calls to `write`, applications may damage the file cache more easily using `mmap` [GM92]. While other systems, such as Mach's Unix server [Golub90], have mapped files into the application address space, we are aware of only three papers that quantify the increased risk [Chen96, Ng97, Ng99]. These papers indicate that (1) the risk of corruption increases only marginally when mapping persistent data into the address space and (2) memory protection can be used to further reduce this risk to below that of using a `write` interface. The first two studies were done on the Rio file cache and are discussed earlier in this section. The third study compares the reliability of `mmap` and `write` in the Postgres database and finds little difference. Users can enhance reliability by applying virtual memory techniques to their address space [Sullivan91, Chen96]. They can also use transactions to reduce the risk of corrupting permanent data structures in the event of a crash.

The reliability of recoverable systems depend on the reliability of the underlying stable storage they employ. Rio is geared to handle power outages as well as software crashes by either the application or operating system. Since software errors account for the vast majority of faults [Gray90], the stable storage provided by Rio's reliable memory is sufficient for most applications. Furthermore, Rio can be modified to handle hardware failures by having it replicate cached data on additional hosts.

CHAPTER 2

THEORY OF CONSISTENT RECOVERY

We begin by carefully formulating the general problem of failures by computer systems, and the challenges associated with surviving their occurrence. We develop a logical framework for our discussion, establish the fundamental definitions we will employ throughout this dissertation, and make explicit the assumptions underpinning our work. We then develop the theory of recovery upon which the systems in later chapters will be based, and begin to show the theory's use in interpreting and clarifying existing recovery research.

2.1 Introduction

The primary goal of operating systems and middleware is to provide abstractions to the user and programmer that hide the shortcomings of the underlying system. For example, threads create the abstraction of more CPUs, and virtual memory creates the abstraction of more memory. While today's operating systems provide many powerful abstractions, they do not hide one of the most critical shortcomings of today's systems, namely, that operating systems and user applications fail. Rather, operating systems have been content to provide a low degree of fault tolerance. For example, popular operating systems are concerned only with saving unstructured file data (and even in this limited domain they accept the loss of the last few seconds of new file data). In particular, non-file state in the system, such as the state of running processes, is lost completely during a crash and this loss exposes failures to users and application writers.

Losing process state inconveniences both application writers and users. Application writers must bear the burden of hiding failures from users by using ad hoc techniques. These ad hoc techniques require considerable work on the part of the application writer, because recovery code is extremely tricky to get right. Losing process state also inconveniences the user, because most applications lose significant state during a crash such as recent changes to a user's file and the state of the user's interaction with the application (e.g. editing mode, cut-and-paste buffers, cursor position, etc.). As a result, recovering from a failure involves significant user intervention and inconvenience—consider people's vehement reaction when their operating system crashes.

We believe operating systems should present the abstraction to users and application writers that operating systems and applications do not fail. We call this abstraction *failure transparency*. Ideally, failure transparency would provide a perfect illusion that operating systems and applications do not fail, they merely pause and resume. As with all abstractions, it may not be possible to provide a perfect illusion of failure transparency. We explore some of the limits to failure transparency in this chapter.

We define *consistent recovery* as recovering from crashes in a way that makes failures transparent. The notion of recovery is hardly new. Many techniques have been proposed that enable applications to recover from failures [Koo87, Johnson87, Strom85, Elnozahy92]. A few isolated techniques have even been implemented in operating systems that provide some flavor of failure transparency [Bartlett81, Powell83, Borg89, Baker92, Bressoud95]. Despite the maturity of the field however, recovery researchers have not proposed a single rule for attaining consistent recovery that is independent of all recovery protocols, and that relates the various classes of application events with events needed to support recovery. As a result, the space of recovery protocols has not been explored systematically, and it is difficult to discern the relationship between existing protocols.

Our goal in this chapter is to provide a definition and theory of consistent recovery. What is the theory good for? Handling failures is tricky business, particularly when many processes are interacting. The theory provides the fundamental invariant that every distributed or localized application must uphold in order to guarantee consistent recovery. The theory also provides a unified way of viewing all existing recovery protocols, elucidating the relationship between disparate protocols. We will explore all these applications of the theory in this paper. In addition, we will use the theory to expose new recovery protocols and point out a few shortcomings in existing recovery research. In Chapter 6, we will show the theory in action by examining the performance trade-offs of seven recovery protocols that arise naturally from the theory. Along the way, we will show the feasibility of providing failure transparency for a difficult class of applications.

2.2 Theory of Consistent Recovery

Our first task is to formally define a general model of computation. We will then use that model in describing the problem of failures, and the challenges of recovering from them.

2.2.1 Computation model

A *computation* is defined as a set of one or more *sequential processes* working together on a task. For example, we would represent a computation *C* consisting of *n* processes as the following set:

$$C = \{p_1, p_2, \ldots, p_n\}$$

A sequential process is a basic participant in a computation. We model sequential processes as finite state machines that change state in response to inputs.

The local state σ of a process consists of its memory (variables, registers, etc.).

σ_k^i = local state of process *k* after processing the i^{th} input
σ_k^0 = initial state of process *k*

The state to which a process atomically transitions as a result of processing an input is governed by the *next-state function* Φ. If *S* is the set of all possible states, and *I* represents the set of all possible inputs, the next-state function is defined as:

$$\Phi : S \times I \rightarrow S$$
$$\Phi(\sigma, input) = \sigma'$$

where σ' is the local state of the process after processing *input* in state σ. If the process ever finishes executing, it does so by reaching a *terminal state*.

We call each state transition the process executes in response to an input an *event*. Thus an event *e* can be characterized by a 2-tuple:

$$e = (\sigma, \sigma')$$

We may also refer to transitions in a process's state diagram as events.

A process is said to have an event history *h*. The event history is the sequence of events executed by the process. If a process has executed *n* events, its event history would be:

$$h = e^1, e^2, \ldots, e^n$$

We may occasionally refer to history *h* as a set when the ordering of events in the history is unimportant to the discussion.

Thus, a sequential process consists formally of local state σ and a next-state function Φ. The execution of the process occurs as it processes a sequence of inputs, resulting in a series of transformations to the process's state.

The global state Σ of a computation is the set containing the local states of all processes in the computation:

$$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$$

Processes in a computation can communicate by sending and receiving messages. A process sends a message by executing a *send* event. The receiver of that message executes a corresponding *receive* event. In real systems, processes can also communicate by writing to a memory space that is shared between processes, such as a file system, a shared memory, or a file mapped by more than one process. We model writes and reads of these shared memories as sends and receives respectively.

Processes can also execute *visible events*. These events have side effects that can be seen by an observer of the computation and are considered part of the computation's result. Such actions as printing to the screen, sending a print job, or launching a missile might constitute visible events. Visible events in our model correspond to events that have been called "output events" or "output messages" in recovery literature [Elnozahy92].

Events can relate to one another through "cause and effect". We say event e^1 *causally precedes* event e^2 if:

> e^1 and e^2 are executed in that order by a single process, e^1 is a send event and e^2 is the corresponding receive by another process, or e^1 causally precedes *e* and *e* causally precedes e^2.

This notion of causality corresponds to Lamport's "happened before" relation [Lamport78]. We may also say e^2 *causally depends on* or *causally follows* e^1 if e^1 causally precedes e^2.

Under our model, computation proceeds *asynchronously.* That is, there exist no bounds on either the relative speeds of processes or message delivery time. For an asynchronous system, the only way to order events is through causal precedence [Lamport78].

With our model for computations established, we next turn our attention to formally defining what it means for a computation to recover from process failures.

2.2.2 General theory of consistent recovery

Sequential processes can fail, losing their state as a result. We would like computations to be able to handle process failures in order to mask their occurrence from the user.

Recovery is the act of reconstructing a state of the computation after one or more of the processes in the computation fail. However, only *consistent recovery* will enable the computation to continue a legal execution after a failure.

Given a computation C that has failed and recovered, outputting a sequence V of visible events before and after its failure, we define consistent recovery as follows:

Definition 1: Consistent Recovery

> C's recovery is consistent if and only if there exists a failure-free execution of the computation that would result in a sequence of visible events V', and V and V' are equivalent.

In other words, recovery is consistent if despite a computation's failure and recovery an external observer (i.e. user) sees a sequence of visible events that is equivalent to one produced by a failure-free execution of the computation [Strom85].

This definition of consistent recovery establishes two constraints on how computations recover: a constraint of *safety* and a constraint of *liveness*. The safety constraint requires that the computation not execute any visible events as a result of a failure that are incompatible with those executed before the failure. The liveness constraint follows from the fact that the definition compares the visible events of complete executions of C with those of complete failure-free executions. If a computation recovers but is unable to output a complete sequence of visible events, its recovery cannot possibly be consistent. In other words, a single failure should not be able to prevent the computation from executing to completion. Of course, continuous failures can always prevent a computation from completing.

There are some interesting implications of this user-centric view of consistent recovery. First of all, a computation that doesn't produce any output seen by the external observer can never be inconsistent. Second, in this definition of consistent recovery, messages are not the currency of consistency—only events visible to the observer can affect consistency. It is true that messages are related to the internal correctness of the computation. But incorrectness resulting from message handling during recovery is only relevant once it affects the visible output of the application.

This view differs somewhat from classical recovery research, which has operated from the premise that, to ensure correct execution after failures, computations must recover their failed processes to a "consistent cut" [Chandy85, Koo87]. A consistent cut is a global state of the computa-

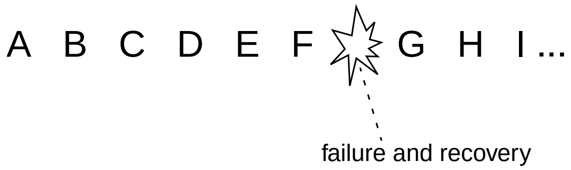


Figure 2.1: Recovering an alphabet program

In this example we see the output of a program that writes the alphabet. Part way through its execution it fails and recovers. Its recovery is consistent by Definition 1, assuming an *identity* equivalence function.

tion where, for each message *m* whose receipt is reflected in a process's local state, the send of *m* is reflected in the sender's local state. In our user-centric view, recovery may be consistent even if the computation does not recover a consistent cut. For example, the computation may not execute any visible events, and thus any recovered state will suffice.

The equivalence of visible event sequences is given by an application-specific *equivalence function*. If *V* is the sequence of visible events executed by the recovered computation and V' is a sequence of visible events output by a legal execution of the computation, then

$$E(V, V') = \{true, false\}$$

where *E* is the equivalence function. $E(V, V')$ is *true* if the two sequences are equivalent, and *false* otherwise.

Equivalence functions are application-specific encapsulations of constraints on recovery. For example, an application may require that the visible events output by the computation be exactly those executed by a failure-free execution of the computation. This requirement can be expressed as the *identity* equivalence function. Consider an application that outputs the alphabet and defines and uses the *identity* equivalence function. Figure 2.1 depicts the output from this application both before and after a failure and subsequent recovery. Clearly there exists a failure-free execution of the process that would result in the output seen, namely its normal execution in which it outputs the complete alphabet. Since the output is equivalent to this failure-free execution, the depicted recovery is consistent by Definition 1.

What would a computation have to do to guarantee consistent recovery using the *identity* equivalence function? Processes can execute *commit events* to aid later recovery. Given that event $e_k^i = (\sigma_k^{i-1}, \sigma_k^i)$ is the *i*'th event executed by process *k*, if e_k^i is a commit event, σ_k^i is the committed state *i* of process *k*. By committing with event e_k^i, process *k* guarantees that should it crash at some point after executing e_k^i, it can restore state σ_k^i and continue executing from there. How the commit is carried out is not important to our discussion, although typically committing a process's state will involve taking a checkpoint[1] or writing a commit record to stable storage. So that process's initial states are preserved under our model, the first event executed by every process is a commit event.

In order to guarantee recovery that is consistent using the *identity* equivalence function, each process in the computation would have to commit its state atomically with each visible event it executes. To see why atomicity is required, consider a scenario in which the process commits immediately before the visible event. After executing the visible event, the process could crash and recover back to its state before the visible, then duplicate the visible event during recovery, which is not allowed by the *identity* equivalence function. Committing after the visible event also does not work, as the process could execute the visible event and fail just before committing. Then the process could recover to an earlier checkpoint and go off to execute a visible event that is incompatible with the visible event executed pre-crash. Committing both before and after the visible event also won't work, as the visible event could still be duplicated during recovery if a crash happens immediately before the second commit. Thus, the visible event and the commit must be executed atomically to satisfy the *identity* equivalence function during recovery.

Since the effects of visible events on the outside world are generally not undoable, visible events and commits can be atomic only if the visible events are *testable* [Gray93]. However, current computer systems cannot test the result of visible events, and so the *identity* equivalence function cannot be implemented.

Consider a weaker equivalence function that is amenable to tractable implementations. An application may define as acceptable the duplication of the last visible event executed. Figure 2.2 depicts two recoveries of our alphabet program and analyzes them according to this equivalence function. An application that used this equivalence function would be able to guarantee consistent recovery by committing immediately *before* every visible event—a protocol that is easier to implement than the atomic commit protocol needed to recover consistently using the *identity* equivalence function.

A wide variety of equivalence functions are possible. Each varies in how tightly it constrains recovery, and how useful are the recovered computations it allows. The above *okay to duplicate last visible* equivalence function is still quite restrictive: it forces a checkpoint before

[1] Note that we use the term "checkpoint" to refer to saving a process's state. This use differs from the database term "checkpoint", which refers to the truncation of a redo log [Lomet98].

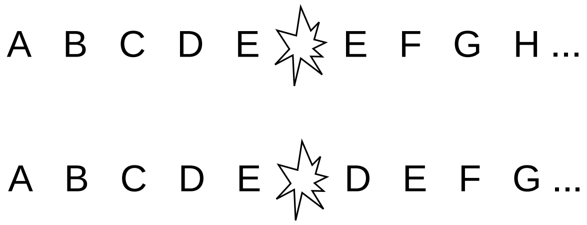


Figure 2.2: Comparison of two recoveries

The first recovery is consistent assuming an *okay to duplicate last visible* equivalence function. The second recovery is not consistent by this equivalence function.

every visible event. An equivalence function that tolerates dropped visible events is less restrictive, but almost useless in practice. Such a function would deem consistent processes that recovered by spinning forever.

All discussion of consistent recovery occurs either implicitly of explicitly in the context of a specific equivalence function. Although equivalence functions are, in general, application specific, we would like to fix an equivalence function that provides a standard of recovery with which most applications would be happy. With an equivalence function established, we will be free to describe in detail how general applications can achieve consistent recovery.

For the remainder of our discussion, we will assume the following equivalence function. Given a sequence of visible events *V* output by a computation *C* that has failed and recovered, and a sequence V' of visible events output by a failure-free execution of *C:*

Definition 2: Equivalence Function

if $\forall v^i \in V$ and $\forall v^j \in V'$ $v^i = v^j$, ignoring v^i if $v^i \neq v^j$ and $\exists v^k \in V$ and $v^i = v^k$ and $k < i$, then *V* and V' are equivalent, otherwise they are not.

This equivalence function requires that *V* and V' be identical except for the duplication of earlier events from *V.* In other words, it allows computations to duplicate visible events as a result of a failure. This function allows a great deal of flexibility in how we guarantee consistent recovery and is closely related to the one implicitly assumed by most existing recovery protocols. Most importantly, it is a reasonable one to use in practice: typical users can overlook the duplication of visible events that result from recovery.

2.2.3 Assumptions for recovery

Providing consistent recovery for general applications implies having the ability to recover these applications without application-specific recovery code. We next describe the recovery primitives that are available in this domain and the nature of faults from which it is possible to recover using them.

Recovering general applications involves two primitives: rollback of a failed process to a prior committed state, and re-execution from that state. Two general constraints arise from these primitives. First, events that are rolled back and not re-executed must be undoable. Second, events that are re-executed must be redoable without causing inconsistency. In general, consistent recovery requires that no visible events be forgotten (rolled back and not re-executed). Hence the constraint of rollback recovery is easy to ensure, because only non-visible events (e.g. store instructions) will be rolled back and not re-executed, and these events are generally undoable.

The constraint of re-execution is more challenging, because it takes some work to make some events redoable. For example, for message send events to be redoable, the system must either tolerate or filter duplicate messages. Similarly, for message receive events to be redoable, received messages must be saved either at the sender or receiver so they can be re-delivered. Note that these re-execution requirements are similar to the demands made of systems that transmit messages on unreliable channels (e.g. UDP)—such systems must already work correctly even with duplicate or lost messages. For many recovery protocols, these systems' normal filtering and retransmission mechanisms will be enough to support the needs of re-execution recovery. For other protocols, messages will have to be saved in a recovery buffer at the sender or receiver so they can be redelivered should a receive event be re-executed.

Recovery involving re-execution also requires that visible events be redoable. As a result, general rollback+re-execute recovery requires an equivalence function such as the one we have assumed that tolerates duplicate visible events.

We next describe the nature of faults from which it is possible to recover using these recovery primitives. There are two constraints on the types of faults from which applications can recover: the fault must obey the fail-stop model [Schneider84], and the fault must be non-deterministic.

First, the fault must be fail-stop. Specifically, the system must detect the error and stop before committing buggy state. If the system fails to do so, recovery will start from a buggy state and the system will assuredly fail again. As with most existing work, we assume that faults are fail-stop. Second, the fault must be non-deterministic (a so-called Heisenbug [Gray86]). Otherwise recovery will simply re-execute the buggy code and the program will re-fail.

Our recovery work is relevant for all faults that obey these two constraints, regardless if the fault occurs in the hardware, operating system, or application. For example, if the hardware fails, the fault must be transient, or a backup system must be used for recovery that does not suffer from the same fault. If the operating system crashes, the crash must be non-deterministic and must not commit buggy operating system state (or corrupt and commit application state). If the application crashes, the application and fault must be non-deterministic and not commit buggy application state. Fortunately, most faults (though not all) are non-deterministic and obey the fail-stop model [Gray86, Chandra98].

Section 2.3.3 unifies these two constraints and explores the interaction of different recovery protocols with the fail-stop model.

2.2.4 Special theory of consistent recovery

We next turn our attention to the problem of achieving for general applications consistent recovery as we have defined it, subject to the assumptions we have made. Our work in this chapter is concerned with the actions needed to ensure consistent recovery. We do not address secondary (though still important) issues such as recovery time and stable storage space.

We have already defined commit events. In order to recover, all failed processes simply continue from their most recently committed state. If *f* is a subset of *C*, we define the global state Σ_f to be made up of the last committed state of all the processes in *f* and the current state of all the processes in $C - f$. The global state Σ_f represents the global state from which recovery would begin were the processes in *f* to fail at this instant in time.

The challenge in building recoverable applications is ensuring that recovery will always be consistent, no matter which processes fail or when. By committing judiciously, processes can guarantee consistent recovery. The following theorem informs how processes must commit in order to do so.

Theorem 1

> Recovery is guaranteed to be consistent for computation *C* if and only if at all times *t*, $\forall f \subseteq C$, every sequence made up of the visible events executed by *C* up to time *t*, followed by the visible events of each execution of *C* from the

state Σ_f, must be equivalent to the sequence of visible events resulting from some legal execution of *C*.

Proof of sufficiency for Theorem 1

We intend to prove that if at all times *t*, $\forall f \subseteq C$, every sequence made up of the visible events executed by *C* up to time *t*, followed by the visible events of each execution of *C* from the state Σ_f, is equivalent to the sequence of visible events resulting from some legal execution of *C* then recovery is guaranteed to be consistent.

At all times and $\forall f \subseteq C$, all executions from Σ_f extend a sequence of visible events that is equivalent to a legal one.

By our premise.

Thus, any subset of processes can fail at any time and recovery will be consistent.

Proof of necessity for Theorem 1

We must prove that if at some time t there exists an $f \subseteq C$ in which some execution from Σ_f extends a sequence of visible events that is equivalent to no legal execution of C, then recovery may be inconsistent.

The processes in *f* could fail at time *t* and recover to state Σ_f.

By the definition of Σ_f.

There exists a possible execution from Σ_f that would result in a sequence of visible events equivalent to no legal execution.

By our premise.

Thus, *C* could follow this execution after its failure and recovery would be inconsistent. *QED*

Since only the visible events executed by the computation can affect the consistency of recovery, we only need to check that this invariant is upheld whenever a visible event is executed. That is, immediately before doing a visible event *e*, a process *p* must ensure that by executing *e* it will not cause the computation to violate Theorem 1. If *p* determines executing *e* may cause Theorem 1 to be violated, it must take steps to correct the situation, which will most likely involve *p* committing its state, or it asking other processes to commit their state.

What might cause recovery to be inconsistent? A process can execute events that we call *non-deterministic* which can lead to inconsistency. A non-deterministic event is a transition from a

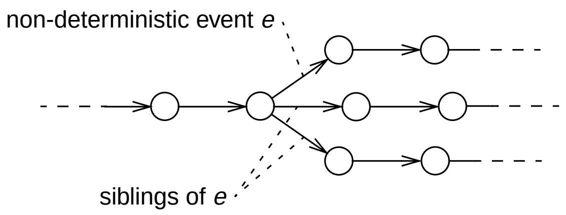


Figure 2.3: Non-deterministic events

state that has multiple next states possible during recovery. The other possible events out of that state we call *sibling* events. Figure 2.3 shows how a non-deterministic event *e* and its siblings might appear in a process's state diagram. In real systems, non-deterministic events correspond to such acts as reading user input, taking a signal, checking the real-time clock, and so on.

Non-deterministic events must be handled specially in recovery because they form the branching point between two paths of execution for a process. Imagine a process like the one shown in Figure 2.4. This process commits its state, executes a non-deterministic event e^1 followed by a visible event e^2, and then fails. During recovery, the process might execute a sibling of e^1 (e^3) rather than e^1 itself. Event e^3 would then lead to the execution of visible event e^4. It should be clear however that there is no legal execution of this process that would result in the execution of both e^2 and e^4. Therefore, should it fail and recover in this manner, its recovery would be inconsistent.

Non-deterministic events are also the source of inconsistent recovery in computations with more than one process, since message send and receive events can propagate dependencies on non-deterministic events between processes. Thus, mishandling of a non-deterministic event can cause other processes to output visible events that cannot appear in any single execution of the distributed computation, as shown in Figure 2.5.

Non-deterministic events are central to the challenge of upholding Theorem 1. Applications with no non-deterministic events have only one possible execution, and thus only one possible legal sequence of visible events. Evaluating the state of this computation to ensure it upholds Theorem 1 is easy: inspect the visible events output by the computation and make sure they are equivalent to a prefix of the one sequence of visible events allowed.

Applications with lots of non-determinism are profoundly difficult to analyze, however. Evaluating the state of such a computation to check whether it upholds the invariant involves gen-

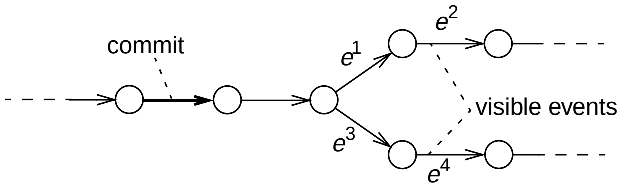


Figure 2.4: Single process inconsistency from non-determinism

A single process can recover inconsistently if it executes two visible events that could never coexist in a legal execution of the process. If this process executes e^1 and e^2, then fails and recovers from its last commit, it could execute e^3 and e^4 during recovery. Although the process can output visible events e^2 or e^4, no legal execution contains both. Thus recovery in this scenario would be inconsistent.

erating all possible executions from global state Σ_f for all *f*, and comparing them with all possible legal executions of the computation for equivalence. Each process in the computation has at least 2^n different executions from that process's initial state, where *n* is the number of non-deterministic events in the process's next-state function. Clearly, the computation as a whole has at least as many possible executions as its process with the largest number of possible executions. Since in real applications non-deterministic events are quite common (that is *n* can be large), we conclude that fully evaluating whether a computation upholds Theorem 1 is not tractable.

Our goal with this work is to provide a simple rule that is necessary and sufficient to guarantee consistent recovery for general applications. Although Theorem 1 provides such a rule, we have argued applying it is not tractable for general applications. To get tractability, we will make a simplifying assumption about what application behavior will lead to inconsistent recovery. For the remainder of our discussion, we will assume:

Assumption 1

> In every computation *C*, each non-deterministic event that causally precedes a visible event e^1 has a sibling event that can causally precede a different visible event e^2, and that e^1 and e^2 are not both contained in any legal execution of *C*.

Assumption 1 turns out to be a reasonable assumption to make for real systems. For most real state machines, if a non-deterministic event leads to the execution of a visible event, a sibling of that non-deterministic event will lead to a different visible event. Furthermore, a path of execu-

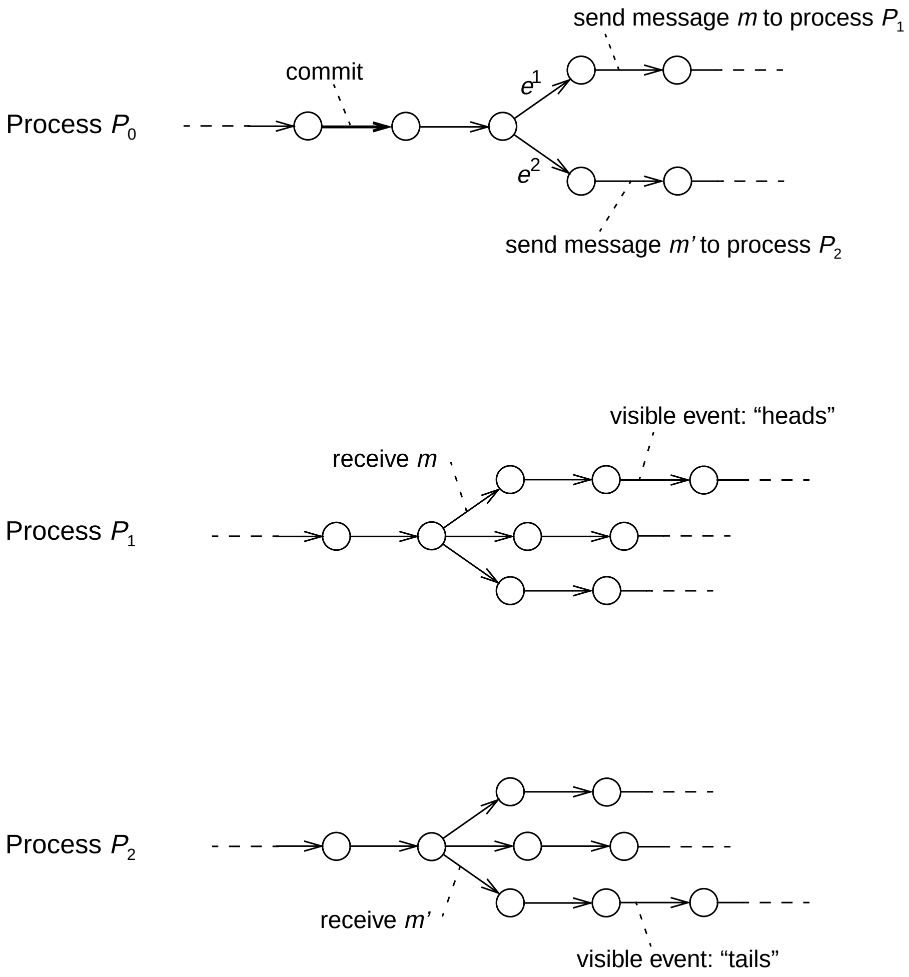


Figure 2.5: Multi-process inconsistency from non-determinism

In this example, we see a computation consisting of three communicating processes. Based on the results of a coin flip, P_0 executes one of two non-deterministic events. If the flip is “heads” it executes event e^1 and sends a message describing the result of the flip to P_1. If the flip is “tails”, it executes event e^2 and sends the result of the flip to P_2. Should P_0 fail after sending its message, it could recover back to its recently committed state, redo the coin flip, and send another message, this time to the opposite process. Although it is clearly legal for either P_1 to output “heads” or P_2 to output “tails”, it is not legal for the computation to output both “heads” and “tails”. Thus, recovery in this case would be inconsistent.

tion through a non-deterministic event does not typically meet up with a path of execution through a sibling of that non-deterministic event. In other words, paths of execution do not often join in real systems. Thus, Assumption 1 is a reasonable one to make for real systems. We discuss the implications of not making this assumption in Section 2.2.6.

By now, it should be evident that the problem of achieving consistent recovery involves managing the non-deterministic, visible, and commit events a computation executes. We next provide a theorem that makes this relationship concrete, and answers the question of when processes must commit in order to guarantee consistent recovery after failures.

Theorem 2

> A subset of computation *C* can fail, and recovery is guaranteed to be consistent if and only if $\forall e_q$ executed by all processes $q \in C$, where e_q is a visible event or a commit event,
>
> $\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q,
>
> $\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$.

This theorem states that to guarantee consistent recovery, computations must meet three requirements. First of all, the computation must ensure that each non-deterministic event that causally precedes a visible event is committed. Second, it must ensure that the commit event that commits that non-deterministic event causally precedes the visible event in question. Finally, the computation must ensure that each commit event causally follows no uncommitted non-deterministic events. We call these requirements the theorem's *commit invariant*. The theorem also states that if the computation does not uphold the commit invariant, the possibility of inconsistent recovery as the result of an untimely failure exists.

As mentioned in Section 2.2.2, the definition of consistent recovery provides both safety and liveness constraints. We will split Theorem 2 into two parts, one that guarantees safety, and one that guarantees liveness.

Theorem 2 - Safety

> A subset of computation *C* can fail, and each partial execution of *C* will be equivalent to a failure-free partial execution of *C* if and only if $\forall e_q$ executed by all processes $q \in C$, where e_q is a visible event,
>
> $\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally

precedes e_q,

$\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$.

Theorem 2 - Liveness

A single failure of some subset of *C* cannot prevent indefinitely some process from executing a visible event if and only if $\forall e_q$ executed by all processes $q \in C$, where e_q is a commit event,

$\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q,

$\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$.

We will prove separately the sufficiency and necessity of both Theorem 2- Safety and Theorem 2 - Liveness.

In our use of causal precedence so far, we have only needed to relate events through cause and effect in failure-free scenarios. For the purposes of the following proof however, we will need a precise definition of causality that works for executions that include failures. Given two events e^a and e^b, we say e^a $causally'$ $precedes$ e^b if:

e^a and e^b are executed by a single process in that order with no failure in between, e^a is a message send by some process and e^b is the corresponding receive, or e^a causally' precedes *e* and *e* causally' precedes e^b.

We may also say e^b $causally'$ $follows$ e^a if e^a causally' precedes e^b. This notion of causal precedence is the same as our normal definition when processes do not fail.

Proof of Sufficiency of Theorem 2 - Safety

We intend to prove that if $\forall e_q$ executed by all processes $q \in C$, where e_q is a visible event, $\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q, $\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$, then all partial executions of *C* will be equivalent to a failure-free partial execution of *C*, despite failures.

Consider the universe of partial executions possible for the computation *C*, including executions in which the computation has failed and recovered. We can think of each member in the universe being a set of events partially ordered by causal precedence.

We can divide the universe into two regions. One region contains all the executions in which no process in *C* fails. We call this region *NoFail.* The other region contains the executions of *C* in which at least one process fails. We call this region *Fail*. Figure 2.6 depicts this partitioning of the universe of partial executions.

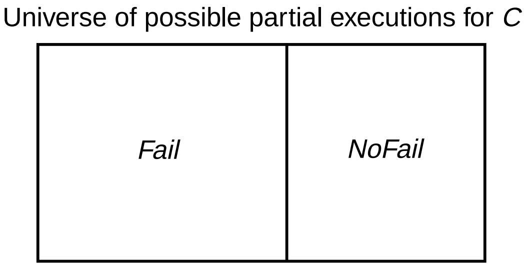


Figure 2.6: Universe partitioned into *Fail* and *NoFail* regions.

We can further divide the *Fail* region into two parts, as shown in Figure 2.7. The first part we call *Redo*. It contains the executions in *Fail* in which all failed processes exactly redo their pre-failure paths during recovery. In other words, the failed processes execute during recovery no siblings of pre-failure non-deterministic events. The second portion of *Fail* we call *Deviate*. It contains the executions in *Fail* in which at least one failed process deviates from

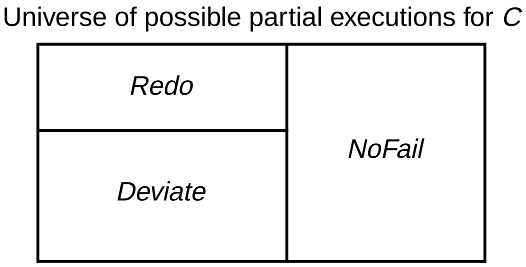


Figure 2.7: *Fail* partitioned into *Redo* and *Deviate* regions.

its pre-failure path by executing a sibling of a pre-failure non-deterministic event.

Finally, we can divide *Deviate* into two sub-regions. For each execution in *Deviate*, some process *p* executes a pre-failure non-deterministic event e_p^{ND}, and a sibling of e_p^{ND} during recovery. We call *Exposed* the portion of *Deviate* in which some e_p^{ND} causally' precedes a visible event, or is itself a visible event. The remaining portion of *Deviate* in which no process's e_p^{ND} is visible or causally' precedes a visible event we call *Hidden*.

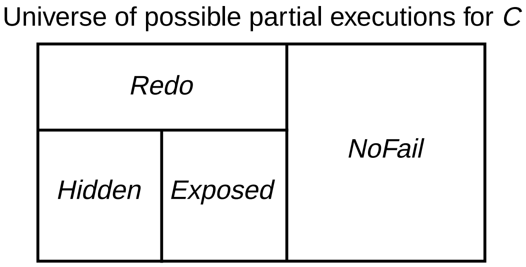


Figure 2.8: *Deviate* partitioned into *Hidden* and *Exposed* regions.

With the execution divided as in Figure 2.8, we can make several observations. First of all, the executions in *NoFail* are all equivalent to a failure-free execution.

> *Each execution in NoFail is failure-free. Therefore, each is equivalent to itself.*

For each execution *E* in *Redo*, there exists an equivalent execution in *NoFail*.

> *E is the same as a failure-free execution in NoFail, except that the failed processes in E re-execute the events between their last commit and their failure. Since our assumption of redoability states re-executing events cannot cause inconsistency and our equivalence function tolerates duplicated visible events, E is equivalent to an execution in NoFail.*

Thus all executions in *Redo* have consistent recovery.

Recall that for all the executions in *Hidden*, at least one process *p* executes a non-deterministic event e_p^{ND} pre-failure, and a sibling of e_p^{ND} during recovery, and e_p^{ND} does not causally' precede a visible event and is not visible itself. We will provide a transformation that will take an execution *E* from

Hidden and transform it into an execution E' from *Redo* with an identical sequence of visible events.

To create E', remove e_p^{ND} from E and all the events that causally' follow e_p^{ND}.

E and E' have identical sequences of visible events.

> *The transformation does not affect E's visible events since* e_p^{ND} *is not a visible event, and* e_p^{ND} *does not* $causally'$ *precede any visible events. Thus the transformation removes only non-visible events.*

E' is an execution in *Redo*.

> E' *is an execution in which the computation failed immediately before each process p executed* e_p^{ND}*, then recovered along the path through the sibling of* e_p^{ND}.

Thus, for each execution in *Hidden* there is an execution in *Redo* with an identical sequence of visible events. Since we have established that the executions in *Redo* all have consistent recovery, it follows that the executions in *Hidden* do as well.

The only remaining region is *Exposed*. By process of elimination, any executions with inconsistent recovery are in the *Exposed* portion of the execution universe.

Since *C* commits according to our premise, no execution of *C* will fall in the *Exposed* region.

> *Every non-deterministic event executed by C that causally precedes a visible event must have a commit after it. Thus it is impossible for any process in C to execute a non-deterministic event* e_p^{ND} *that* $causally'$ *precedes a visible event, fail, then execute a sibling of* e_p^{ND} *during recovery.*

Thus, the sufficiency of Theorem 2 - Safety is proved.

It remains to show the necessity of Theorem 2 - Safety. Proving necessity amounts to showing that if *C* does not commit correctly according to our premise, then the *Exposed* region of *C*'s execution universe is not empty and some execution there has inconsistent recovery.

Proof of Necessity of Theorem 2 - Safety

We intend to prove that if at some time $\exists e_q$ executed by a process $q \in C$, where e_q is a visible event, and $\exists e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q, and $\forall e_p^j \in h_p$ where e_p^j is a commit event, $j \le i$, then there exists a partial execution of *C* that is not equivalent to any failure-free execution of *C*.

Imagine that process *p* fails immediately after process *q* executes e_q. Without loss of generality, assume that e_p^i is the first non-deterministic event *p* executed after a commit by *p*. There are no commits in h_p after e_p^i.

By our premise that $\forall e_p^j \in h_p$ *where* $i \le j$, e_p^j *is not a commit event.*

Thus process *p* will begin recovery in a state σ that was reached before executing e_p^i in *p*'s pre-failure execution.

Process *p* could execute a sibling of e_p^i during recovery.

There are only deterministic events between σ *and* e_p^i. *Thus, p will execute post-failure up to the initial state of* e_p^i, *from there it can transition to a state other than the one to which process p transitioned by executing* e_p^i.

We call this sibling e_p^{sib}. There can exist a process *l* that executes during recovery a visible event *e*' that causally follows e_p^{sib}.

By Assumption 1.

e_q and e_l do not both appear in any legal execution of *C*.

By Assumption 1.

Thus, we have described a possible partial execution of *C* that is not equivalent to any failure-free execution of *C* as a result of its having violated the commit invariant established in Theorem 2.

We have established so far that upholding Theorem 2 - Safety is necessary and sufficient to guarantee that, despite failures, all partial executions of a computation are equivalent to some failure-free partial execution of the computation. Given that computation *C* is trying to uphold the commit invariant, what happens if some process $k \in C$ tries to execute a visible event *e*, and finds that some non-deterministic event e_i^{ND} that causally precedes *e* cannot be committed? This scenario could arise if process *i* has already failed and aborted e_i^{ND}. In order to uphold the commit invariant, process *k* will have no choice but to avoid executing *e* and to abort back to a prior committed state such that all future events it executes will not causally depend on e_i^{ND}.

However, a computation that commits naively could conceivably preserve a dependency on e_i^{ND} in its committed state, preventing it from aborting this dependency. Such a scenario is shown in Figure 2.9. At some point in its computation, process P_3 wants to execute visible event e_3^3, and notices that the event would depend on uncommitted non-deterministic event e_1^1. As a

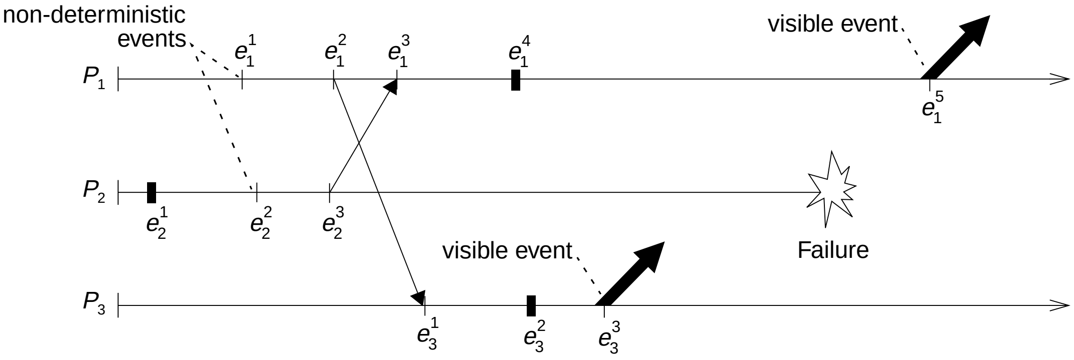


Figure 2.9: A problematic computation for liveness

This figure depicts a computation of three processes in a *space-time* diagram. The execution of the processes proceeds in time from left to right. Solid arrows represent messages sent between processes, black boxes represent commit events, and heavy short arrows represent visible events. This computation has violated the commit invariant. As a result, after P_2 's failure it cannot make the needed commits when P_1 attempts to execute visible event e_1^5.

result, it asks process P_1 to commit its state. P_1 obliges by executing commit event e_1^4. Next, P_3 commits its own state and executes its visible event. Process P_2 then fails. When process P_1 later decides to execute a visible event e_1^5 that depends on uncommitted non-deterministic event e_2^2, it finds P_2 cannot commit after e_2^2 because P_2 has failed and aborted that event. Even worse, P_1 finds it has preserved its dependency on e_2^2 in its last committed state and therefore cannot abort its dependency on e_2^2. In this scenario, process P_1 is livelocked, and the computation cannot complete.

The computation in Figure 2.9 got into trouble because it only upheld Theorem 2 - Safety but not Theorem 2 - Liveness. That is, it only ensured that no visible event executed by the computation causally followed an uncommitted non-deterministic events. It left open the possibility that a commit event could depend on an uncommitted non-deterministic event, as e_1^4 does on e_2^2.

We will next prove the sufficiency of Theorem 2 - Liveness.

Proof of Sufficiency for Theorem 2 - Liveness

> We intend to prove that if $\forall e_q$ executed by all processes $q \in C$, where e_q is a commit event, $\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q, $\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$, then a single failure cannot prevent indefinitely some process from executing a visible event.
>
> To show that committing according to our premise guarantees that the liveness constraint is met, we need to show the following: whenever a process *p* in *C* wants to execute a visible event *e* but finds there is a non-deterministic event e_i^{ND} that causally precedes *e* that it cannot commit, *p* can abort to its last committed state and be assured that all events it executes from that point on will not causally depend on e_i^{ND}.
>
> $\forall q \in C$, given that e_q is the last commit event executed by process *q*, e_q does not causally depend on any uncommitted non-deterministic events.
>
> > *By our premise that $\forall e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q, $\exists e_p^j \in h_p$ where e_p^j is a commit event, $i \le j$, and either e_p^j causally precedes e_q or $e_p^j = e_q$.*
>
> Since the last commit event executed by each process does not causally depend on any uncommitted non-deterministic events, each process can safely abort a dependency on any uncommitted non-deterministic event by aborting back to its last committed state. Thus, the sufficiency Theorem 2 - Liveness is proved.

Our final task is to prove ensuring no commit event causally follows an uncommitted non-deterministic event is necessary to guarantee liveness. For the following proof we make the following assumption:

Assumption 2

> Once a computation executes a commit event *e* in some execution, every completion of the execution will execute a visible event that causally depends on *e*.

Making this assumption allows us to prove necessity without tackling the computationally thorny issue of computing exactly which commit events may causally precede some later visible event (this problem is hard for the same reason computing which non-deterministic events may causally precede visible events is hard, as discussed earlier in this section). Furthermore, most commit events do causally precede visible events in real systems.

Proof of Necessity for Theorem 2 - Liveness

We intend to prove that if at some time $\exists e_q$ executed by a process $q \in C$, where e_q is a commit event, and $\exists e_p^i \in h_p$ for some $p \in C$, where e_p^i is non-deterministic and causally precedes e_q, and $\forall e_p^j \in h_p$ where e_p^j is a commit event, $j \le i$, then a single failure can prevent indefinitely some process from executing a visible event.

At some point in the execution of *C*, there exists a visible event e_r^{vis} executed by process *r* such that e_q causally precedes e_r^{vis}.

> *By Assumption 2.*

e_r^{vis} causally follows e_p^i.

> e_r^{vis} *causally follows* e_q*, which causally follows* e_p^i*. Therefore* e_r^{vis} *causally follows* e_p^i *by the definition of causal precedence.*

Consider a scenario in which process *p* has failed and aborted event e_p^i by the time process *r* gets around to executing e_r^{vis}. Process *r* cannot execute e_r^{vis} because it depends on non-deterministic event e_p^i which it cannot commit. Therefore, process *r* will have to abort its dependency on e_p^i. Since process *r* causally depends on e_p^i by causally depending on e_q, process *r* will also have to ask process *q* to abort its dependency on e_p^i.

> *Assumption 2 implies that without aborting process q's dependency on* e_p^i *and simply aborting locally, process r will regain its dependence on* e_p^i *from process q and again try to execute a visible event that depends on* e_p^i*.*

Process *q* cannot abort its dependency on e_p^i.

> e_q *is a commit event that causally depends on* e_p^i*. Therefore, process q has preserved its dependency on* e_p^i *and cannot abort it.*

Thus, process *r* cannot abort its dependency on e_p^i either and as a result process *r* cannot continue execution. Thus, we have proved the necessity of Theorem 2 - Liveness.

Since we have proved the sufficiency and necessity of both Theorem 2 - Safety and Theorem 2 - Liveness, we have proved that upholding Theorem 2 is both necessary and sufficient to guarantee consistent recovery under the assumptions we have made. *QED*

Theorem 2 has a corollary concerning the possibility of achieving consistent recovery in the presence of events that are both visible and non-deterministic.

Corollary 1

> Consistent recovery is impossible to guarantee if any process in a computation executes an event that is both non-deterministic and visible, unless the event can be made atomic with a commit.

This result follows naturally from the theorem. Since a commit event cannot be executed in between such an event's non-deterministic and visible portions, we conclude the theorem's invariant is impossible to uphold for these events unless the event can be executed atomically with a commit.

The theorem defines how all processes must commit in order to guarantee consistent recovery, basing the commit decision on the history of events executed by each process. Given that a computation that has failed and recovered will want to continue to uphold the commit invariant after recovery, what local process histories should it use? The process histories should mirror the effects of recovery on each process. Since each failed process begins recovery in its last committed state, all events in that process's local history after the last commit event in the history should be removed. Events the process executes during recovery and after then get appended to the new history like usual.

2.2.5 Upholding the theorem

The commit invariant can be broken up into three separate requirements:

1.) There exists a commit event after every non-deterministic event that causally precedes a visible event *e*.

2.) The commit event causally precedes *e*.

3.) Every commit event does not causally depend on any uncommitted non-deterministic event.

There are many ways an application can uphold the commit invariant in order to guarantee consistent recovery, each with a different set of trade-offs between commit frequency and implementation effort.

A protocol that forced a commit immediately after every non-deterministic event would clearly uphold the commit invariant. This protocol trivially satisfies requirement 1 since it immediately commits all non-deterministic events, a set that includes those non-deterministic events that causally precede visible events. This protocol also satisfies requirement 2: any visible event causally preceded by a non-deterministic event cannot come between the non-deterministic event and the commit immediately following. Therefore, if the non-deterministic event causally precedes a particular visible event, its commit event will too. Finally, since no uncommitted non-deterministic events exist under this protocol, no commit can depend on one, meeting requirement 3. Since this protocol meets all three of the commit invariant's requirements, it guarantees consistent recovery. We call this protocol *commit after non-deterministic* (CAND). It has the advantage of not needing to identify which application events are visible.

Since application non-determinism can manifest in many forms, identifying which events an application executes are non-deterministic can be challenging. Rather than perform this identification, an application may instead choose to uphold the commit invariant by committing immediately before every visible or send event. By committing immediately before each visible event, a process guarantees a commit after any of its non-deterministic events that causally precede the visible event. Each process also commits immediately before every message send event, ensuring a commit after any of its non-deterministic events that may causally precede a downstream visible event. Thus, this protocol upholds requirement 1. Since, each commit immediately before a visible event causally precedes that visible event, and each commit immediately before a send event must also causally precede any downstream visible events, this protocol also meets requirement 2. By committing immediately before every message send event, each process ensures that it will not pass a dependency on an uncommitted non-deterministic event to the message recipient. Thus, every commit executed by the message recipient will not depend on any of the sender's uncommitted non-deterministic events. Furthermore, since the commit event before each visible event commits all of that process's non-deterministic events, this protocol upholds requirement 3. Since this protocol upholds all aspects of the commit invariant, it guarantees consistent recovery. We call it *commit prior to visible or send* (CPVS). One can view CPVS as treating sends events as if they were visible events, since they can lead to visible events on other processes.

If an application is willing to identify both visible and non-deterministic events, it can use a protocol in which it commits between every non-deterministic event and visible or send event. After a process executes a non-deterministic event under this protocol, it will commit before a subsequent visible or send event. In so doing, it ensures a commit after the non-deterministic event if it causally precedes a visible event on the same process. It also ensures a commit after the non-deterministic event if it causally precedes a send event on the same process, since the send event may in turn lead to a visible event on another process. Thus this protocol meets requirement 1. Each commit executed before a visible event causally precedes that visible event, and each commit before a send event causally precedes any downstream visible events. As a result, this protocol meets requirement 2. Finally, each commit event executed before a visible or send event obviously cannot depend on an uncommitted non-deterministic event on the same process, since each commit event commits all that process’s non-deterministic events. Furthermore, since this protocol guarantees a commit before any send event that causally follows a local non-deterministic event, no process’s commit events depend on another process’s uncommitted non-deterministic event. Thus, this protocol meets requirement 3. Since this protocol upholds all three requirements of the commit invariant, it guarantees consistent recovery. We call this protocol *commit between non-deterministic and visible or send* (CBNDVS). This protocol will always execute the same or fewer number of commits than CAND or CPVS.

CAND, CPVS, and CBNDVS can lead to a large number of commits, as we will see in Section 6.2.4. Since commits can be slow, it may make sense for performance reasons to reduce commit frequency. There are two orthogonal classes of techniques that can help reduce commit frequency: treating as few events as possible as visible, and converting non-deterministic events into deterministic events.

As mentioned above, we can think of the CPVS protocol as treating send events as visible since they may lead to visible events on the receiving process. To reduce commit frequency while still guaranteeing consistent recovery, applications may seek to identify which events are truly visible, and which are not. For example, applications can avoid treating send events as visible if the application uses an agreement protocol to commit all processes atomically before any process executes a visible event. Committing in this manner ensures that all processes’ non-deterministic events are committed, including those that causally precede visible events. Thus, this protocol meets requirement 1 of the commit invariant. Furthermore, the agreement protocol will add messages to the computation to force each visible event to causally follow every commit it initiates. Thus, this protocol also meets requirement 2. Finally, since all processes commit together under this protocol, no process’s commit can causally depend on an uncommitted non-deterministic event. Therefore, this protocol also meets requirement 3. Since this protocol meets all three of the commit invariant’s requirements, it too guarantees consistent recovery. If visible events are less frequent than non-deterministic events or message sends, such an approach can result in fewer commits than CAND, CPVS, or CBNDVS.

Applications may also reduce commit frequency while upholding the commit invariant using CAND or CBNDVS by making non-deterministic events deterministic. There exist general techniques for performing this conversion, such as logging [Gray78]. In a logging system, the result of a non-deterministic event is appended to a persistent log. The log can then be used during recovery to ensure that each event has the same result during recovery that it had pre-crash. The typical application of logging is to make message receives deterministic, although logging can also be used for other events such as signals and interrupts [Bressoud95, Slye96]. Some recovery protocols endeavor to uphold the commit invariant by making *all* non-deterministic events deterministic, avoiding all commits. We call such protocols *complete logging protocols*.

The commit invariant not only informs the question of when processes must commit to recover consistently. It also addresses the question “how long can a process that is converting non-determinism leave an event non-deterministic without forcing a commit?” The answer: up until the next causally dependent visible event. Hence, the commit invariant suggests a kind of *lazy logging* in which the results of non-deterministic events are queued in a volatile buffer, to be flushed just before the execution of a causally dependent visible event [Elnozahy92]. For some workloads, a lazy protocol could reduce logging overhead significantly by amortizing the cost of writes to stable storage, without making any optimistic assumptions. Applications can also use a simpler version of lazy logging that does not require dependency tracking between processes. In this slightly more eager protocol, processes would simply flush their log tails to stable storage before sending a message or executing a visible event.

Applications can achieve the lower bound on commit frequency by implementing the commit invariant directly. In this scenario, process A would piggyback information on its outgoing messages to inform downstream processes of their dependence on any non-deterministic event executed by A. When some process B later executes a visible or commit event *e*, it first asks the processes that have executed causally preceding non-deterministic events to commit their state, upholding requirements 1 and 3 of the commit invariant. It waits for confirmation of each process’s commit before executing *e* to ensure that all other process’s commits causally precede *e*, upholding requirement 2.

2.2.6 Refining Assumptions 1 and 2

A challenging aspect of future work is to remove Assumption 1 in Section 2.2.4. Recall that under Assumption 1, each non-deterministic event that causally precedes a visible event may have a sibling event that causally precedes a different and conflicting visible event. Recall that this assumption only affects the proof for the necessity of Theorem 2 - Safety. In the absence of such an assumption, Theorem 2 - Safety is still sufficient but may no longer be necessary for all state machines.

For example, the assumption does not hold for the state machine shown in Figure 2.10, and it does not need to commit between executing e^1 and e^2.

Intuitively, the state machine in Figure 2.10 does not need to commit between the non-deterministic event e^1 and the visible event e^2 because recovery will be consistent even if a crash

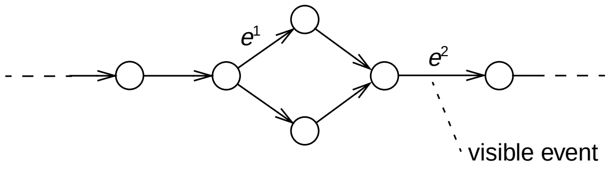


Figure 2.10: Example of state machine in which Assumption 1 does not hold

occurs after executing the visible event and the sibling event is executed during recovery. However, note that it is difficult to conceive of a realistic program whose execution path joins after diverging in this manner. In order to join, the state of the process must be *identical* after taking either branch and executing the visible event.

It is challenging to construct a simple invariant that describes the necessary commit pattern for state machines for which Assumption 1 does not hold. One possibility is to analyze dynamically the computation's execution before executing a visible event and decide whether there exists a recovery path related to that visible event that could lead to inconsistent execution. If so, the computation must commit a subset of its processes to eliminate this possible recovery path. We believe these analyses are computationally intractable for state machines of realistic size, because they seem to require analyzing all possible executions. If it were tractable to analyze all possible executions, one could use the same enumeration to verify formally the correctness of the program. However, an interesting area of future work is to analyze the possible recovery scenarios in a manner that does not require enumerating all recovery paths. For example, programmer assistance may simplify this analysis, or different recovery paths may be able to be analyzed together if they join. It may also be the case that for real state machines it is sufficient to analyze the state transition graph within a fixed radius of a non-deterministic event.

Similar analysis would be necessary to remove Assumption 2. Under Assumption 2, once a computation executes a commit event *e* in some execution, every completion of the execution will execute a visible event that causally depends on *e*. Note that this assumption only affects the proof of necessity for Theorem 2 - Liveness. If it does not hold, Theorem 2 - Liveness is sufficient, but not necessary for all state machines.

To remove this assumption, the recovery system must be able to determine at run time that a commit event will never causally precede any visible event executed by the computation. If it can do so, then it is possible to construct a theorem similar to Theorem 2 - Liveness that is necessary without Assumption 2. Like in the case for Assumption 1 however, determining whether a commit event will ever causally precede a visible event seems to require analyzing all possible executions from the state at the time of the commit. As before, we believe this analysis is intractable for real state machines, although it could prove an interesting area of future research.

Finally, we should point out that classical recovery research also makes assumptions equivalent to Assumptions 1 and 2, as consistent cuts are also not necessary for safety and liveness without them.

2.3 Applying the Theory

The theory has a number of practical uses. As we have seen, it helps make evident new recovery protocols. We next turn our attention to examining how the theory can be used to unify existing recovery protocols, reveal shortcomings in existing recovery research, and explore interactions with the fail-stop model.

2.3.1 Unifying existing recovery protocols

All existing recovery research can been seen as providing different techniques for upholding the commit invariant. Each varies in its approach to identifying non-deterministic and visible events, converting events, tracking causal dependencies, and initiating commits. In order to show how a number of these protocols fit into our theory of consistent recovery, we will use the theory to describe the workings of several representative protocols.

2.3.1.1 Pessimistic logging

Pessimistic logging protocols endeavor to greatly reduce or eliminate application non-determinism, allowing them to uphold the commit invariant without committing before most visible events. For the applications traditionally considered, most of these protocols focus on making message receive events deterministic [Powell83, Borg89].

In Targon/32, message data and receive order are logged in the input queue of a backup process running on another processor [Borg89]. Since the backup process is the one that will be used for recovery, its receive events are guaranteed to execute with the same result during recovery as they did for the primary process before its failure. Thus message receive events are made deterministic. After a failure by a process, the system recovers the process by activating its backup and letting it roll forward through its queue of received messages. Since Targon/32 endeavors to recover general Unix applications, it has to contend with sources of non-determinism other than message receive events, such as signals. When a process receives a signal, the system ensures consistent recovery by checkpointing the process to the memory of the backup processor after the delivery of the signal, as mandated by the commit invariant.

Bressoud and Schneider’s hypervisor-based system extends the class of non-deterministic events that are made deterministic from message receives to general interrupts [Bressoud95]. The hypervisor makes interrupts deterministic by delivering them at the end of fixed intervals called epochs and logging them to a backup processor, which will also deliver them at the end of the epoch.

2.3.1.2 Sender-based logging

Sender-based logging (SBL) allows applications whose only non-deterministic events are message receives to survive the failure of a single process in the system [Johnson87]. It works by making all receive events deterministic to avoid ever having to commit. Receive events are made deterministic by logging messages and the order in which they were delivered in the volatile memory of the sender. After a process fails, surviving processes resend the logged messages and inform the recovering process of the order in which to process them, guaranteeing deterministic re-execution of receive events during recovery. Thus, applications with no other sources of non-determinism need not commit before executing visible events.

2.3.1.3 Causal logging

Like other logging protocols, causal logging protocols uphold the commit invariant by converting all non-deterministic events into deterministic ones, avoiding ever having to commit.

Consider a message *m* sent from process *p* to process *q*. In Family-Based Logging (FBL) and in Manetho, the contents of *m* are logged at the sender [Alvisi94, Elnozahy92]. However, the order in which *m* is delivered to *q* (the true non-determinism in the message delivery) is not logged until the last possible moment.

For FBL, that moment comes once *q* next executes a send event. Since FBL tries to survive the failure of any single process, it suffices for *q* to piggyback the receive sequence number (*rsn*) for *m* on its next send. The recipient of that message is then responsible for adding the piggybacked *rsn* to its log before processing the message. Once a process has received confirmation that all its *rsn*’s have been logged by other processes (these confirmations are piggybacked on other application messages), it can safely execute a visible event.

The situation is a bit trickier for Manetho, because it aims to survive the simultaneous failure of all processes in the computation. Each process keeps track of the relative order of all non-deterministic events on which its state depends (across all processes) in an *antecedence graph* (*AG*). When a process sends a message to another, it piggybacks its *AG* on the message to aid maintenance of the recipient's *AG*. Before any process executes a visible event, it must simply write its current *AG* to stable storage to uphold the commit invariant.

Once a process fails under either protocol, the surviving processes and stable storage can provide the failed process with the messages it received pre-crash, and the order in which to process them (either as *AG*'s or *rsn*'s).

2.3.1.4 Optimistic logging

Optimistic logging upholds the commit invariant by making all an application's non-deterministic events deterministic. Under this protocol, each process writes periodic checkpoints and a message log asynchronously to stable storage [Strom85]. Each process maintains a vector clock which summarizes the state of every process's asynchronous logging efforts. Before executing a visible event, a process can inspect its vector clock to determine if the commit invariant is currently upheld. If not, the process simply waits until it is.

2.3.1.5 Coordinated checkpointing

Coordinated checkpointing protocols uphold the commit invariant by committing right before each true visible event in the system. They employ an agreement protocol to avoiding treating send events as visible, and to force the correct causal ordering of commits and visible events. Such an approach can reduce commit frequency without requiring processes to track their causal dependencies. When some process wants to execute a visible event, it first coordinates with the other processes in the application to ensure that all processes commit atomically [Koo87]. If any process cannot perform its commit, all processes are aborted back to their last commit. In essence, these protocols directly move the recovery line forward when some process executes a visible event.

2.3.1.6 Process pairs

Process pair systems, such as Tandem NonStop, uphold the commit invariant by committing before each visible event [Bartlett81, Gray86]. Process pairs (and primary-backup systems in general) are usually discussed in the context of a client-server system, in which the main type of visible events are a server's response messages to client requests. In process pairs, the primary process checkpoints its state to a backup process before each visible event. Multi-process servers can extend this checkpoint to be a coordinated checkpoint from the primary processes to the backup processes.

2.3.1.7 Protocol space

Each of these recovery protocols represents a different technique for upholding the commit invariant. Each to varying degrees trades off programmer effort and system complexity for reduced commit frequency (and hopefully overhead).

Some protocols focus their efforts on the problem of identifying and reducing non-determinism. Others work to treat as few events as possible as visible events. Each protocol can be seen as representing a point in a two-dimensional space of protocols. One axis in the space represents effort made to identify and possibly convert application non-determinism. The other axis represents effort made to identify and possibly convert visible events. Figure 2.11 shows how the protocols of Section 2.3.1 might appear in such a space.

The origin of this plot represents a protocol that atomically commits every event in an application. A protocol that falls close to the origin on the horizontal axis treats almost all events as non-deterministic, whether they are or not. Protocols that fall further and further out on this axis exert increasing effort to identify a progressively larger portion of the non-deterministic events, eventually committing only the events that truly are non-deterministic. Beyond this point on the axis, protocols begin to exert effort to convert non-deterministic events into deterministic ones.

Similarly, a protocol that falls close to the origin on the vertical axis commits almost all events in case they are visible events. As protocols fall further up the axis they exert more effort to treat fewer events as visible. At some point, protocols treat only send events and true visible events as visible. Beyond that point, protocols exert further effort (such as using an agreement protocol) to allow them not to treat send events as visible.

The distance from the origin in this plane is inversely proportional to commit frequency. That is, the farther a protocol is from the origin, the less frequently most applications using that protocol will have to commit. Thus the farther a protocol is from the origin, the more it can maintain good performance, even with a slow commit. Conversely, a fast commit allows a programmer to guarantee consistent recovery with less effort, i.e. use a protocol that would fall closer to the origin in this plane.

Note that in Figure 2.11 most existing protocols fall very close to the axes. This is most likely due to the historical separation between checkpointing and logging-based distributed recovery research. We believe there is significant potential for efficient recovery protocols in the middle of the plot, however. We explore several of them in Section 6.2.2, and many more are possible.

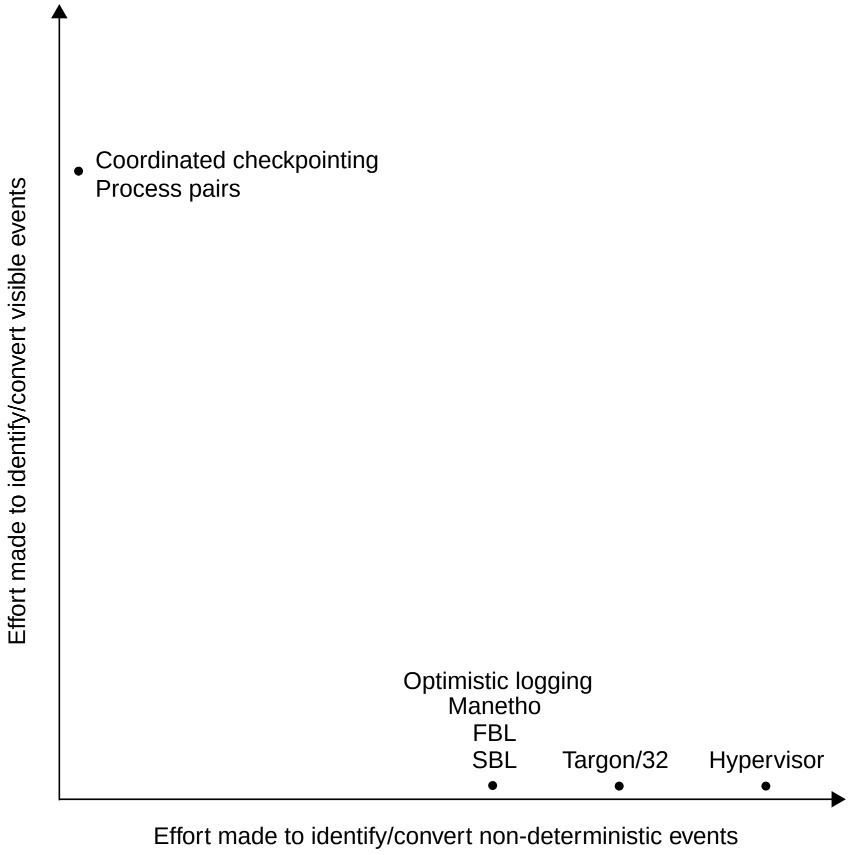


Figure 2.11: Protocol space

We can parameterize the space of all recovery protocols by effort made to identify and convert non-deterministic events and effort made to identify or convert visible events. This plot shows the point in such a space each protocol from Section 2.3.1 might occupy.

2.3.1.8 Logging as commit

As mentioned in Section 2.2.5, applications can use a complete logging recovery protocol that converts all non-deterministic events into deterministic ones, avoiding all commits. However, some researchers may feel more comfortable regarding a complete log as simply a compressed encoding for a commit. Such a viewpoint is compatible with our theory, as well as the protocol space, although the “units” of the horizontal axis of the space have to change to accommodate it. We can instead think of the horizontal axis of the protocol space as being “effort to identify and log completely”. With this parameterization, protocols on the axis, close to the origin again commit most every event in case the event is non-deterministic (using the traditional, non-log-based style of commit). At some point further out on the axis, the protocols have bitten off the task of only committing after the true non-deterministic events. Beyond this point, each protocol begins to add the results of non-deterministic events to a complete log. As long as the application maintains a complete log, each non-deterministic event can be considered committed and the application need not resort to a traditional commit. However, if the application is unable to maintain a complete log (perhaps because it has executed a non-deterministic event it does not know how to log), it must resort to a traditional commit. In other words, a complete log can serve as a commit only as long as there are no unlogged non-deterministic events between the last commit and the current event. Protocols further out the axis have to resort to traditional commits less often. A protocol in which the logging mechanism can log all non-deterministic events always maintains a complete log, committing each event without ever resorting to a standard commit.

2.3.2 Revealing shortcomings in existing recovery research

Our theory of consistent recovery helps make evident a few shortcomings in existing recovery research. At least one proposed protocol does not uphold the commit invariant, and the traditional discussion of the domino effect is incomplete.

Classic recovery papers have mentioned protocols in which applications take local checkpoints immediately *after* message sends, as in Section V of [Koo87]. However, such a protocol does not uphold the commit invariant and therefore it will not guarantee consistent recovery. Consider an application using this protocol in which process A executes a non-deterministic event, sends a message to process B, then crashes before executing the commit. Process B could receive the message and execute a visible event that is causally dependent on the non-deterministic event process A executed. Since that non-deterministic event is not committed when process B executes its visible event, the commit invariant has been violated and consistent recovery cannot be guaranteed.

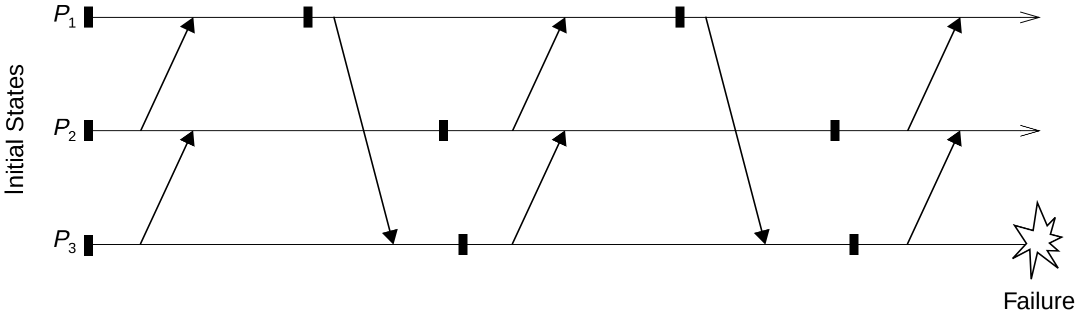


Figure 2.12: Domino effect

Three processes send messages while taking independent local checkpoints (represented by black boxes). After a process fails, all processes coordinate to find a suitable state to which to recover. Unfortunately, it is possible the only global state the system can recover is the initial state.

The *domino effect* is a well-known problem affecting uncoordinated checkpointing protocols [Randell75]. Unfortunately, the traditional discussion of the domino effect contains a hidden assumption that is not always true.

Uncoordinated checkpointing protocols attempt to find a consistent cut preserved in processes' local checkpoints after one or more processes crash. If restarting a failed process from a recent checkpoint aborts a message send, then the recipient of that message is rolled back to a recent checkpoint in order to abort the receive (and thus preserve a consistent cut). Unfortunately, such a technique for aborting processes can cause cascading rollback of the system to some initial state. Consider the computation shown in Figure 2.12. After the failure of process P_3, the recovery protocol attempts to roll back the system to a consistent cut. However, each rollback of a process to a recent checkpoint undoes a send, forcing the further rollback of the receiver. As a result, the only global state that can be recovered in the above computation is the initial state.

Although the traditional description of the domino effect is correct under some circumstances, it presupposes the existence of a non-deterministic event between each checkpoint and the send after it. Without the existence of this non-deterministic event, there is no need to roll back the recipient of a message once the send is aborted—the sender will recompute a state that reflects the send. To our knowledge, the need for these inopportune non-deterministic events as a necessary precondition of the domino effect has never been observed.

Even if there is a non-deterministic event between each checkpoint and the send immediately after it, it is still the case that rollbacks may not cascade. If those non-deterministic events do not causally precede some visible event, no rollbacks are necessary after processes fail.

To further underscore this point, in Section 6.2.2 we will describe several protocols that can lead to commit patterns like the one in Figure 2.12 but that don't lead to the domino effect (e.g. CPVS).

2.3.3 Exploring interactions with the fail-stop model

Of central concern in examining the fail-stop property is the point of execution at which commits occur relative to when buggy code paths are initiated. Commit events have traditionally been viewed as a way to preserve the past work of a process up to the point of the commit. Using the theory, commit events can also be seen as committing all *future* work executed up until the next non-deterministic event. We call this view that commit events preserve later execution *forward commit*. Unfortunately, forward commit increases the chances that failures are not fail-stop by increasing the probability that buggy state is committed. In fact, for some applications, recovery protocols, and failure types, forward commit can rule out the possibility of applications ever failing in a fail-stop manner.

Failures of applications with no non-determinism will always violate fail stop. This observation follows naturally from forward commit: in such applications, all states are always committed, including buggy ones. Thus recovery protocols that eagerly make all of an application's non-deterministic events deterministic also ensure that all application failures will not be fail-stop. Similarly, a recovery protocol that upholds the commit invariant by committing after every non-deterministic event (the CAND protocol described in Section 2.2.5) is guaranteed to commit buggy application state, and all application failures will violate fail-stop. Therefore, systems that use eager, complete logging (e.g. sender-based logging) or CAND cannot recover from application failures. Of course, applications can still survive hardware and operating systems failures using eager, complete logging or CAND as long as those failures are fail-stop (i.e. they don't corrupt committed or logged application state).

There are two general strategies for increasing the likelihood of failures being fail-stop. One way is to enhance the error-detection code (e.g. voting among independent replicas [Schneider84]) so that the faulty system stops sooner. The second way is to defer the commit of possibly buggy state; this deferral gives the error-detection code more time to catch the error. The theory helps with this second method by showing how long it is possible to defer committing state or defer converting a non-deterministic event into a deterministic one. Committing or logging can only be deferred up until the next causally dependent visible event. Protocols that defer these commits, such as CPVS, CBNDVS, lazy logging, and coordinated checkpointing, all increase the likelihood that application failures will be fail-stop.

2.3.4 Protocol space and other design variables

In our discussion of the protocol space of Section 2.3.1.7, we describe how protocols that fall farther away from the origin typically commit less frequently than those that lie closer to the origin. Exploration of the protocol space can be motivated by design factors other than commit frequency, however.

All protocols that fall on the horizontal axis of the space guarantee non-fail-stop application failures. Each of these protocols either commits, or converts all non-deterministic application events, ruling out fail-stop application failures using the reasoning from the last section. As protocols fall further and further up in the space however, they will generally provide more fail-stop application failures. This observation follows from the fact that protocols higher in the space commit less frequently, potentially preserving more application non-determinism. Thus, in choosing a recovery protocol, systems are constrained to choose protocols that fall further up in the space if they want fail-stop application failures.

Although we have not focused on recovery time in our study of recovery, recovery time is an important issue for many systems. Different points in the protocol space will typically imply different recovery times. For example, a protocol at the origin that atomically commits every event will have the shortest possible recovery time because it will never lose more than the current, in-progress event. Protocols that fall farther from the origin will commit less often, and thus potentially will have a larger re-execution component to their recovery. Systems that aim to reduce recovery time might therefore choose protocols that fall closer to the origin of the protocol space.

2.4 Related Work

A few researchers have attempted to provide a general view of recovery research.

Elnozahy et al. provide a thoughtful overview of existing rollback recovery protocols for distributed systems [Elnozahy96]. Their main contribution is to describe and compare the great variety of protocols in the recovery literature. However, the authors do not attempt to unify existing research by providing a theory of consistent recovery.

Alvisi et al. provide a theory of recovery specific to causal logging protocols [Alvisi95]. They show that there exists a family of causal logging protocols that is parameterized by the number of simultaneous failures each protocol can tolerate. Family-Based Logging and Manetho can both be viewed as members of this family [Alvisi94, Elnozahy92]. Their theory is useful primarily in elucidating the relationship between different causal logging protocols; it is less useful for other recovery techniques.

Johnson and Zwaenepoel provide a model for reasoning about recovery in distributed systems that takes into account forward commit [Johnson88]. They describe an algorithm for computing the maximum recoverable state of a distributed system that logs messages and takes checkpoints. They also specify that a process wishing to execute a visible event must ensure that the maximum recoverable state of the computation is beyond the visible event. Since the author's algorithm takes into consideration non-deterministic events, their rule is very similar to our Theorem 2.

Like Johnson and Zwaenepoel, a number of researchers have pointed out components of our theory. For example, several classic papers have observed the importance of guaranteeing a consistent cut before visible events [Strom85, Elnozahy92, Elnozahy96]. Others have considered the relationship between non-deterministic and commit events [Johnson88, Elnozahy92].

Our work goes beyond that of these other researchers in a few important ways. First of all, classical research has relied on a set of separate rules for achieving consistent recovery: one rule defines consistent global states, one rule defines when an application must preserve such a global state, and a final rule relates non-determinism to commits. In comparison, we provide a single invariant that captures the exact relationship between the non-deterministic, commit, and visible events at the center of consistent recovery. Furthermore, we are able to analyze all existing recovery protocols in light of this single invariant, unifying the various approaches to achieving consistent recovery and exposing the existence of a protocol space in which all recovery protocols fall. We also make explicit a number of assumptions lurking behind all recovery protocol research, ours and others'. For example, other researchers all assume an equivalence function like ours, and all make Assumptions 1 and 2, although these assumptions are usually not stated. Finally, we explore the implications of our invariant for the fail-stop assumptions made by most recovery systems.

2.5 Conclusion

With this chapter, we have provided a framework for reasoning about consistent recovery. Our theory of consistent recovery provides a simple invariant that all applications must uphold to mask failures from users.

We have also shown that all existing recovery protocols can be viewed as different techniques for upholding the commit invariant. Logging, checkpointing, and committing studies have historically come from separate camps in the fault tolerance, systems, and database communities. We hope this theory helps clarify the field of recovery research by unifying these separate camps.

For the remainder of this dissertation, we will use the theory to help us build systems that provide failure transparency for general applications.

CHAPTER 3

LIGHTWEIGHT TRANSACTIONS USING RELIABLE MEMORY

In the last chapter, we show that central to the problem of guaranteeing consistent recovery is the need to periodically commit application state. We argue that the simple recovery protocols that fall close to the origin of the protocol space commit their state often, and therefore require a fast commit to be efficient. So that in later chapters we are afforded the maximum flexibility in our exploration of the protocol space, we will next develop a fast commit in the form of our Vista transaction library. Transactions are a very common tool used by fault tolerant applications. They simplify reliable design by letting applications perform a sequence of separate actions in a single, atomic step. However, for all their usefulness, transaction systems' reliance on synchronous writes to stable storage—typically slow disks—reduces their attraction for performance-critical applications. Vista is designed to address this problem by using Rio's reliable memory for stable storage rather than disk.

3.1 Introduction

Any application that modifies a file takes a risk, since a crash during a series of updates can irreparably damage permanent data. Grouping these updates into an *atomic transaction* addresses this problem by ensuring that either all the updates are applied, or none are [Gray93].

Transactions are acclaimed widely as a powerful mechanism for handling failures. They simplify the design of reliable applications by restricting the variety of states in which a crash can leave a system. Systems that provide transactions for application use come in many forms. One such form is *recoverable memory.*

Recoverable memory provides atomic updates and persistence for a region of virtual memory [Satyanarayanan93]. It allows programs to manipulate permanent data structures safely in their native, in-memory form, without having to convert between persistent and non-persistent formats [Atkinson83].

Unfortunately, while transactions are useful in both kernel and application programming, their high overhead prevents them from being used ubiquitously to manipulate all persistent data. Committing a transaction has traditionally required at least one synchronous disk I/O. This several-millisecond overhead has persuaded most systems designers to give up the precise failure semantics of transactions in favor of faster but less-precise behavior. For example, many file systems can lose up to 30 seconds of recently committed data after a crash and depend on ad-hoc mechanisms such as ordered writes and consistency checks to maintain file system integrity.

The main contribution of this chapter is to present a system that reduces transaction overhead by a factor of 2000. We believe the resulting 5 µsec overhead makes transactions cheap enough that applications and kernels can use them to manipulate all persistent data. Two components, Rio and Vista, combine to enable this speedup.

We described the Rio File Cache in Section 1.4. Combined with an uninterruptible power supply, Rio provides persistent memory to applications that can be used as a safe, in-memory buffer for file system data. Existing recoverable-memory libraries such as RVM [Satyanarayanan93] can run unchanged on Rio and gain a 20-fold increase in performance.

Vista is a user-level, recoverable-memory library tailored to run on Rio [Lowell97]. Because Vista assumes its memory is persistent, its logging and recovery mechanisms are fast and simple. The code for Vista totals 720 lines of C, including its basic transaction routines, recovery code, and persistent heap management. Vista achieves a 100-fold speedup over existing recoverable-memory libraries, even when both run on Rio. We expect Vista performance to scale well with faster computers because its transactions use no disk I/Os, no system calls, and only one memory-to-memory copy—factors that have been identified repeatedly as bottlenecks to scaling [Ousterhout90, Anderson91, Rosenblum95].

3.2 Related Work

Out of the vast literature on transaction processing, Vista is related most closely to RVM and a number of persistent stores.

RVM is an important, widely referenced, user-level library that provides recoverable memory [Satyanarayanan93]. RVM provides a simple, lightweight layer that handles atomicity and persistence. To keep the library simple and lightweight, the designers of RVM did not support other transaction properties, such as serializability and nesting, arguing that these could be better provided as independent layers on top of RVM [Lampson83].

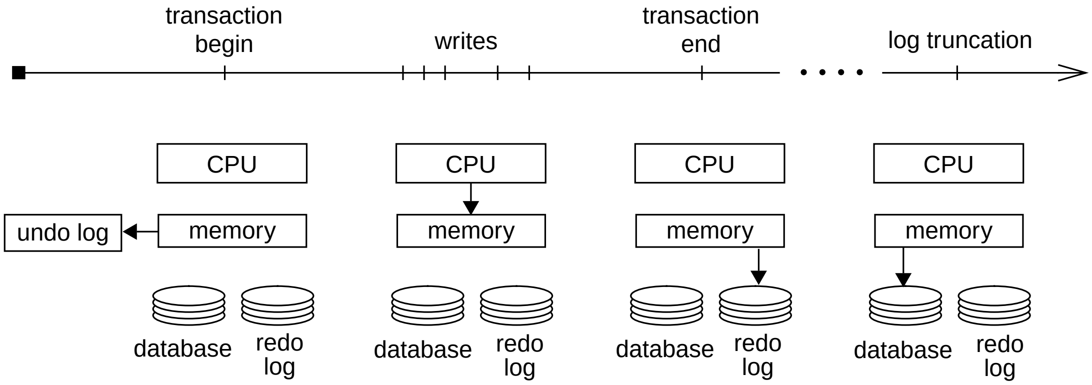


Figure 3.1: Operations in a Typical Recoverable Memory System

Recoverable memory systems perform up to three copies for each transaction. They first copy the before-image to an in-memory region called the undo log, which is used to support user-initiated aborts. When the transaction commits, they reclaim the space in the undo log and synchronously write the updated data to an on-disk region called the redo log. After many transactions commit and the redo log fills up, they truncate the log by propagating new data to the database on disk and reclaiming space in the redo log.

Figure 3.1 shows the steps involved in a simple RVM transaction. After declaring the beginning of the transaction, the program notifies RVM of the range of memory the transaction may update. RVM copies this range to an in-memory region called the undo log. The program then writes to memory using normal store instructions. If the transaction commits, RVM reclaims the space in the undo log and synchronously writes the updated data to an on-disk region called the redo log. If the user aborts the transaction, the undo log is used to quickly reverse the changes to the in-memory copy of the database. After many transactions commit and the redo log fills up, RVM truncates the log by propagating new data to the database on disk and reclaiming space in the redo log. After a process or system crash, RVM recovers by replaying committed transactions from the redo log to the database.

In addition to providing atomicity and durability, writing data to the redo log accelerates commit, because writing sequentially to the redo log is faster than writing to scattered locations in the database. Writing to the redo log in this manner is known as a *no-force* policy, because dirty database pages need not be forced synchronously to disk for every transaction [Haerder83]. This policy is used by nearly all transaction systems.

Note from Figure 3.1 that RVM performs up to three copy operations for each transaction. RVM performs two copies during the transaction: one to an in-memory undo log and one to an on-disk redo log. Log truncation adds at least one additional copy for each modified datum in the log.

Persistent stores such as IBM 801 Storage [Chang88] and ObjectStore [Lamb91] provide a single-level storage interface to permanent data [Bensoussan72]. Like RVM, most persistent stores write data to a redo log to avoid forcing pages to the database for each transaction.

RVM and persistent stores build and maintain an application's address space differently. RVM *copies* the database into the application's address space using `read` and maintains it using `read` and `write`. In contrast, persistent stores *map* the database into the application's address space and depend on the VM system to trigger data movement. We discuss the implications of these two choices in Section 1.4.

Vista differs from current recoverable memories and persistent stores in that Vista is tailored for the reliable memory provided by Rio. Tailoring Vista for Rio allows Vista to eliminate the redo log and its accompanying two copy operations per transaction. Vista also eliminates expensive system calls such as `fsync` and `msync`, which other systems must use to flush data to disk.

Having placed our work in context, we next describe the two systems that combine to reduce transaction overhead.

3.3 The Rio File Cache

As explained in Section 1.4, Rio makes the memory of the Unix file cache persistent by protecting it from kernel bugs, and then writing the cache contents to disk after a crash. Rio lets applications map a portion of the file cache directly into the application's address space using the `mmap` system call. Once this mapping is established, the application essentially possesses a region of memory it can address to which all stores are immediately permanent. Furthermore, after the `mmap` call has completed the mapped region can be updated without the overhead of system calls. Vista relies heavily on this ability to map persistent memory into the application's address space.

3.4 The Vista Recoverable Memory

The Rio file cache automatically intercepts writes that would otherwise need to be propagated synchronously to disk. For example, when RVM runs on Rio, writes to the redo log and database are not forced further than the file cache. Short-circuiting synchronous writes in this manner

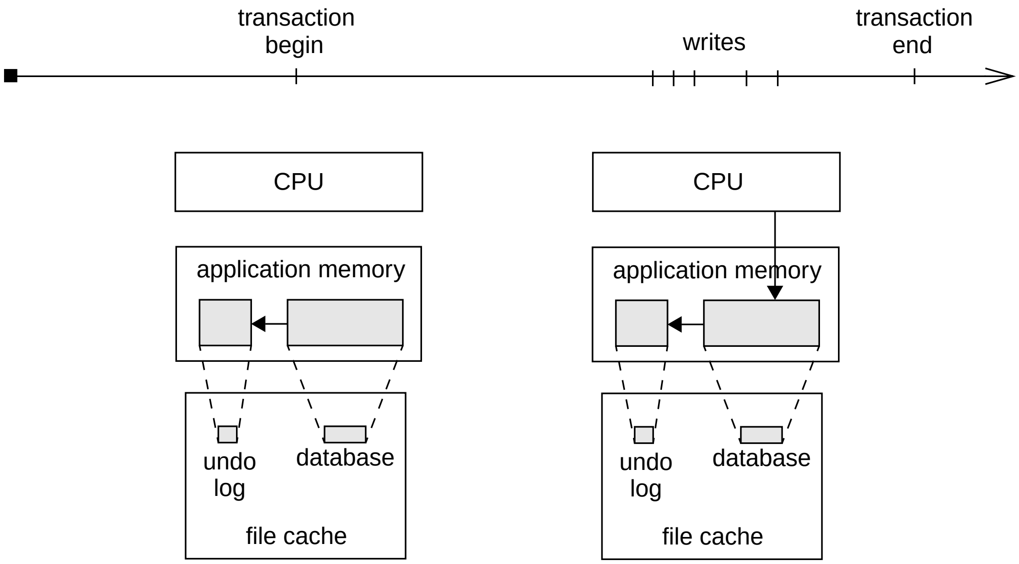


Figure 3.2: Operations in Vista

Vista’s operation is tailored for Rio. The database is mapped into the application’s address space then demand-paged into the file cache. At the start of a transaction, Vista logs the original image of the data to an undo log, which is kept in the persistent memory provided by Rio. During the transaction, the application makes changes directly to the persistent, in-memory database. At the end of the transaction, Vista simply discards the undo log.

accelerates RVM by a factor of 20. However, it is possible to improve performance by another factor of 100 by tailoring a recoverable-memory library for Rio.

Vista is just such a recoverable-memory library. While Rio accelerates existing recoverable memories by speeding up disk operations, Vista can eliminate some operations entirely. In particular, Vista eliminates the redo log, two of the three copies in Figure 3.1, and all system calls. As a result, Vista runs 100 times as fast as existing recoverable memories, even when both run on Rio. Because Vista is lightweight (5 µsec overhead for small transactions) and simple (720 lines of code), it is an ideal building block for higher layers. Like RVM, Vista provides only the basic transaction features of atomicity and persistence. Features such as concurrency control, nested transactions, and distribution can be built above Vista. Also like RVM, Vista is targeted for applications whose working set fits in main memory.

Figure 3.2 shows the basic operation of Vista. After calling `vista_init` and `vista_map` to map the database into the address space, applications start a transaction with vista_begin_transaction. As with RVM, applications call vista_set_range to declare which area of memory a transaction will modify. We insert these calls by hand, but they could also be inserted by a compiler [O'Toole93]. Vista also has an option to automatically generate the range of modifications on a page granularity by using the virtual memory system [Lamb91]. Generating modification ranges in this manner is slower than calling vista_set_range explicitly, but makes Vista easier for the programmer to use. Vista saves a before-image of the data to be modified in an undo log. Like the main database, the undo log resides in mapped persistent memory provided by Rio.

After invoking vista_set_range one or more times, the transaction modifies the database by storing directly to its mapped image. Because of Rio, each of these stores is persistent. The transaction commits by calling vista_end_transaction. This function simply discards the undo log for the transaction. The transaction may abort by calling vista_abort_transaction. This function copies the original data from the undo log back to the mapped database.

Rio and Vista cooperate to recover data after a system crash. During reboot, Rio writes the contents of the file cache back to disk, including data from any Vista segments that were mapped at the time of the crash. Each time a Vista segment is mapped, Vista inspects the segment and determines if it contains uncommitted transactions. Thus, the next time the application runs after a failure, Vista rolls back the changes from its uncommitted transactions in the same manner as user-initiated aborts. Recovery is idempotent, so if the system fails during recovery, Vista needs only to replay the undo log again.

Designing a transaction library with Rio's persistent memory in mind yields an extremely simple system. Vista totals only 720 lines of C code (not including comments), including the recovery code, begin/end/abort transaction, two versions of vista_set_range (explicit calls and VM-generated), and the persistent memory allocator described below. Several factors contribute to Vista's simplicity:

- All modifications to the mapped space provided by Rio are persistent. This enables Vista to eliminate the redo log and log truncation. Satyanarayanan, et al. report that the log truncation was the most difficult part of RVM to implement [Satyanarayanan93]. We conclude eliminating the redo log reduces Vista's complexity significantly.
- Recovery is simplified considerably without a redo log [Haerder83]. Recovery code is often an extremely complex component of transaction systems. In contrast, Vista lacks a redo log and its recovery code is fewer than 20 lines.

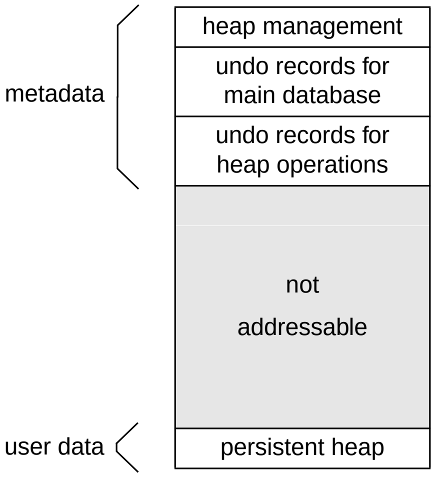

metadata
heap management
undo records for main database
undo records for heap operations
not addressable
user data
persistent heap

Figure 3.3: Memory Allocation in Vista

Metadata such as free lists and undo records are vulnerable to corruption because they are mapped into the application's address space. Vista reduces the risk of inadvertently corrupting this data by separating it from the user data.

- High-performance transaction-processing systems use optimizations such as group commit to amortize disk I/Os across multiple transactions. Vista issues no disk I/Os and hence does not need these optimizations and their accompanying complexity.
- Like RVM, Vista handles only the basic transaction features of atomicity and persistence. We believe features such as serializability are better handled by higher levels of software. We considered and rejected locking the entire database to serialize all transactions. While Vista's fast response times (5 µsec overhead for small transactions) makes this practical, adopting any concurrency control scheme would penalize the majority of applications, which are single-threaded and do not need locking.

Vista provides a general-purpose memory allocator that applications can use to dynamically allocate persistent memory. `vista_malloc` and `vista_free` can be invoked during a transaction and are automatically undone if the transaction aborts. Vista enables this feature by logically logging `vista_malloc` and `vista_free` calls. If a transaction commits, all memory that was freed in the transaction is returned to the heap. Similarly, if the transaction aborts, all memory that was allocated during the transaction is returned to the heap, and the memory that was freed during the transaction remains accessible. As a result, aborted transactions leave the heap unchanged. Vista uses the persistent heap internally to store undo records for transactions.

Persistent heaps provide functionality similar to traditional file systems but have some unique advantages and disadvantages. Persistent heaps may be more flexible than file systems, because programs can manipulate permanent data structures in their native form—programs need not convert between persistent and non-persistent formats [Atkinson83]. For example, programs can store native memory pointers in the persistent heap, as long as the system maps the heap in a fixed location.

With the increased flexibility of heaps comes increased danger, however. Whereas metadata of a file system is inaccessible to ordinary users, the metadata describing a persistent heap is mapped into the user's address space. Vista reduces the risk of inadvertent corruption by mapping each segment's metadata into an isolated range of addresses (Figure 3.3). This approach is similar to the technique used in anonymous RPC [Yarvin93]. Other approaches to protecting the metadata could also be used, such as software fault isolation and virtual memory protection [Wahbe93].

3.5 Performance Evaluation

Vista's main goal is to drastically lower the overhead of atomic, durable transactions. In order to evaluate how well Vista achieves this goal, we compare the performance of three systems: Vista, RVM, and RVM-Rio (RVM running on a Rio file system).

We use RVM (version 1.3) as an example of a standard recoverable memory. We maximize the performance of RVM by storing the database and log on two raw disks. Doing so saves the overhead of going through the file system and prevents the system from wasting file cache space on write-mostly data. The log size is fixed at 10% of the database size.

We also run RVM unmodified on a Rio file system (RVM-Rio) to show how Rio accelerates a standard recoverable memory. Storing RVM's log and data on a Rio file system accelerates RVM by allowing Rio to short-circuit RVM's synchronous writes. However, it causes the system to duplicate the database in both the file cache and application memory.

3.5.1 Benchmarks

We use three benchmarks to evaluate performance. We use a synthetic benchmark to quantify the overhead of transactions as a function of transaction size. We also use two benchmarks based on TPC-B and TPC-C, industry-standard benchmarks for measuring the performance of transaction-processing systems.

Each transaction in our synthetic benchmark modifies data at a random location in a 50 MB database. The size of the database was chosen to fit in main memory on all three systems. We vary the amount of data changed by each transaction from 8 bytes to 1 MB. We use the synthetic benchmark to calculate the overhead per transaction of each system. Overhead per transaction is defined as the time per transaction of a given system minus the time per transaction of a system that does not provide atomicity and durability. Time per transaction is the running time divided by the number of transactions.

TPC-B processes banking transactions [TPC90]. The database consists of a number of branches, tellers, and accounts. The accounts comprise over 99% of the database. Each transaction updates the balance in a random account and the balances in the corresponding branch and teller. Each transaction also appends a history record to an audit trail. Our variant of TPC-B, which we call **debit-credit**, follows TPC-B closely. We differ primarily by storing the audit trail in a 2 MB, circular buffer. We limit the size of the audit trail to 2 MB to keep it in memory and better match our target applications.

TPC-C models the activities of a wholesale supplier who receives orders, payments, and deliveries for items [TPC96]. The database consists of a number of warehouses, districts, customers, orders, and items. Our variant of the benchmark, which we call **order-entry**, uses three of the five transaction types specified in TPC-C: new-order, payment, and delivery. We do not implement the order-status and stock-level transactions, as these do not update any data and account for only 8% of the transactions. Order-entry issues transactions differently from TPC-C. In TPC-C, a number of concurrent users issue transactions at a given rate. In contrast, order-entry issues transactions serially as fast as possible. Order-entry also does no terminal I/O, as we would like to isolate the performance of the underlying transaction system.

Most transaction processing systems employ an optimization called *group commit.* Systems that perform this optimization wait for a number of concurrent committing transactions to accumulate, then synchronously write to disk all commit records for this group in one I/O [DeWitt84]. Doing so amortizes the cost of writing to the redo log over many transactions.

In order to see how Vista's performance fares against a system using this common optimization, we implement a simple form of group commit in RVM. We run each of our benchmark applications on RVM when 1, 8, or 64 transactions are grouped together to share a single write to the redo log.

For all graphs, we run each benchmark five times, discard the best and worst runs, and present the average of the remaining three runs. The standard deviation of these runs is generally

Seagate ST31200N	
spindle speed	5411 RPM
average seek	10 ms
transfer rate	3.3-5.9 MB/s

Seagate ST15150N	
spindle speed	7200 RPM
average seek	9 ms
transfer rate	5.9-9.0 MB/s

Seagate ST12550N	
spindle speed	7200 RPM
average seek	9 ms
transfer rate	4.3-7.1 MB/s

Table 3.1: Disk Parameters

less than 1% of the mean. We run each benchmark long enough to reach steady state; hence RVM and RVM-Rio results include truncation costs.

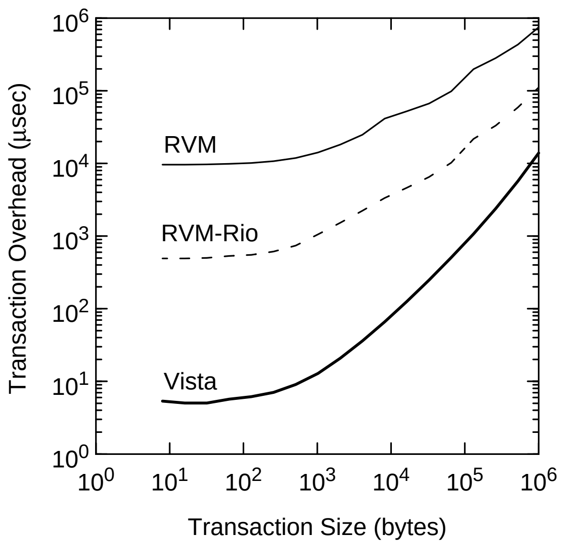
3.5.2 Environment

All experiments use a 175 MHz DEC 3000/600 Alpha workstation with 256 MB of memory. We use Digital Unix V3.0 modified to include the Rio file cache.

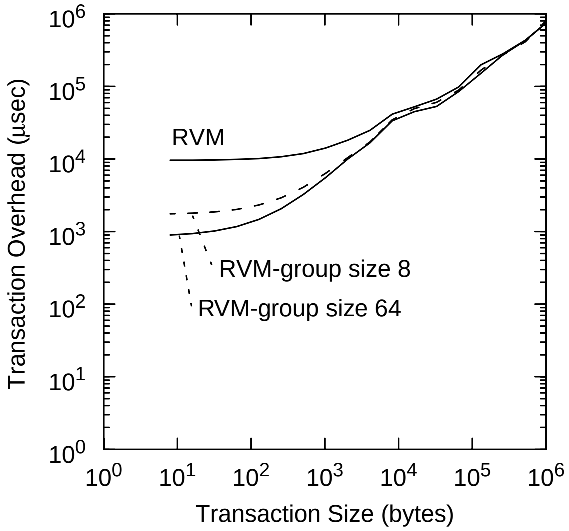
The workstation has three data disks (Table 3.1). RVM and RVM-Rio use ST31200N to page out the database when it overflows memory. This disk is attached via a separate SCSI bus from ST15150N and ST12550N. RVM and RVM-Rio use ST15150N to store the redo log and ST12550N to store the database. Vista uses ST15150N to store the database.

3.5.3 Results

The graphs in Figures 3.4, 3.5, and 3.6 compare the performance of Vista, RVM-Rio, and RVM on our three benchmarks, with and without group commit. The synthetic benchmark measures transaction overhead as a function of transaction size, so lower values are better. Debit-credit and order-entry measure transaction throughput, so higher values are better. Note that all y-axes use log scales.



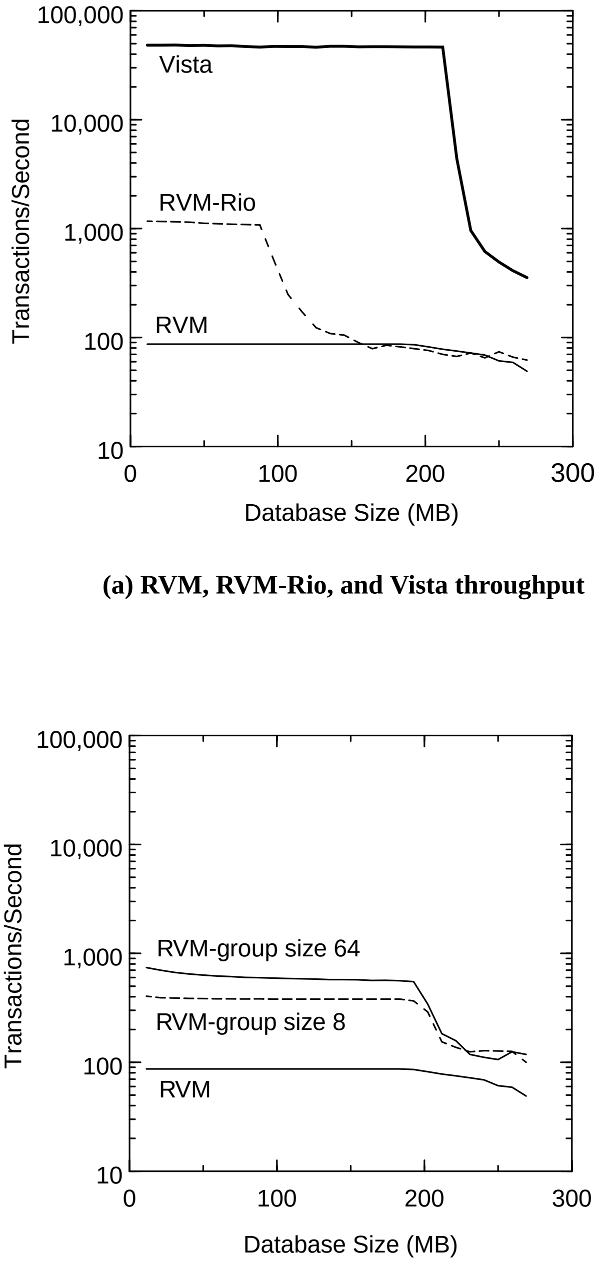
(a) RVM, RVM-Rio, and Vista overhead



(b) RVM overhead using group commit

Figure 3.4: Transaction overhead

This figure shows the overhead of RVM, RVM-Rio, and Vista using our synthetic benchmark. Figure 3.4a shows the overhead per transaction for a synthetic, 50 MB database. For small transactions, Vista reduces overhead by a factor of 100 over RVM-Rio and a factor of 2000 over RVM. In Figure 3.4b we vary the group commit size of RVM. Group commit reduces RVM's overhead by roughly a factor of 10 for small transactions.

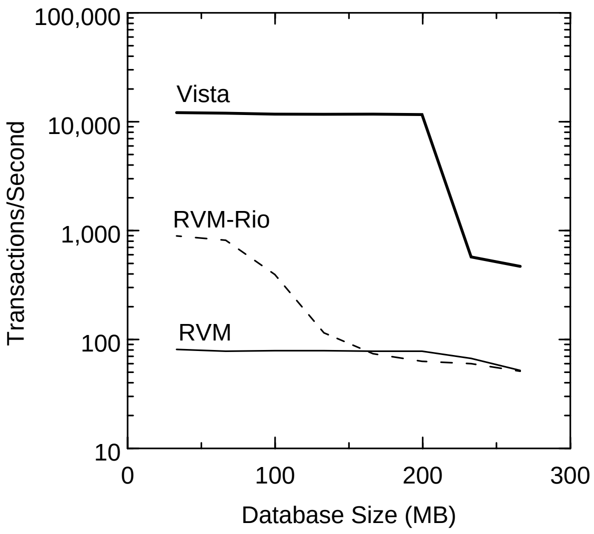


(a) RVM, RVM-Rio, and Vista throughput

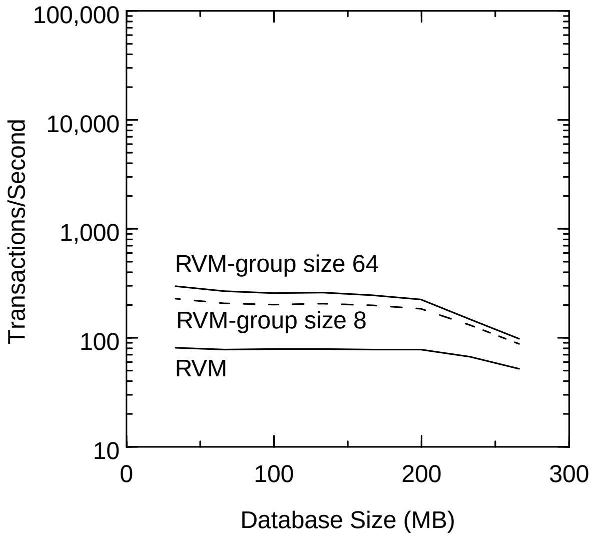
(b) RVM throughput using group commit

Figure 3.5: Small transaction throughput

This figure shows the throughput of three recoverable memories using the debit-credit benchmark, which generates small transactions. In Figure 3.5a we observe that Vista improves throughput by a factor of 41 over RVM-Rio and 556 over RVM. Figure 3.5b shows that group commit improves RVM's throughput by about a factor of 9 over the baseline version of RVM that commits one transaction at a time.



(a) RVM, RVM-Rio, and Vista throughput



(b) RVM throughput using group commit

Figure 3.6: Large transaction throughput

This figure shows the throughput of three recoverable memories using the order-entry benchmark, which generates larger transactions than debit-credit. In Figure 3.6a we observe that Vista improves throughput by a factor of 14 over RVM-Rio and 150 over RVM for this benchmark. Figure 3.6b shows that group commit improves RVM's throughput by a factor of about 4 over the baseline RVM that commits one transaction at a time.

Results for the synthetic benchmark show that Vista incurs only 5 µsec of overhead for small transactions. This overhead represents the cost of beginning the transaction, saving the old data with a single `vista_set_range`, and committing the transaction. For transactions larger than 1 KB, overhead scales roughly linearly with transaction size at a rate of 17.9 µsec/KB. The increasing overhead for larger transactions is due to the need to copy more data to the undo log. Each transaction in RVM-Rio incurs a minimum overhead of 500 µsec. Its higher overhead compared to Vista comes from its `write` and `fsync` system calls, copying to the redo log, and log truncation. Without Rio, RVM incurs one synchronous disk I/O to the redo log per transaction, as well as some portion of an I/O per transaction as a result of log truncation. As a result, it has an overhead of 10 ms for even the smallest transaction.

Debit-credit and order-entry show results similar to the synthetic benchmarks when the database fits in memory. For debit-credit, Vista improves performance by a factor of 41 over RVM-Rio and 556 over RVM. For order-entry, Vista improves performance by a factor of 14 over RVM-Rio and 150 over RVM. These speedups are smaller than the improvement shown in the synthetic workload because the body of the debit-credit and order-entry transactions have larger fixed overheads than synthetic, and because they issue several `vista_set_range` calls per transaction.

RVM and Vista both begin thrashing once the database is larger than available memory (roughly 200 MB). Note that RVM-Rio begins thrashing at half the database size of RVM and Vista due to *double buffering*. Double buffering results from frequent writes to the database file, effectively copying the database into the file cache. The result is two copies of the database: one in the process's address space and another in the Rio file cache.

While group commit does speed up RVM in our experiments, it is not a panacea. Group commit improves RVM's performance by a factor of 4 to 10 for small transactions, but for large transactions, group commit does not help. These large transactions already use the maximum IO bandwidth when writing only one commit record to disk at a time. Grouping multiple transaction commits together provides no additional efficiency.

Even for systems that execute small transactions, group commit may not be appropriate. First of all, it does not improve response time—waiting for a group of transactions to accumulate actually lengthens the response time of an individual transaction. Group commit also works only if there are many concurrent transactions. Single-threaded applications with dependent transactions cannot use group commit, since in these applications each transaction must commit before the next one can begin. There is no substitute for low latency transactions in these applications.

The performance improvements we have demonstrated can be broken into two components. As we expected, Rio improves performance for RVM by a factor of 11-20 by absorbing all synchronous disk writes. The biggest surprise from our measurements was how much Vista improved performance over RVM-Rio (a factor of 14-100). Once Rio removes disk writes, factors such as system calls, copies, and manipulating the redo log comprise a large fraction of the remaining overhead in a transaction. Vista does away with the redo log, all system calls, and all but one of these copies. The performance improvement resulting from the simplicity of Vista—Vista is less than 1/10 the size of RVM—is hard to quantify but is probably also significant.

Current architectural trends indicate that the performance advantage of Vista will continue to increase. RVM-Rio is slower than Vista because of the extra copies and system calls, while RVM is slower than Vista primarily because of synchronous disk I/Os. Many studies indicate that memory-to-memory copies, system calls, and disk I/Os will scale more slowly than clock speed [Ousterhout90, Anderson91, Rosenblum95].

3.6 Conclusions

The persistent memory provided by the Rio file cache is an ideal abstraction on which to build a recoverable memory. Because stores to Rio are automatically persistent, Vista can eliminate the standard redo log and its accompanying overhead, two out of the three copy operations present in standard recoverable memories, and all system calls. The resulting system achieves a performance improvement of three orders of magnitude and a reduction in code size of one order of magnitude.

Vista's minimum overhead of 5 μsec per transaction is small enough that it can be used even for fine-grained tasks such as atomically swapping two pointers—tasks for which I/Os are too expensive. Furthermore, Vista's fast transactions should prove very useful as we construct protocols that provide consistent recovery.

CHAPTER 4

DISTRIBUTED RECOVERY USING VISTAGRAMS

In this chapter we take a first stab at providing consistent recovery for distributed applications. We develop a system called *Vistagrams* that implements a recovery protocol in which messages are persistent and sent and received within local Vista transactions. Applications that use this protocol for communication and management of permanent data are guaranteed consistent recovery. With Vistagrams, we begin to fill in the empty portions of the protocol space developed in Section 2.3. Furthermore, Vista's fast commit lets us do so with a simple and efficient protocol, one that falls closer to the origin in the protocol space than existing protocols.

4.1 Introduction

As described in Chapter 2, writing a distributed application that can reliably manage persistent data is difficult. Such applications have to contend with the possibility of system crashes occurring at inopportune moments, possibly causing the computation to output a sequence of visible events it never could in failure-free execution. Furthermore, this problem worsens as distributed systems grow: the probability of some node in a distributed system experiencing a crash increases as more and more nodes are involved in the distributed computation.

In Section 2.3.1, we describe a variety of recovery protocols that have been proposed. In this chapter we introduce a new distributed recovery protocol called *Vistagrams*. Vistagrams is designed to enable systems designers to build their systems around the *Messages in Local Transactions* (MLT) model, which we also develop in this chapter. Under MLT, applications send and receive messages atomically with relevant local state changes.

The efficient implementation of systems based on MLT depends on the availability of fast stable storage, such as the reliable memory provided by the Rio file cache (Section 1.4). Such memory can provide the necessary durability for data and messages under our model, without the latency of disk writes. Vistagrams is an extension to our Vista transaction system (Chapter 3). Both Vista and Vistagrams make extensive use of Rio's persistent memory.

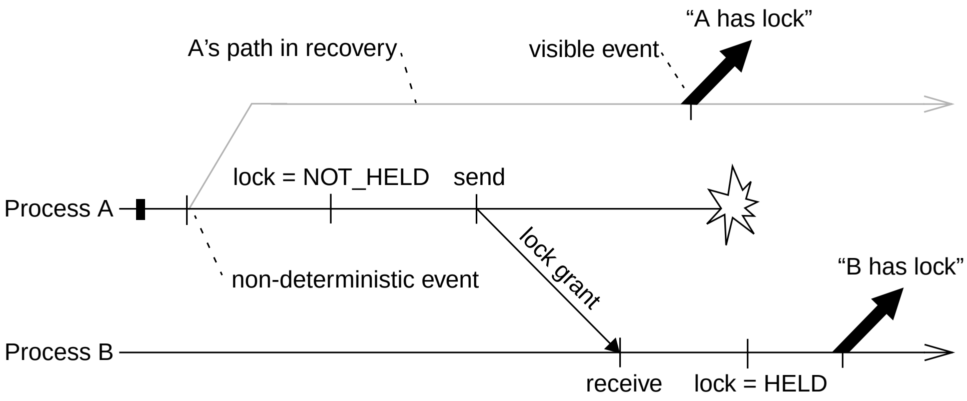


Figure 4.1: A distributed computation

This space-time diagram depicts a distributed computation in which two processes share a lock. At some point in the computation, process A releases the distributed lock and grants that lock to process B. After doing so, A fails and recovers to a state in which it still holds the lock. In this scenario, both processes think they hold the lock and could easily execute visible events that expose this inconsistency to the observer.

4.2 Background

In Chapter 2, we examined the circumstances of inconsistent recovery. Consider Figure 4.1 depicting a computation in which two processes share a distributed lock. In the portion of execution shown, process A begins holding the lock. It takes a checkpoint (the black box), then executes a non-deterministic event and passes the lock to process B. Process B then accepts the lock and executes a visible event to announce to the outside world that it is holding the lock. If process A should fail after granting the lock to process B, it would recover back to the state preserved in its checkpoint. It could then take a different path during recovery as a result of executing a sibling of its pre-failure non-deterministic event. Along this other path, it could well execute a visible event declaring that it holds the lock. Clearly, in no legal execution of the computation should both processes announce that they hold the lock, and recovery is inconsistent in this example. We will use this example to motivate our Messages in Local Transactions Model.

4.3 Messages in Local Transactions Model

A traditional atomic durable transaction groups a series of updates to persistent data into an indivisible unit. The transaction mechanism guarantees that either all the updates in a transac-

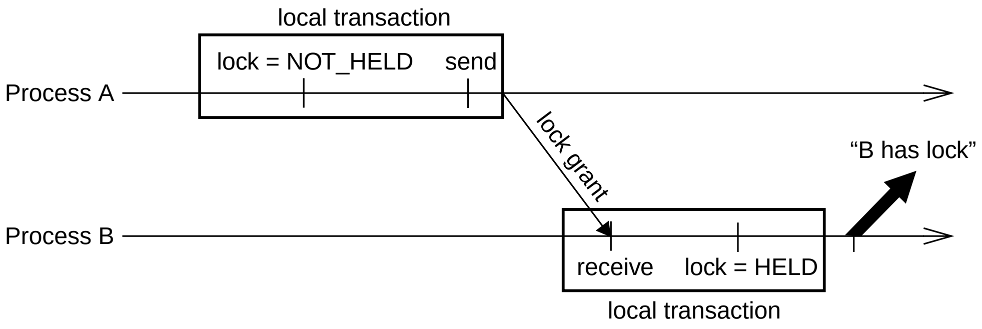


Figure 4.2: Distributed computation using MLT

The same distributed computation as shown in Figure 4.1, except that this system uses the MLT model. Persistent state changes are grouped in local transactions with relevant message sends and receives.

tion will take place, or none will, and maintains this invariant even if a crash occurs in the middle of processing the transaction. In the Messages in Local Transactions model, we add message sends and receives to the class of operations that can be performed atomically and durably within a local transaction.[1]

The Messages in Local Transactions model has two principal components [Lowell98b]. First, committed messages and data are not lost across system crashes. Second, sending and receiving messages are done atomically with the other operations in the transaction.

An application using this model would group a series of persistent updates, sends, and receives into an atomic unit using a local transaction.

Once the local transaction commits, the application receives several assurances. First, the updates performed in the transaction are permanent. The messages sent in the transaction are guaranteed to be sent and not lost, even in the presence of crashes by either the sender or receiver. Furthermore, it is impossible for the sender to "forget" that a message has been sent: since the send and relevant local state changes are done together atomically, the process' local persistent state will always accurately reflect whether the message was sent.

Second, if the transaction aborts, the messages "sent" within the transaction will not go out, and the state of the process' persistent data will be rolled back to its state at the beginning of

[1] Aspects of this model were proposed by Soparkar et al. as a construct for databases [Soparkar90]. They did not explore the model thoroughly because they lacked fast stable storage.

the transaction. Any receives of messages performed within the transaction will be undone so that those messages will be delivered again during a subsequent receive.

Making message sends atomic requires deferring message sends until commit. An alternative is to send the message before commit, then abort the receiver if the sender aborts. However, this alternative requires senders and receivers to coordinate during commit, a potentially expensive proposition. One immediate implication of deferring sends until commit is that an application cannot send a request and receive the response to that request in a single transaction. The application would instead have to commit the transaction in which the send of the request appears, and then begin another transaction in which to receive the response. Hence the application may need to recover to an increased number of points in the program.

4.3.1 Protocol independence

MLT can be implemented as a part of any reliable protocol, without adding extra copies or messages. To see how this is so, consider that reliable protocols must already buffer messages and retransmit them. To provide an MLT service, a reliable protocol must simply perform its message buffering on stable storage, and retransmit messages until received, even over crashes. In essence, providing MLT service merely constrains where a protocol buffers messages, and how long it retransmits.

MLT implementations must also defer message operations in order to commit them atomically with local state changes. Deferring message operations again does not constrain the specifics of the reliable protocol employed. It merely constrains when the protocol is initiated.

4.4 Design and implementation of Vistagrams

We have implemented a system based on the Messages in Local Transactions model. Our system, which we call Vistagrams, is an extension to our lightweight transaction library Vista.

As shown in Figure 4.3, Vistagrams provides a request/response-based interface. Clients send requests to servers and get back responses, and can do work in between. Each request by the client and corresponding response from the server are correlated by a globally unique request ID. Although we refer to processes as clients and servers, any process can be a client or server in our system. In a request/response cycle the process initiating the request is called the client and the process that receives the request and sends the response is called the server. In a subsequent cycle, these roles may be reversed.

Our message delivery protocol is optimized for request/response interactions. The server's response confirms that it received the client's request, and the acknowledgment for the server's

```
vista_begin_transaction
vista_end_transaction
vista_abort_transaction

vista_send_request
vista_receive_request
vista_send_response
vista_receive_response
vista_select
```

Figure 4.3: Some of the routines in the Vista and Vistagrams API

response is then piggybacked on a subsequent request. This protocol preserves the common case of a total of two messages used in a reliable request/response cycle between client and server.

At a high level, Vistagrams has several key tasks to attend to.

First, Vistagrams has to buffer outgoing messages persistently so it can retransmit them after packet loss or system crashes.

Second, Vistagrams has to defer many operations until commit, such as sending messages, freeing buffers, removing requests and responses from tables, and enqueueing acknowledgments to be sent to servers. These operations are deferred by adding them to various operation logs that are played back at commit.

Third, Vistagrams must commit outgoing message sends atomically with updates in the surrounding transaction. Committing atomically involves setting a single flag that marks the transaction committed, and then performing all the deferred operations. Those operations are carefully designed to be idempotent so that they can be redone if a crash occurs during log playback.

Finally, Vistagrams has to keep retransmitting committed messages until they are received, possibly after a crash by the sender or receiver.

Because Vistagrams messages are retransmitted over crashes, and because the information needed for the recipient to filter messages is persistent, Vistagrams provides efficient *exactly once* message delivery semantics. Exactly once semantics guarantees that a sent message will be delivered exactly one time by the receiving process. Providing such a semantic efficiently has been widely acknowledged to be difficult, since system crashes can cause processes to forget which messages have already been delivered [Mullender93].

4.4.1 Proof that Vistagrams guarantees consistent recovery

With Vista and Vistagrams, the programmer is responsible for allocating persistent data, updating it within transactions, and committing those transactions. As a result, we need to make several assumptions about how applications will handle their state and commits in order to prove that Vistagrams guarantees consistent recovery.

Assumption 2

> In each Vistagrams application, all state changes are done atomically with relevant message sends and receives. The state modified in those state changes is allocated persistently. Furthermore, each Vistagrams application commits its state before executing a visible event.

If applications behave as we assume, Vistagrams guarantees consistent recovery. This result follows from Theorem 2. Applications commit before doing a visible event, by our assumption. Vistagrams defers all message sends until after a commit event. Thus, there is a commit before every send event and visible event. As a result, there is a commit after every non-deterministic event that causally precedes every visible event executed, by the definition of causal precedence. Thus, by Theorem 2, consistent recovery is guaranteed. *QED*

Note that Vistagrams's protocol is closely related to the *commit prior to visible or send* (CPVS) protocol described in Section 2.2.5. The difference is that rather than committing before every send event as is done in CPVS, Vistagrams defers sends until the next naturally occurring commit.

4.4.2 Vistagrams implementation

To illustrate the structure of our Vistagrams implementation we will follow a request message from the client to the server and back.

Client sends request: When the client makes a call to `vista_send_request`, the outgoing message is assembled in a persistent buffer and added to the request table. The request table is the main client-side data structure. It is a hash table of vistagrams (hashed on request ID). It is used to correlate requests and responses, as well as to buffer messages for retransmission, or responses that are received out of order. Once the new request is in the request table, it is added to a log of messages to send when the enclosing transaction commits. At commit, the request is sent to the server using UDP.

Server receives request: The server waits for a new request in `vista_receive_request`. When it receives a request, it adds an entry to the response table. The response table is the central server-side data structure. Like the request table, it is a hash table hashed on request IDs. It is used to record the return address for a response, as well as to buffer outgoing response messages. After adding an entry to the response table, the incoming request message is returned to the user.

Server sends response: The server processes the request and then sends a response by calling `vista_send_response`. `vista_send_response` builds the outgoing message using the return address stored in the response table. It then persistently buffers the response in the appropriate response table entry (in case the response message is lost and needs to be retransmitted), and adds the response message to the log of messages to be sent at commit. When the server's transaction commits, the response message is sent from the server to the client.

Client receives response: The client calls `vista_receive_response` to receive the response message. The `vista_receive_response` function waits up to a fixed interval for the response to arrive. If the response does not arrive in that time, `vista_receive_response` sends the request to a special retransmit port on the server, and the routine waits again. Clients can retransmit indefinitely.

Once the response arrives, the response message is copied into a user-supplied buffer. An ack for the response is added to a special "ack queue" so that if the surrounding transaction commits, the response will be acknowledged during a subsequent request to the server. In addition, entries are made in appropriate operation logs so that the request table entry and its buffered request message are both deleted on commit.

Retransmission of messages: The server maintains retransmit ports that receive any retransmitted requests so that they can be handled specially. The server is notified asynchronously via a signal when a retransmitted message arrives. When a request arrives on one of those ports, the signal handler checks to see if the server already knows about the request by looking up the request ID in the response table. If the table contains an entry with a committed response message, that response is resent to the client (presumably the original response message was lost). If the entry in the response table does not contain a committed response, the retransmitted request is dropped. This situation could arise when the server has received the request, but not yet sent or committed the response. Finally, if the response table does not already contain an entry for the retransmitted request (which could occur if the original request message was lost), the request message is forwarded to the waiting server port on the same host.

4.4.3 Sample application

Figure 4.4 shows a very simple recoverable distributed application built with Vista and Vistagrams.

Client

```
          .
          .
          .
vista_begin_transaction();
   lock->status = PENDING;
   vista_send_request(lock_req);
vista_end_transaction();

vista_begin_transaction();
   lock->status = LOCKED;
   vista_receive_response(lock_grant);
vista_end_transaction
          .
          .
          .
```

Server

```
main()
{
   /*
    * Vista setup and allocation of
    * persistent data...
    */

   for (;;) {
      vista_begin_transaction();
         vista_receive_request(req);
         process_request_in_trans();
         vista_send_response(lock_msg
      vista_end_transaction();
   }

}
```

Figure 4.4: A sample recoverable lock manager using Vistagrams

The client is shown at some point in its execution requesting a lock from the server. Pseudocode is used in the parameter lists for brevity.

The client is shown at some point in its execution requesting a lock from the server. The client is forced to break up its lock request into two transactions because the sending of the lock request is deferred until commit. If the client should crash, it will have to check during recovery if the lock is in a “PENDING” state, and if so redo the second transaction. The server is completely recoverable as is, and needs to do no extra work to recover after a crash.

4.4.4 Comparison to existing systems

Vistagrams provides applications the same guarantees of recoverability and consistency as other protocols. However, Vistagrams lacks much of the complexity and overhead of other recovery systems.

First, since Vistagrams does not require the state of the system to be recomputed from past input, it works perfectly well with processes that execute non-deterministic events.

Second, when a Vistagrams process crashes, surviving processes are unaware of the fault (other than a delay while the crashed node recovers) and are unaffected. Surviving nodes are not involved in the recovery of failed nodes: they will never have to roll back to a prior state or be involved in any coordination of checkpoints to support recovery.

Third, all decisions by a process to commit transactions, send or deliver messages, or modify local state are strictly local under Vistagrams. No distributed commit is needed. Nor is there any need to track causal dependencies between processes to maintain consistent recovery.

Fourth, recoverable distributed systems can be built in which only a subset of the nodes use Vistagrams. It is sufficient for processes to pass messages using Vistagrams's reliable request-response protocol, but to handle their own recovery using any appropriate method. Each process must simply guarantee that all the messages it sends are committed.

Finally, systems built to use Vistagrams can tolerate the simultaneous failure of every node in the system and can safely fail while recovering.

An interesting feature of Vistagrams is its ability to group multiple message sends into atomic units, providing a relaxed variant of atomic multicast. In traditional atomic multicast, the sender is guaranteed that either all recipients will deliver a multicast message, or none will. When grouping multiple sends within a local transaction using Vistagrams, the sender receives a related guarantee: if the sender's transaction commits, all destination processes will eventually deliver the message. If the sender's transaction aborts, none will deliver.

To further highlight the relationship between Vistagrams and existing recovery protocols, Figure 4.5 depicts where Vistagrams might fall in the protocol space of Section 2.3.1.7.

4.5 Vistagrams performance

Vistagrams are designed to be very fast. Our Vistagrams implementation does not add any messages to the reliable request/response protocol used for message delivery. Nor does our Vistagrams implementation add any copies to the process of sending messages through the protocol. All the persistent buffering and logging that Vista and Vistagrams do to ensure atomicity and durability of messages and data is done using persistent VM provided by the Rio file cache. No disk I/Os are done during the course of performing a transaction or sending a message. We would expect that avoiding extra messages, copies, and disk I/Os would translate into a very fast system.

4.6 Microbenchmark

To evaluate our claims about Vistagrams performance, we use a simple distributed application that sends requests of varying sizes to a server and receives responses from that server. The server simply sends the request data it receives back to the client. We use two versions of this distributed application.

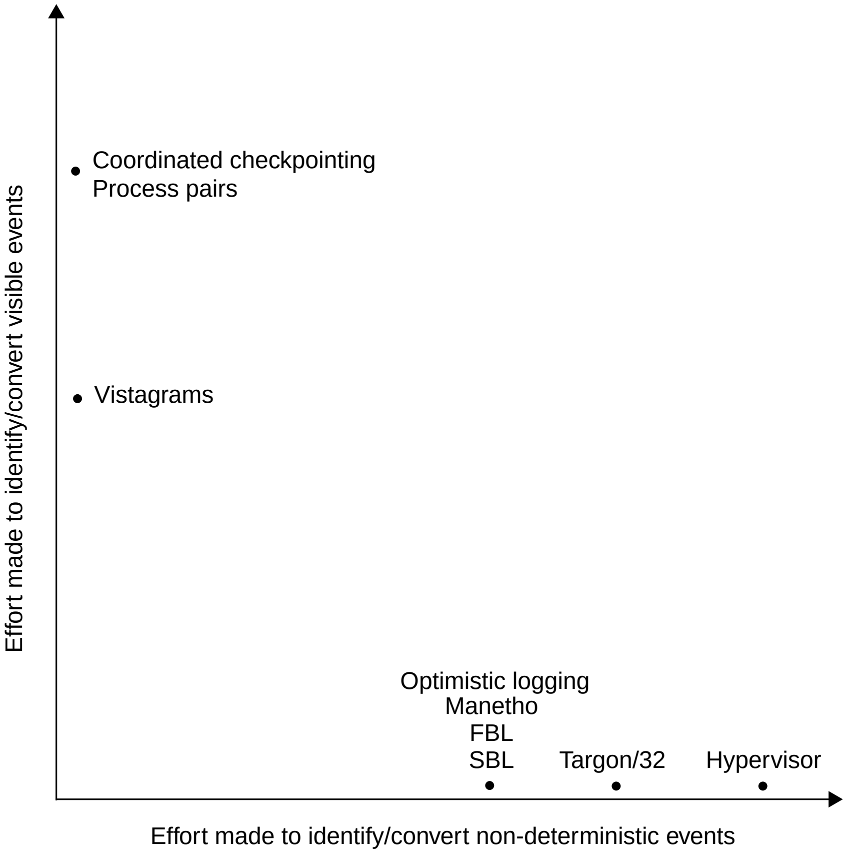


Figure 4.5: Vistagrams in the protocol space

In this plot, we show where the Vistagrams protocol might appear in the protocol space we developed in Section 2.3.1.7. Vistagrams is completely unconcerned with application non-determinism. Thus is appears with similar pure rollback protocols close to the Y axis. Vistagrams is not as far up the Y axis as coordinated checkpointing because Vistagrams treats message send events as visible rather than coordinate with the message recipient.

Our baseline system is a volatile and non-recoverable implementation of this system. Like Vistagrams, it retransmits messages to handle packet loss, and it correlates requests and responses. However, it does not use local transactions to send and receive messages atomically with local state, and messages and data are lost after crashes. We compare this baseline system with a fully persistent and recoverable implementation, built using Vista and Vistagrams.

Both versions of the application use identical reliable request/response protocols for message delivery. Hence by comparing the round-trip request/response times for each of these two systems, we get an accurate picture of the cost of making the sample application recoverable using Vistagrams and Vista transactions.

4.6.1 Environment

We run our benchmark on two Intel-based PCs, each with a 266 MHz Pentium II CPU and 128 Megabytes of memory. The systems are connected via a switched 100Base-T network, using an Intel Express 10/100 Fast Ethernet Switch. The server and client processes run on separate machines.

4.6.2 Results

Figure 4.6 shows the results of our benchmark runs. The plot shows the time for a request/ response cycle for each of our two test systems. For reference, we also show the round-trip time to send a UDP message of the given size from client to server and back.

As expected, Vistagrams adds almost no overhead to the baseline system (the Vistagrams and baseline curves are coincident). A Vistagrams request/response cycle takes 2-6 microseconds longer than a non-recoverable request/response cycle. This is a negligible fraction of the 240-2300 microsecond round-trip time. Vistagrams achieves high performance by avoiding extra copies and messages, and by using the reliable memory and fast transactions provided by Rio and Vista.

4.7 Using MLT and Vistagrams

The MLT model and Vistagrams have two principle drawbacks. First, applications have to be written using transactions. Transactions can be difficult to retrofit into existing applications. Second, as mentioned in Section 4.3, applications might have to commit transactions earlier than desired because sends are deferred until commit. These early commits can in some instances complicate recovery because it may force an application to end a transaction in the middle of a logically atomic series of steps. Hence the application must be able to recover to an increased number of points in the program.

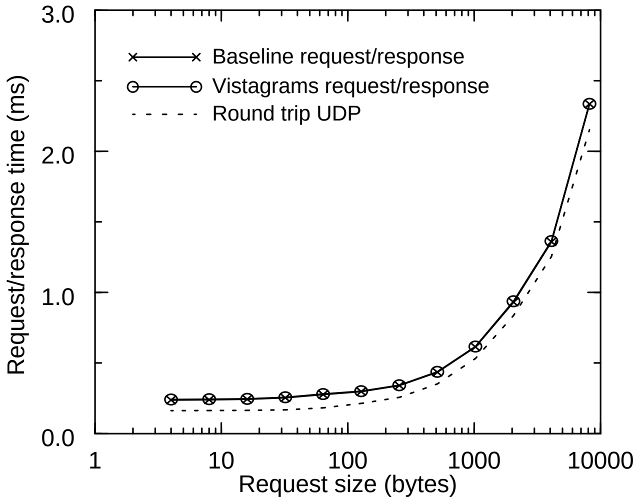


Figure 4.6: Vistagrams performance

This graph shows the time for a reliable request/response cycle between client and server with requests and responses of varying size. The baseline system is a non-recoverable system with volatile messages and data. The Vistagrams system is fully persistent and recoverable. Note that the Vistagrams and baseline curves are coincident. For reference, we show the round-trip time to send a UDP message of the given size from client to server and back.

In Chapter 5 we will describe a lightweight checkpointing system that addresses both of these issues.

4.8 Related Work

A significant amount of research has been done investigating the use of transactions in distributed applications. For example, *Distributed transactions* [Gray78] are also used to provide recovery for distributed systems. In distributed transactions, the decision to commit a transaction by one process involves coordinating the commit of related transactions in other processes. Several rounds of messages are needed to ensure that all the processes commit or abort together. Distributed commit protocols are similar to coordinated checkpointing, and like coordinated checkpointing, they can involve a significant amount of overhead at commit time.

Distributed transactions allows processes to send and receive uncommitted messages. If a process sends a message and later aborts, any causally dependent processes will also be aborted. In contrast, Vistagrams takes pains to ensure that all messages that are received by any process are committed. As a result, processes can receive messages knowing they will never be asked to abort on behalf of another process. In order to provide this guarantee, Vistagrams must buffer each message on stable storage and must constrain when processes commit. Therefore, Vistagrams is most useful when stable storage is very fast (e.g. Rio), and programmers do not mind having to commit more often than is strictly necessary. Distributed transactions lack these constraints, but at the cost of a more expensive commit.

4.9 Conclusion

Writing distributed applications that reliably manage persistent data involves navigating a host of thorny issues, such as ensuring consistent recovery. Furthermore, many of the traditional techniques for recovery of distributed applications have not been widely adopted in the field because of performance issues, or because of the limited scope of applications those techniques support [Birman96].

The advent of reliable memory and fast transactions have created new options for the recovery of distributed systems. Our Vistagrams system guarantees that applications will not lose persistent data or messages as a result of a system crash, surviving processes will never have to roll back, and systems will always recover to a consistent point in the computation. Furthermore, systems that use Vistagrams can be high performance, and can be non-deterministic.

However, for all Vistagrams’s advantages, it cannot be said to provide failure transparency. This shortcoming is a direct implication of the style of the commit employed in Vistagrams. Vista transactions are a programming tool, and programmers are therefore the parties responsible for inserting commits. In the next chapter, we will design a commit that can easily be inserted into an application without programmer assistance, getting us closer to the goal of providing failure transparency.

CHAPTER 5

DISCOUNT CHECKING: FAST, USER-LEVEL, FULL-PROCESS CHECKPOINTS

At this point in our quest for failure transparency, it should prove helpful to take stock of our progress. We have constructed a fast commit in the form of Vista's transactions. We have also used those transactions in a distributed recovery protocol—Vistagrams—that falls out of our theory of consistent recovery. However, as discussed in Chapter 4, Vista and Vistagrams are merely and aid to the programmer who must take responsibility for building recoverability into his or her applications. In order to provide failure transparency, we need to build a system that can recover applications transparently to the user, without involving the programmer. Towards that end, in this chapter we construct a style of commit which, unlike Vista transactions, is amenable to automatic insertion into existing applications.

5.1 Introduction

In Chapter 2, we firmly establish the importance of periodically committing an application's state when seeking to ensure the application can recover after a failure. However, there are a great many ways an application can commit. We have already constructed a useful, fast commit in the form of our Vista transaction library. However, our success with Vista is tempered by the acknowledgment that transaction-based commits are very difficult to retrofit into existing applications.

In this chapter, we instead shift our focus to a different style of commit. Checkpointing offers a general way to commit and recover a process. In applications that use checkpoints for recovery, the checkpointing subsystem periodically saves the complete state of a running process to stable storage. After a failure, the checkpointing system can reconstruct one of the saved states of the process and allow it to continue its execution from there. Since checkpointing systems commit the complete state of the process, there is no need for the programmer to oversee which state is made persistent.

Checkpointing systems strive to have very low overhead during failure-free operation. Furthermore, they endeavor not to increase significantly the disk or memory space required to run the program. With an ideal checkpointing system, users should be unaware of any slowdown during normal operation.

Unfortunately, current checkpointing systems have fairly high overhead during failure-free operation, typically many seconds per checkpoint. Since interactive applications often execute many non-deterministic and visible events per second, a commit that takes several seconds is too slow to be used to provide failure transparency for these applications.

In this chapter, we present a checkpointing system, *Discount Checking*, that is built on Rio's reliable main memory and Vista's high-speed transactions. Discount Checking's overhead per checkpoint for typical applications ranges from 50 μs to 2 milliseconds. This low overhead makes it possible to commit often enough to provide failure transparency, even for interactive applications.

5.2 Design and Implementation of Discount Checking

We next describe in detail the design of Discount Checking [Lowell98a]. The key to its fast checkpointing is Rio's reliable main memory (see Chapter 1) and Vista's fast transactions (see Chapter 3).

5.2.1 Transactions and Checkpointing

Although they are rarely discussed together in the literature, transactions and application checkpointing are very similar concepts. Figure 5.1 shows a process executing and taking checkpoints. The same process can be viewed as a series of transactions, where an interval between checkpoints is equivalent to the body of a transaction. Taking a checkpoint is equivalent to committing the current transaction and beginning the next. After a crash, the state of the process is rolled back to the last checkpoint, an operation equivalent to aborting the current transaction. The similarity between transactions and checkpointing leads naturally to the idea of using Vista's low-latency transactions to build a very fast checkpointing library.

5.2.2 Saving Process State

Building a checkpointing system on a transaction system is conceptually quite simple: map the process state into persistent memory and insert "transaction_begin" and "transaction_end" calls to make sure all updates to the process state are done within a the body of a transaction, and atomically committed. Discount Checking is a library that can be linked with the application to perform these functions.

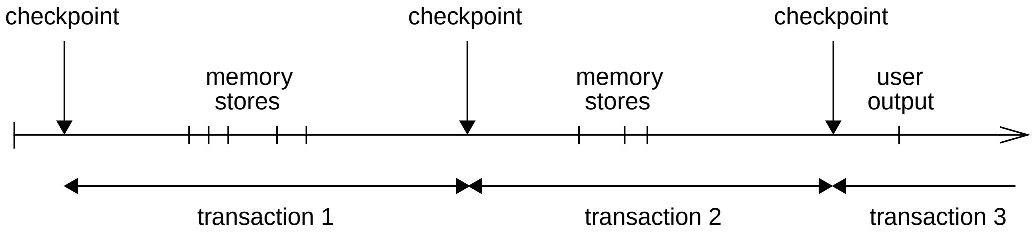


Figure 5.1: Equivalence of Checkpointing and Transactions

Transactions and checkpointing are very similar concepts. This diagram shows a process taking checkpoints as it executes. The interval between checkpoints is equivalent to the body of a transaction. Taking a checkpoint is equivalent to committing the current transaction and beginning the next. After a crash, the state of the process is rolled back to the last checkpoint, an operation that is equivalent to aborting the current transaction.

Discount Checking has three main types of process state to save in recoverable memory: the process's address space, registers, and kernel state.

The bulk of a process's state is stored in the process's address space. When the process starts, Discount Checking loads the process's data and stack segments into Vista's recoverable memory. It loads the data segment by creating a Vista segment, initializing it with the current contents of the data segment, and mapping it in place over the original data segment with `vista_map`. To minimize the number of Vista segments needed, Discount Checking moves the stack into a static 1 MB buffer in the data segment. A second Vista segment contains data that is dynamically allocated using malloc. The process then executes directly in the Vista segments—memory instructions directly manipulate persistent memory. In contrast, other checkpointing libraries execute the process in volatile memory and copy the process state to stable storage at each checkpoint. To restore the state of the process as of the last checkpoint after a process crash, Discount Checking must undo the memory modifications made during the current checkpoint interval. Vista logs this undo data in Rio using copy-on-write [Appel91] and restores the memory image during recovery.

A process's address space is easy to checkpoint because it can be mapped into Vista's recoverable memory. However, a process's state also includes register contents, which can not be mapped into memory. To preserve the register contents (stack pointer, program counter, general-purpose registers), Discount Checking simply copies them at each checkpoint using libc's `setjmp` function, and logs the old values into Vista's undo log using `vista_set_range`.

Some processes can be made recoverable by checkpointing only the address space and registers. However, making a general process recoverable requires saving the state stored in the kernel on behalf of the process. Ideally, the checkpointing system would have full access to the kernel so that it could preserve the relevant process state stored there. However, our strategy has been to see how much kernel state we could preserve with purely user-level operations. Discount Checking saves the pieces of kernel state that are most commonly required by general applications. We occasionally need to add the ability to recover new kernel state as we use Discount Checking for new applications.

Our basic strategy for saving kernel state is to intercept system calls that modify kernel state, save the updated kernel state values in Vista's memory, and directly restore that kernel state during recovery by redoing the system calls as needed.

The following are some examples of the types of kernel state recovered by Discount Checking:

Open files/sockets and file positions: Discount Checking intercepts calls to `open`, `close`, `read`, `write`, and `lseek` to maintain a list of open files and their file positions. During recovery, Discount Checking re-opens these files and re-positions their file position pointers. Discount Checking also intercepts calls to `unlink` in order to implement the Unix semantic of delaying `unlink` until the file is closed.

File system operations: File system operations such as `write` update persistent file data. Discount Checking must undo these operations during recovery, just like Vista undoes operations to its recoverable memory. To undo this state, Discount Checking copies the before-image of the file data to a special undo log and plays it back during recovery. Discount Checking does not need to log any data when the application extends a file, because there is no before-image of that part of the file.

Bound sockets: Discount Checking intercepts calls to `bind`, saves the name of the binding, and re-binds to this name during recovery.

Connected sockets: Discount Checking intercepts calls to `connect`, remembers the destination address for a particular socket, and uses this address when sending messages on that socket.

TCP: Much of the state used to implement the TCP protocol is in the kernel. To access this state, we implemented a user-level TCP library built on UDP. Since our TCP library is part of the process, Discount Checking saves its state automatically. To support applications that use X Windows, we modified the X library and server to use our TCP library.

Signals: Discount Checking intercepts `sigaction`, saves the handler information, and reinstalls the handler during recovery. Discount Checking also saves the signal mask at a checkpoint and restores it during recovery.

Timer: Discount Checking saves the current timer interval and restores the interval during recovery.

Page protections: Some applications manipulate page protections to implement functions such as copy-on-write, distributed shared memory, and garbage collection [Appel91]. Vista also uses page protections to copy the before-image of modified pages to its undo log. Vista supports applications that manipulate page protections by intercepting `mprotect`, saving the application's page protections, and installing the logical-*and* of Vista's protection and the application's protection. When a protection signal occurs, Vista invokes the appropriate handler(s).

5.2.3 Minimizing Memory Copies

As discussed above, Vista logs the before-image of memory pages into Rio's reliable memory. Vista then uses this undo log during recovery to recover the memory image at the time of the last checkpoint. As we will see in Section 5.4.1, copying memory images to the undo log comprises the dominant overhead of checkpointing (copying a 4 KB page takes about 40 µs on our platform).

Discount Checking uses several techniques to minimize the number of pages that need to be copied to the undo log. One basic technique it employs is copy-on-write. Instead of copying the entire address space during a checkpoint, Vista uses copy-on-write to lazily copy only those pages that are modified during the checkpoint interval. Copy-on-write is implemented using the virtual memory's write protection. On some systems, such as FreeBSD, system calls fail when asked to store information in a protected page. Discount Checking intercepts these system calls and prefaults the page before making the system call. Discount Checking reduces the number of stack pages that need to be copied by not write-protecting the portion of the stack that is unused at the time of the checkpoint. Since the current top of the stack at the time of the checkpoint will be restored after a failure, no undo-images in this unused portion of the stack will ever be needed.

Discount Checking must take special steps to use copy-on-write on stack pages, because naively write protecting the stack renders the system incapable of handling write-protection signals (delivering the signal generates another write-protection signal). To resolve this conflict, we use BSD's `sigaltstack` system call to specify an alternate stack on which to handle signals. The signal-handling stack is never write protected. Instead, its active portion is logged eagerly during a checkpoint. This eager copy adds very little overhead, because the signal stack contains data only when the checkpoint occurs in a signal handler, and the signal stack is usually not very deep even when handling a signal.

In addition to the signal stack, Discount Checking stores most of its own global variables in a Vista segment that is *not* logged using copy-on-write. These variables include the register contents at the last checkpoint, signal mask, list of open files, and socket state. Discount Checking copies these variables to the undo log as needed, rather than using copy-on-write to copy the entire page containing a variable.

As a result of these optimizations, Discount Checking can copy very little data per checkpoint. The minimum checkpoint size is 4360 bytes: a 4 KB page for the current stack frame, plus 264 bytes for registers and some of Discount Checking's internal data.

5.2.4 Vista-Specific Issues

Discount Checking benefits substantially by building on Vista's recoverable memory. The standard Vista library provides much of the basic functionality needed in checkpointing. For example, Vista provides a call (`vista_map`) to create a segment and map it to a specified address. Vista provides the ability to use copy-on-write or explicit copies to copy data into the undo log, and Vista recovers the state of memory by playing back the undo log. Vista also supplies primitives (`vista_malloc`, `vista_free`) to allocate and deallocate data in a persistent heap; Discount Checking transforms calls to `malloc/free` into these primitives. Vista provides the ability to group together modifications to several segments into a single, atomic transaction by using a shared undo log.

For the most part, Discount Checking required no modifications to Vista. The sole exception relates to Vista's global variables. Because Vista is a library, it resides in the application's address space. Our first implementation of Discount Checking used Vista to recover the entire address space, including Vista's own global variables! Consequently, Vista's global variables (such as the list of Vista segments) would magically change to their pre-failure values when the undo log was played back during Vista's recovery procedure. Vista, understandably, was not designed to handle this. Instead, we want Vista to manually recover its own variables, as it does when not running with Discount Checking. To address this problem, Discount Checking moves Vista's global variables to a portion of the address space that is not recovered by Vista. This was done by moving Vista's global variables to the segment that is not logged via copy-on-write, and not copying the variables to the undo log. Vista does not modify these variables during recovery because there are no undo images for them.

5.3 Using Discount Checking

Checkpointing libraries provide a substrate for making general applications recoverable. It should be much simpler to handle recovery using a checkpointing library than by writing custom recovery code for each new application. Towards this goal, Discount Checking requires only two minor source modifications to most programs. First, the program must include `dc.h` in the module that contains `main`. Second, the program must call `dc_init` as the first executable line in `main`. `dc_init` loads the program into Vista's recoverable memory, moves the stack, and starts the first checkpoint interval. The `dc_init` function takes several parameters, the most important of which is the file name to use when storing checkpoint data. After making these two changes, the programmer simply links with `libdc.a` and runs the resulting executable. Discount Checking currently requires the executable to be linked statically to make it easy to locate the various areas of the process's address space. We could most likely do away with the two required source code modifications by having the linker specially link `libdc.a`.

Although we have described in detail the sequence of events involved in taking a checkpoint, we have not described how a checkpoint gets initiated. The programmer is free to manually insert checkpoints into his or her application code, invoking `dc_checkpoint` as needed to ensure the desired recoverability. However, the real strength of Discount Checking comes from letting it automatically take checkpoints during execution. In so doing, it can ensure the recoverability of applications that have no recovery code of their own.

Discount Checking traps application events such as certain system calls, writes to the screen, messages sent or received, and signals delivered, in order to decide when best to commit. Using this ability, Discount Checking can essentially implement one of many recovery protocols. For example, if Discount Checking is configured to trap message sends and writes to the screen, it can easily insert a checkpoint before those events, implementing a *commit prior to visible or send* (CPVS) protocol, which we know from Chapter 2 guarantees consistent recovery. We explore in Chapter 6 the various recovery protocols we can implement using Discount Checking's commit, and the performance issues of each for general applications.

Regardless of how checkpoints are initiated, recovery is the same: the user simply reinvokes the failed application. When the application invokes `dc_init`, Discount Checking notices that there is an existing checkpoint file and initiates recovery. The recovery procedure restores the registers, address space, and kernel state at the time of the last checkpoint, then resumes execution from that state. From the user's point of view, the program has simply paused. The program also is unaware that it has crashed, as it simply resumes execution from the last checkpoint.

We expect most programmers to rest happily while Discount Checking transparently checkpoints and recovers their program. However, some may want to extend the checkpointing library or play a more direct role during recovery. For these advanced uses, Discount Checking allows the programmer to specify a function to run during recovery (e.g. to perform application-specific checks on its data structures). Discount Checking also allows the programmer to specify a function to run before each checkpoint.

To summarize, Discount Checking lets a programmer write an application that modifies persistent data, without having to worry about a myriad of complex recovery issues. Once the application works, the programmer can make it recoverable easily by linking it with Discount Checking and making two minor source modifications.

5.4 Performance Evaluation

Our goal is to use checkpointing to recover general applications without unduly degrading failure-free performance. The feasibility of this goal hinges on the checkpointing speed of Discount Checking. In this section, we measure the overhead added by Discount Checking during failure-free operation under a variety of conditions.

We first use a microbenchmark to quantify the relationship between checkpoint overhead and working set size. Then, we explore the relationship between checkpoint interval and overhead for two synthetic workloads from the SPEC CINT95 suite. In Chapter 6, we do a detailed performance study of Discount Checking's performance when implementing a variety of recovery protocols and recovering several real applications.

Table 5.1 describes the computing platform for all our experiments. For all data points, we take five measurements, discard the high and low, and present the average of the middle three points. Standard deviations for all data points is less than 1% of the mean.

5.4.1 Microbenchmark

Copying memory pages to Vista's undo log comprises the dominant overhead of checkpointing. Figure 5.2 shows the relationship between the number of bytes touched per checkpoint interval and the overhead incurred per interval. The program used to generate this data is a simple loop, where each loop iteration touches the specified number of bytes, then takes a checkpoint.

Processor	Pentium II (400 MHz)
L1 Cache Size	16 KB instruction / 16 KB data
L2 Cache Size	512 KB
Motherboard	Acer AX6B (Intel 440 BX)
Memory	128 MB (100 MHz SDRAM)
Network Card	Intel EtherExpress Pro 10/100B
Network	100 Mb/s switched Ethernet (Intel Express 10/100 Fast Ethernet Switch)
Disk	IBM Ultrastar DCAS-34330W (ultra-wide SCSI)

Table 5.1: Experimental Platform

Discount Checking's minimum overhead is 50 µs per checkpoint. This overhead is achieved when touching a single 4 KB page per checkpoint (for example, as might be done by a program operating only on the stack). As expected, checkpoint overhead increases linearly with the number of bytes touched. Each 4 KB page takes 45 µs to handle the write protection signal and copy the before-image to Vista's in-memory undo log (32 µs for working sets that fit in the L2 cache). For example, a program that touches 1 MB of data during an interval will see 12 ms overhead per checkpoint.

In general, there is no fixed relationship between absolute overhead (seconds) and relative overhead (fraction of execution time). Real programs that touch a larger amount of data in an interval are likely to spend more time computing on that data than programs that touch only a small amount of data. Hence their checkpoint overhead will be amortized over a longer period of time. The main factor determining relative overhead is locality. Programs that perform more work per touched page will have lower relative overhead than programs that touch many pages without performing much work. We explore Discount Checking's overhead in real applications in Chapter 6.

5.4.2 Varying the Checkpoint Interval

We next measure how Discount Checking performs for different checkpoint intervals. The workloads we employ are non-interactive computations from the SPEC CINT95 suite. Prior checkpointing research has focused on these types of applications because they generate little output and hence require very few checkpoints for failure transparency. We use `ijpeg` and `m88ksim`; other benchmarks in the SPEC95 suite give similar results. The only modification needed to make

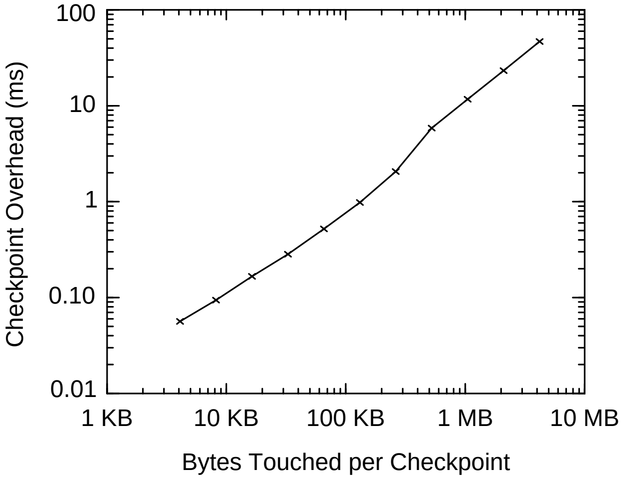


Figure 5.2: Overhead vs. checkpoint size.

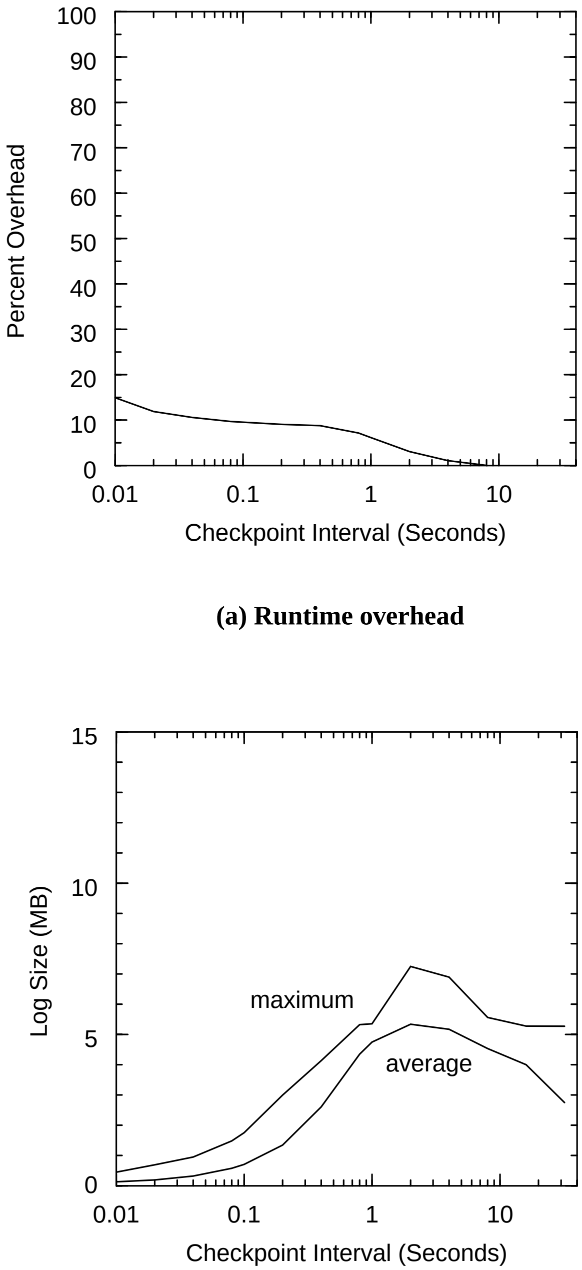
Checkpoint overhead is proportional to the number of pages touched per checkpoint. Each 4 KB page takes approximately 45 µs to handle the write-protection signal and copy the page to Vista's in-memory undo log (32 µs for working sets that fit in the L2 cache).

these applications recoverable with Discount Checking was inserting the call to `dc_init` (and the accompanying `#include`).

Figures 5.3a and 5.4a graph the relative overhead incurred as a function of checkpoint interval. We use periodic timer signals to trigger checkpoints at varying intervals. As expected, relative overhead drops as checkpoints are taken more frequently. At very short checkpoint intervals, overhead remains relatively constant because lengthening the interval increases the amount of bytes logged in the interval. In other words, the working set of these applications is proportional to interval length for short intervals, then increases more slowly for longer intervals.

The interval used in prior checkpointing studies has ranged from 2-30 minutes for these types of applications. Discount Checking is able to take checkpoints every second while adding only 6-10% overhead. Such a high frequency of checkpointing is not needed for these applications, but it serves to demonstrate the speed of Discount Checking. Other SPEC benchmarks gave similar results, with overhead ranging from 0-10% with a 1 second checkpoint interval.

Figures 5.3b and 5.4b show how the average and maximum sizes of the undo log vary as a function of checkpoint interval. As checkpoints become less frequent, the size of the undo log generally increases because more data is being logged during longer intervals. `ijpeg` shows a deviation from this general trend, where the undo log size eventually shrinks for longer checkpoint

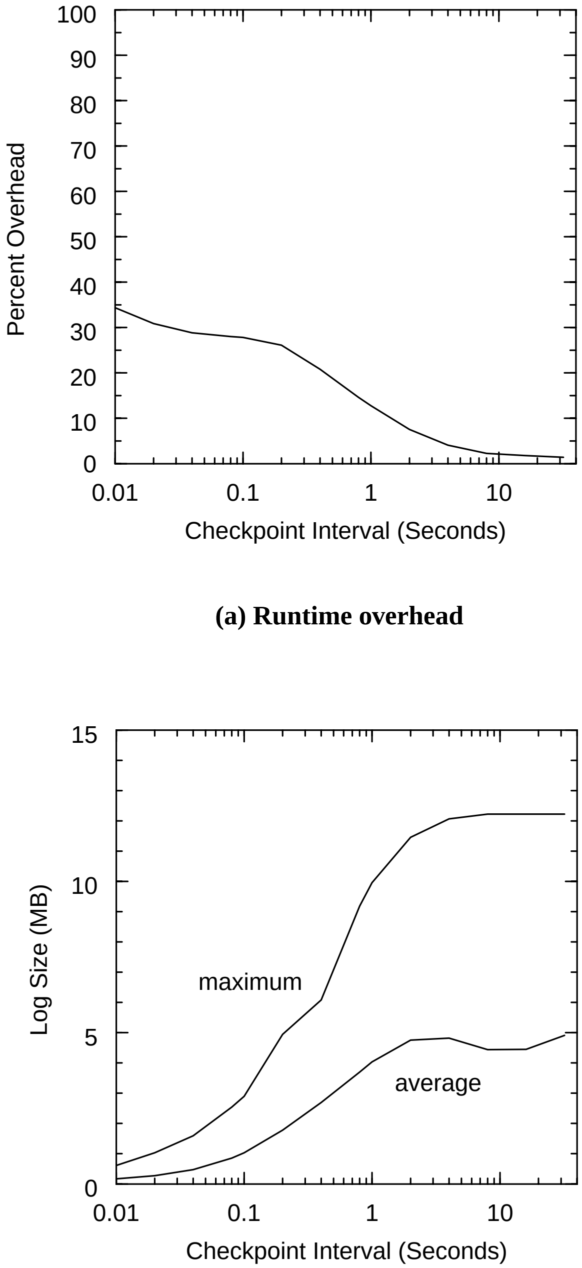


(a) Runtime overhead

(b) Log Size

Figure 5.3: Checkpoint Overhead for Varying Intervals: ijpeg

We modify the checkpoint interval in `ijpeg` to evaluate how the interval affects checkpoint overhead. As checkpoints are taken less frequently, the relative overhead drops and the size of the undo log generally increases. Checkpointing adds little overhead even at high rates of checkpointing. For `ijpeg`, the log size drops for large checkpoint intervals. As a result of how Vista handles mallocs and frees, large checkpoint intervals can result in more memory reuse and less data in the undo log.



(a) Runtime overhead

(b) Log Size

Figure 5.4: Checkpoint Overhead for Varying Intervals: m88ksim

We evaluate checkpoint overhead in `m88ksim`, which touches more data than `ijpeg`. Its larger memory footprint results in higher, though not unacceptable runtime overhead. `m88ksim` also uses significantly more space in the undo log.

intervals. This anomaly results from how Vista handles memory allocation within a transaction [Lowell97]. Vista defers `free` operations until the end of the transaction unless the corresponding `malloc` was performed in the current transaction. Hence, longer intervals allow Vista to re-use memory regions for new allocations, without adding more data to the log.

5.5 Related Work

Checkpointing has been used for many years [Chandy72, Koo87] and in many systems [Li90, Plank95, Tannenbaum95, Wang95]. The primary limitation of current checkpointing systems is the overhead they impose per checkpoint. For example, [Plank95] measures the overhead of a basic checkpointing system to be 20-159 seconds per checkpoint on a variety of scientific applications. To amortize this high overhead, today's systems take checkpoints infrequently. For example, the default interval between checkpoints for `libckpt` is 30 minutes [Wang95]. The high overhead per checkpoint and long interval between checkpoints limit the use of checkpointing to long-running programs with a minimal need for failure transparency, such as scientific computations. In contrast, we would like to make checkpointing a general tool for recovering general-purpose applications. In particular, interactive applications require frequent checkpoints to mask failures from users.

Researchers have developed many optimizations to lower the overhead of checkpointing. *Incremental checkpointing* only saves data that was modified in the last checkpoint interval, using the page protection hardware to identify the modified data. Incremental checkpointing often, but not always, improves performance. For example, [Plank95] measures the overhead of incremental checkpointing to be 4-53 seconds per checkpoint.

Asynchronous checkpointing (sometimes called forked checkpointing) writes the checkpoint to stable storage while simultaneously continuing to execute the program [Li94]. In contrast, synchronous checkpointing (sometimes called sequential checkpointing) waits until the write to stable storage is complete before continuing executing the process. Asynchronous checkpointing can lower total overhead by allowing the process to execute in parallel with the act of taking the checkpoint. However, asynchronous checkpointing sacrifices failure transparency to gain this performance improvement. To achieve failure transparency, a checkpoint must *complete* before doing work that is visible externally. In asynchronous checkpointing, the checkpoint does not complete until many seconds after it is initiated. Visible work performed after this checkpoint will be lost if the system crashes before the checkpoint is complete. This loss may be acceptable for programs that do not communicate frequently with external entities, but it hinders the use of checkpointing for general applications.

Memory exclusion is another technique used to lower the overhead of checkpointing [Plank95]. In this technique, the programmer explicitly specifies ranges of data that do *not* need to be saved. Memory exclusion can reduce overhead dramatically for applications that touch a large amount of data that is not needed in recovery or is soon deallocated. However, memory exclusion adds a significant burden to the programmer using the checkpoint library, and thus sacrifices failure transparency.

5.6 Conclusions

In this chapter, we present our checkpointing system that successfully provides a commit that is fast enough and general enough to be of use in our quest for failure transparency. Discount Checking's checkpoints take between 50 µs and 2 milliseconds for typical applications. We show that synthetic applications that touch significant amounts of memory per checkpoint can still take a checkpoint once per second with only 6-10% overhead. Our final task will be to employ Discount Checking's fast checkpoints to provide failure transparency for real applications.

CHAPTER 6

FAILURE TRANSPARENCY FOR GENERAL APPLICATIONS USING FAST CHECKPOINTS

At last the pieces are in place in our quest for failure transparency. We have a theory that informs how applications need to commit to assure that we can mask failures from users. We also have a fast commit that we can insert into applications without programmer assistance. All that remains is to use these resources to provide failure transparency for real applications. Towards this end, in this chapter we explore how best to use fast checkpoints to provide failure transparency for several real, interactive applications. In the end, we are successful in providing failure transparency for our challenging target applications, and surprisingly, we are able to do so with a user-level recovery system and with very low overhead.

6.1 Introduction

Operating systems that endeavor to provide the abstraction of failure transparency have several factors to worry about. First of all, these operating systems need a roadmap for recovery that details exactly what must be done in order to recover from a failure. They also need a fast commit that will work for general applications. Finally, they need recovery protocols using that commit that do not perceptibly degrade application performance.

In our quest for failure transparency, we have so far assembled the road map and the fast commit. In this chapter, we examine the last remaining issue: how best can we put these pieces together to provide failure transparency. We seek to answer several questions. What recovery protocol works best for our target applications? How does failure transparency affect failure-free performance? Can we provide failure transparency using disk instead of reliable memory for stable storage? Can a user-level checkpointing system commit sufficient kernel state to allow the recovery of general applications? In answering these questions, we hope to conclude our inquiry into failure transparency with an affirmation that failure transparency is feasible for the difficult applications we target.

6.2 Providing Failure Transparency

In order to transparently recover applications after failures, we need to insert commits into our sample applications in a manner that upholds the commit invariant of Section 2.2.4. We are interested in determining how best to uphold the commit invariant for our target applications using the fast checkpoints we develop in Chapter 5.

6.2.1 Discount Checking

In Chapter 5, we describe our user-level checkpointing system called Discount Checking that can commit and recover process state as needed for consistent recovery. Discount Checking is built on the reliable main memory provided by the Rio File Cache (see Section 1.4) and the fast transactions provided by Vista (see Chapter 3). There are many ways to provide the consistent recovery failure transparency requires. We are interested in determining how best to use Discount Checking's fast checkpoints to do so.

We are also interested in ascertaining whether failure transparency is possible in systems that lack reliable memory. We run all our experiments with both Discount Checking and a variant called Discount Checking-disk that does not assume the presence of the Rio File Cache. Discount Checking-disk uses the same code base as Discount Checking but is modified to write out a redo log to disk at each commit. This redo data includes the newest version of any application data changed during the last interval and any additional logs used by the recovery protocol. We use Discount Checking-disk to measure approximately how our recovery protocols perform without reliable main memory. Unlike Discount Checking which successfully recovers all our sample applications, Discount Checking-disk does not yet have the code needed to recover applications after failures.

As currently written, Discount Checking treats data in the file system as extensions of process state. Like all other process state, it undoes changes to file data after a crash. However, since the file system can serve as a large shared memory that allows processes to coordinate through reads and writes of the file system, it may make more sense to treat writes to the file system as a type of send event. Discount Checking could be trivially reconfigured to handle writes to the file system as send events, and reads from the file system as receives. Correct recovery of programs that use other forms of shared memory (such as System V IPC or `mmap`) would require Discount Checking to treat reads and writes of shared memory regions as receives and sends as well. Configuring Discount Checking to do so is less trivial however, since every load or store instruction destined for a shared region of memory is a potential receive or send event and would have to be trapped.

6.2.2 Recovery Protocols

There are many ways to uphold the commit invariant, as described in Section 2.2.5. We design seven different recovery protocols to illustrate the breadth of protocols to which our theorem gives rise, and to study which workloads perform best with which protocols. Each protocol guarantees consistent recovery, but does so using a different number of commits and with varying complexity and overhead. We implement each protocol as an option that can be set when compiling Discount Checking. The protocols are:

Commit prior to visible event or send (CPVS): Discount Checking forces each process to take a checkpoint immediately before executing a visible event or send by trapping every call to `send`, `sendto`, `sendmsg`, and `write`. As described in Section 2.2.5, if each process checkpoints prior to every message send event, it is sure to have committed after any non-deterministic events upon which the send event will causally depend. As a result, the recipient knows that after receiving a message, it will never have to coordinate with the sender to force it to checkpoint if the recipient needs to execute a visible event. Thus, if each process also checkpoints immediately before every visible event it executes, consistent recovery is guaranteed. This protocol has several advantages. First of all, it is very simple. We can implement it without having to track down all non-deterministic events executed by the application we intend to make recoverable. Identifying all sources of non-determinism can be daunting for some applications. Furthermore, all this protocol's checkpoints are strictly local; no coordination is needed between committing processes. This protocol should do well for applications that execute many more non-deterministic events than visible or send events. For applications with more send or visible events than non-deterministic events, this protocol can lead to superfluous checkpoints.

Commit after non-deterministic event (CAND): As described in Section 2.2.5, each process commits immediately after executing a non-deterministic event. Examples of non-deterministic events are receiving a message (non-deterministic because of message order), reading input from the user, taking a signal, calling `gettimeofday`, calling `select` to see if a port is ready to be read/written, and calling `bind` to request a system-selected port number. Discount Checking implements this protocol by trapping calls to `recv`, `recvfrom`, `recvmsg`, `read`, `gettimeofday`, `select`, `bind`, and the application's signal handlers, and forcing the process to checkpoint after executing the non-deterministic event. Clearly, by checkpointing immediately after all non-deterministic events, applications are trivially guaranteed a checkpoint after every non-deterministic event that causally precedes a visible event. Thus, they are guaranteed consistent recovery by Theorem 2. Like CPVS, this protocol has the advantage of being very simple. It should be ideal for applications that execute many more send events and visible events than non-deterministic events.

Commit between non-deterministic event and visible event or send (CBNDVS): Under this protocol each process commits between executing a non-deterministic event and executing a visible event or sending a message, as described in Section 2.2.5. To implement this protocol, Discount Checking traps the same non-deterministic system calls as in CAND. When one of these calls is made, it sets a flag noting that a non-deterministic event has been executed. Then, if a process attempts to execute a visible or send event when the flag is set (using one of `send`, `sendto`, `sendmsg`, or `write)`, Discount Checking forces a checkpoint before allowing the event to execute. Discount Checking clears the flag after taking a checkpoint. This protocol clearly guarantees a checkpoint between every non-deterministic event and causally dependent visible event, and as a result, it guarantees consistent recovery. Under this protocol, applications will always execute with the same or fewer checkpoints than with CPVS or CAND. In fact, CBNDVS represents the lower bound on checkpoint frequency possible without actively converting non-determinism or taking the steps needed to treat send events as non-visible. CBNDVS should be well suited to applications that execute either large numbers of non-deterministic events, or large numbers of sends or visible events, but not large numbers of both.

Commit after non-deterministic event, with logging (CAND-LOG): This protocol is identical to CAND, with logging added to convert some non-deterministic events into deterministic ones in an attempt to reduce commit frequency. Discount Checking implements this protocol by trapping the same non-deterministic events as in CAND (calls to `recv`, `recvfrom`, `recvmsg`, `read`, `gettimeofday`, `select`, `bind`, and the application's signal handlers), and writing the results of the `recv`, `recvfrom`, `recvmsg`, `read`, and `select` calls to a log in order to make those events deterministic. After the remaining non-deterministic events, this protocol simply forces a checkpoint. It is possible to log a larger class of non-deterministic events, such as signals [Slye96], however doing so can be quite difficult and is beyond the scope our inquiry. This protocol guarantees consistent recovery following the same reasoning we followed for CAND. Like CAND, CAND-LOG should be ideal for applications that execute many more send and visible events than non-deterministic events. However, CAND-LOG is more tolerant than CAND of applications that send lots of messages, since it is able to make the receives of those messages deterministic. For some workloads, it is conceivable that logging overhead will outweigh the benefit it confers.

Commit between non-deterministic event and visible event or send, with logging (CBNDVS-LOG): This protocol is identical to CBNDVS, with logging used to convert some non-deterministic events into deterministic events. Like in CBNDVS, Discount Checking sets a flag whenever a process executes a non-deterministic event (i.e. a process calls `gettimeofday`, `bind`, or receives a signal). However, events like `recv`, `recvfrom`, `recvmsg`, `read`, and `select` that would ordinarily be non-deterministic are logged, and thus rendered deterministic. As a result, executing these events does not cause the flag to be set. Like CBNDVS, if a process later executes a visible event or send event (using one of `send`, `sendto`, `sendmsg`, or `write`) while this flag is set, Discount Checking forces a checkpoint. In principle, the logging CBNDVS-LOG employs should reduce commit frequency to well below that of CBNDVS. However, it is possible that for some applications, the overhead of logging is greater than the commit overhead it reduces. CBNDVS-LOG should work well for applications that execute either large numbers of signals and calls to `gettimeofday` and `bind`, or large numbers of visible events or sends, but not large numbers of both.

Commit prior to visible event with two-phase commit (CPV-2PC): Under this protocol, any time some process wants to execute a visible event, it asks all the processes in the computation to commit their state using a *two-phase commit* protocol (2PC) [Gray78]. Discount Checking traps calls to `write` whose output is destined for the screen, and initiates the distributed checkpoint using 2PC. CPV-2PC clearly upholds the commit invariant: before any process does a visible event, each process takes a checkpoint, which must be after all non-deterministic events that causally precede this visible. CPV-2PC should be suited to applications that may execute large numbers of non-deterministic and send events, but comparatively few visible events. However, because all processes checkpoint whenever any process executes a visible event, this protocol might actually add commits to the computation for some distributed workloads.

Commit between non-deterministic event and visible event (CBNDV-2PC): This protocol improves on CPV-2PC. Under this protocol, Discount Checking only initiates a distributed checkpoint if the process executing the visible event has executed a non-deterministic event since its last commit. As in CBNDVS, Discount Checking traps non-deterministic events in order to set a flag. If some process executes a `write` to the screen while the flag is set, Discount Checking asks all processes to commit using 2PC. This protocol should handle applications that execute frequent visible events better than CPV-2PC, as only those visible events that follow an uncommitted non-deterministic event lead to the full distributed checkpoint.

The two protocols that employ two-phase commit (CPV-2PC and CBNDV-2PC) use a separate 2PC server process to coordinate their distributed checkpoints. A process wishing to initiate a distributed checkpoint contacts the 2PC server who then handles the coordination between the dif-

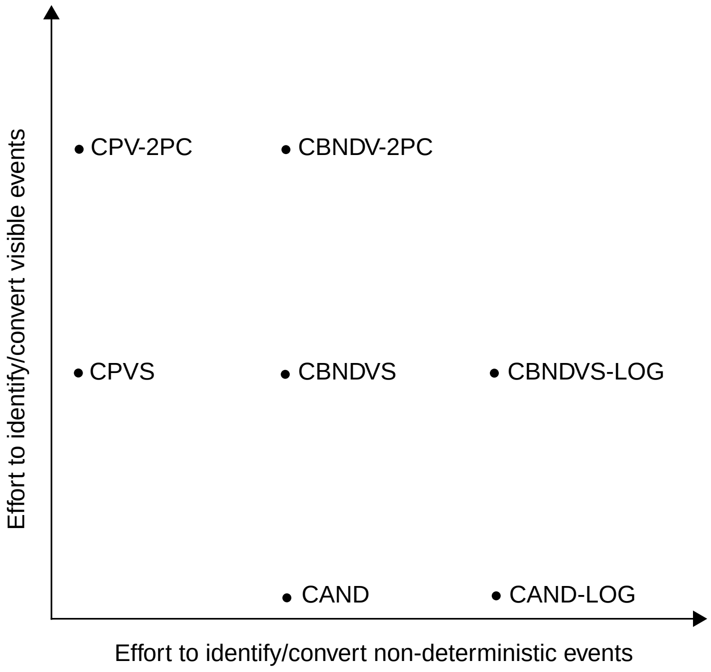

CPV-2PC
CBNDV-2PC
CPVS
CBNDVS
CBNDVS-LOG
CAND
CAND-LOG
Effort to identify/convert visible events
Effort to identify/convert non-deterministic events

Figure 6.1: Recovery protocols in the protocol space

We implement seven recovery protocols that each represent a different way to uphold Theorem 2. The seemingly missing CBNDV-2PC-LOG protocol that would occupy the upper right corner is technically possible. However, the logging mechanism we use in which we make message receives deterministic is not compatible with two-phase commit, which may need to abort messages. If our logging system only logged forms of non-determinism other than receive events, then we could combine 2PC and logging.

ferent processes in the computation, ensuring that all the processes commit or abort atomically. Interestingly, we implemented the 2PC server as a volatile process that keeps transaction commit records in a volatile array. Then, so that computations can survive the failure of the system on which the 2PC server runs, we made the server recoverable by simply linking it with Discount Checking.

Figure 6.1 places these recovery protocols in the protocol space developed in Section 2.3.1. Note that we do not combine logging with two-phase commit. Logging receive events to make them deterministic assumes the received message will not be aborted. Hence the receiving process cannot participate properly in a subsequent two-phase commit if the sender asks it to abort the logged receive event. A protocol that logged forms of non-determinism other than message receive events could take advantage of two-phase commit.

In Figure 6.2, we show the relation between our seven recovery protocols, Vistagrams, and the existing recovery protocols we discussed in Section 2.3.1. Note that several of the protocols we implement in this chapter fall in the previously unexplored center of the protocol space. These protocols take advantage of both identifying or logging non-determinism and tracking or converting send and visible events.

6.2.3 Applications

Recovery research has focused traditionally on recovering long-running, scientific computations. For our study, we will focus instead on recovering general, interactive applications. We choose this application domain because it is precisely this domain in which a human user is most directly inconvenienced by failures. Furthermore, recovering interactive applications is particularly challenging. These applications typically maintain large amounts of kernel state, and they execute visible and non-deterministic events frequently. As a result, these applications can stress any recovery protocol.

We use our checkpointing systems and recovery protocols to recover five applications: *vi*, *oleo*, *magic*, *TreadMarks*, and *xpilot*.

vi is one of the earlier text editors for Unix (the version we use is *nvi*); *oleo* is a spreadsheet program; and *magic* is a graphical VLSI layout editor. We make the execution of *nvi*, *oleo*, and *magic* repeatable by having them read user input from a script. For *nvi* and *oleo*, we simulate a very fast typist by delaying 100 milliseconds each time the program asks for a character. For *magic*, we delay 1 second between mouse-generated commands. *vi*, *oleo*, and *magic* are all local applications, i.e. they send no messages (we treat *magic*'s messages to the X server as visible events).

TreadMarks is a distributed shared memory system [Keleher94]. We run an N-body simulation called *Barnes-Hut* in the shared memory environment *TreadMarks* provides, with the simulation configured to run on four processors. We choose Barnes-Hut because it is the largest of the example applications shipped with *TreadMarks*.

xpilot is a distributed, graphical, real-time game. We run *xpilot* with three clients and a game server (all on separate computers) and play the game with a continuous stream of input at all clients.

For *magic*, *xpilot*, and *TreadMarks*, the only source modifications needed to run these applications on Discount Checking are those described in Section 5.3: adding a call to `dc_init` in `main()` and including `dc.h` in `main.c`. *nvi* and *oleo* are full-screen text programs that manipulate the terminal state through the `curses` library. In addition to the two modifications above, to

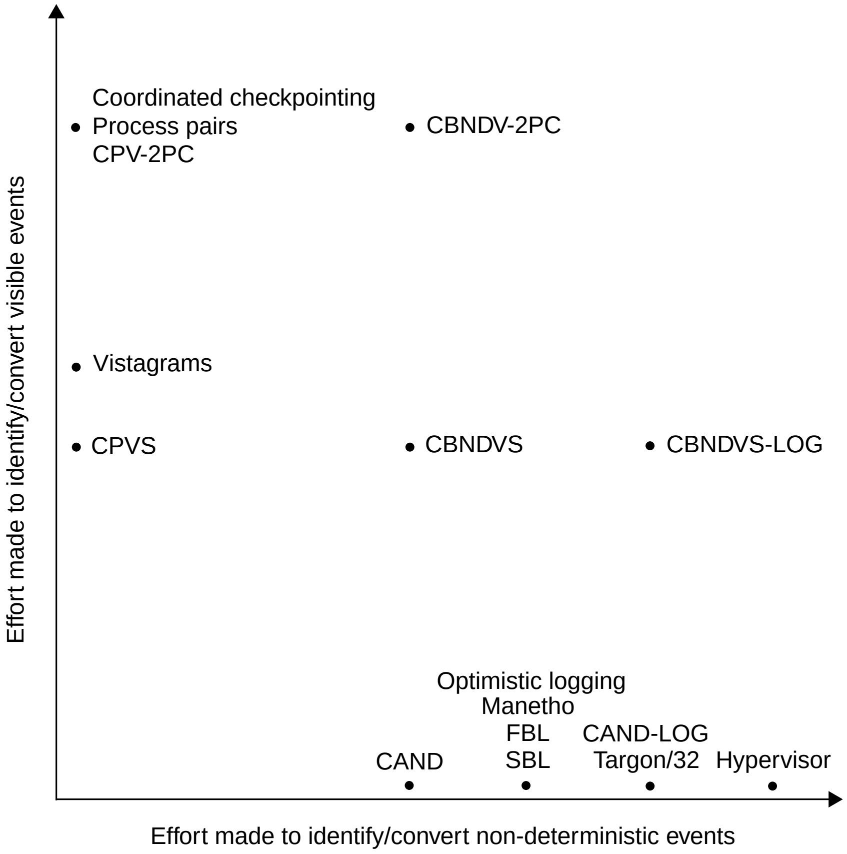


Figure 6.2: Relation between all recovery protocols we study

In this plot we show the relative positions of the new protocols we implement in this chapter, along with Vistagrams and the protocols we discussed in Section 2.3.1. In Section 2.3.1, we observed that the vast middle of the protocol space, away from the axes, was completely unexplored. Note that several of our new protocols fall in this portion of the space.

Processor	Pentium II (400 MHz)
L1 Cache Size	16 KB instruction / 16 KB data
L2 Cache Size	512 KB
Motherboard	Acer AX6B (Intel 440 BX)
Memory	128 MB (100 MHz SDRAM)
Network Card	Intel EtherExpress Pro 10/100B
Network	100 Mb/s switched Ethernet (Intel Express 10/100 Fast Ethernet Switch)
Disk	IBM Ultrastar DCAS-34330W (ultra-wide SCSI)

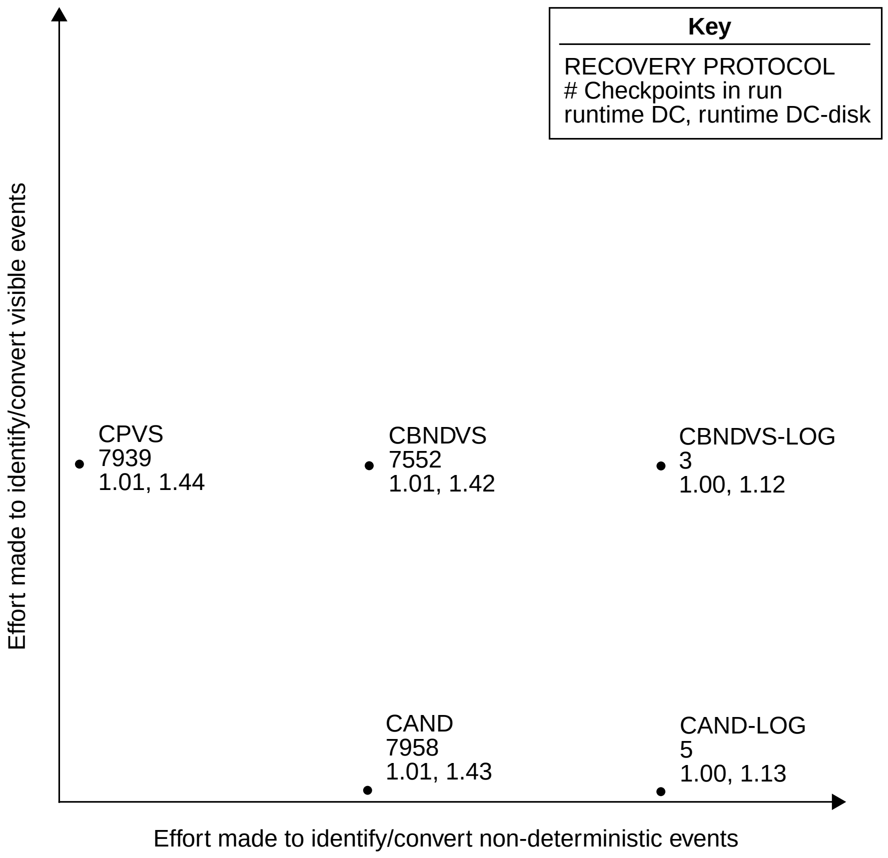
Table 6.1: Experimental Platform

each of these applications we add a function that reinitializes the terminal. This function consists of six lines of `curses` calls. We pass a pointer to this function to `dc_init`, requesting that it should be called during recovery. All five applications are then recompiled and linked with `libdc.a`.

6.2.4 Results

We summarize the hardware we use for our experiments in Table 6.1. Each machine runs FreeBSD 2.2.7 with Rio. Rio was turned off when using Discount Checking-disk. All points represent the average of five runs. Standard deviation for each application was less than 1% of the mean for Discount Checking, and less than 4% of the mean for Discount Checking-disk.

Figures 6.3 through 6.7 display the results of running the five applications using both Discount Checking and Discount Checking-disk to implement all seven recovery protocols. Since we use the two-phase commit protocol to coordinate commits between processes in message passing applications, we show the results for CPV-2PC and CBNDV-2PC only for applications that send messages. For each application, we show each of the protocols we implement in the protocol space we developed in Section 2.3.1. At each point on the plot, we list the name of the protocol that point represents, the number of commits executed in the run of the application, and the run-time ratios for Discount Checking and Discount Checking-disk. The run-time ratio is the running time of the recoverable version of the application divided by the running time of the baseline, non-recoverable application. Baseline running times are 798 seconds for *nvi*, 54 seconds for *oleo*, 89 seconds for *magic*, and 15 seconds for *TreadMarks Barnes-Hut*. Because *xpilot* is a real-time, continuous program, we report its performance as the frame rate it can sustain rather than run-time expansion.

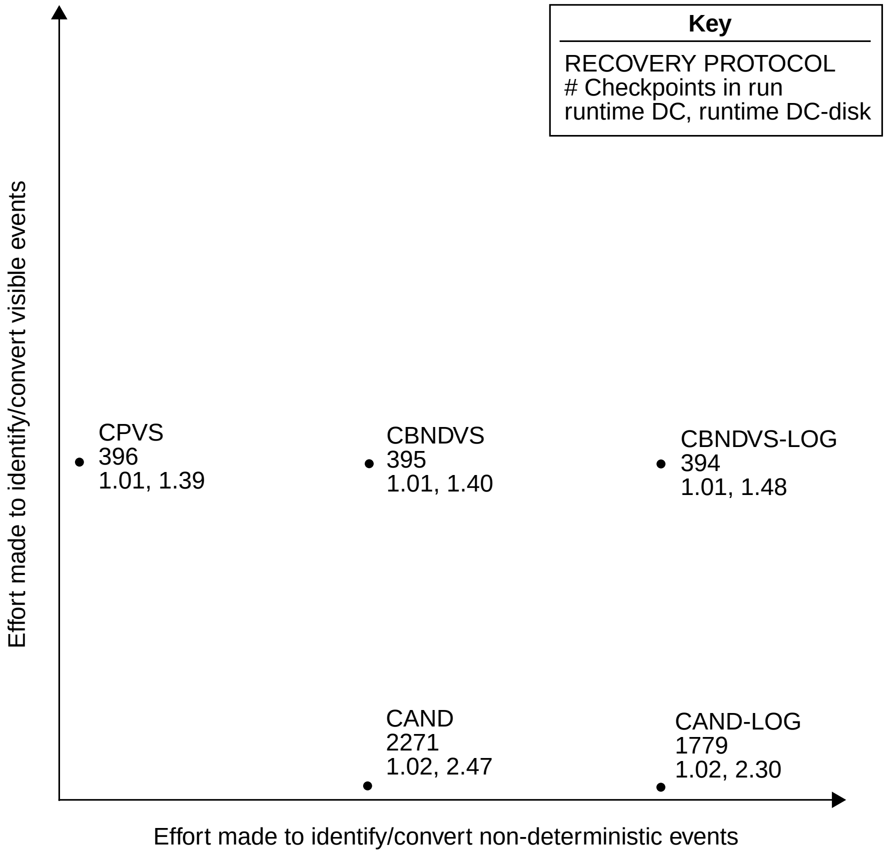


Key to Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CAND-LOG: commit after non-deterministic with keyboard input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with keyboard input logged.

Figure 6.3: Performance of recovery protocols for *nvi*

This plot shows the performance of *nvi* on five of the recovery protocols from Section 6.2.2. For *nvi*, committing only for visible events does not greatly reduce the number of checkpoints taken. Making non-deterministic events deterministic with logging does however. Logging reduces the number of checkpoints from almost 8000 to just a handful. Although overhead is negligible for all the Rio-based runs, logging reduces the disk-based overhead to an acceptable 12%.

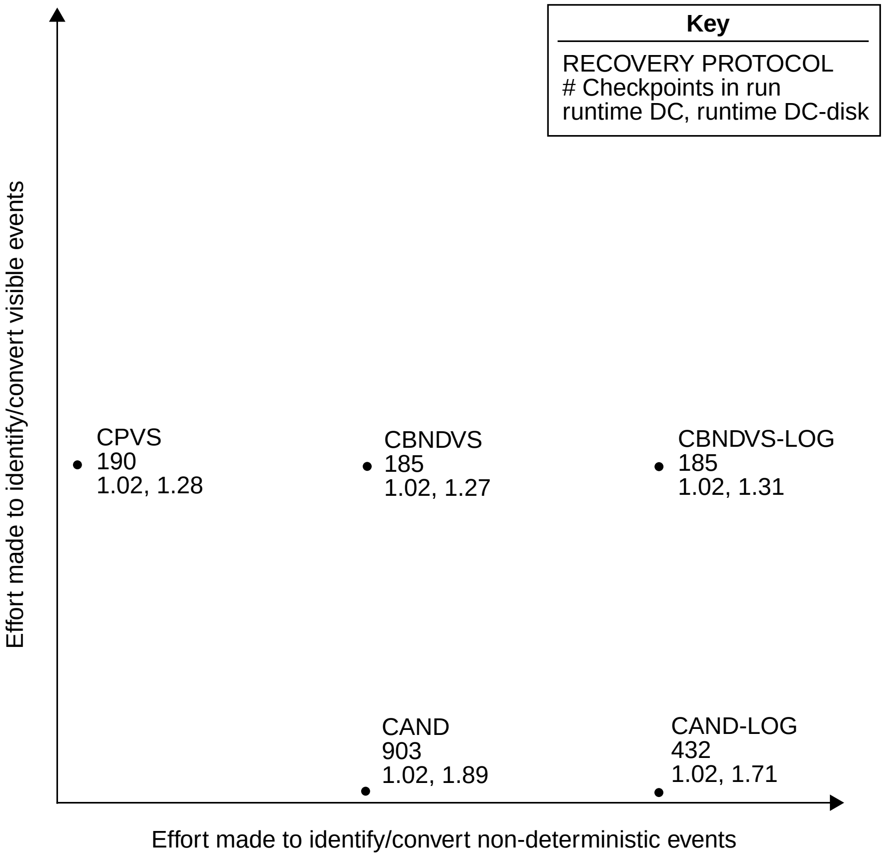


Key to Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CAND-LOG: commit after non-deterministic with keyboard input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with keyboard input logged.

Figure 6.4: Performance of recovery protocols for *oleo*

This plot shows the performance of *oleo* on five of the recovery protocols from Section 6.2.2. Unlike *nvi*, logging does not help *oleo*. Instead, *oleo* is most helped by reducing the number of events it treats as visible. Treating only writes to the screen as visible (rather than potentially all events) reduces the number of checkpoints in the run from around 2000 to around 400. Although the Rio-based runs are already quite low overhead, this reduction greatly reduces the disk-based overhead.

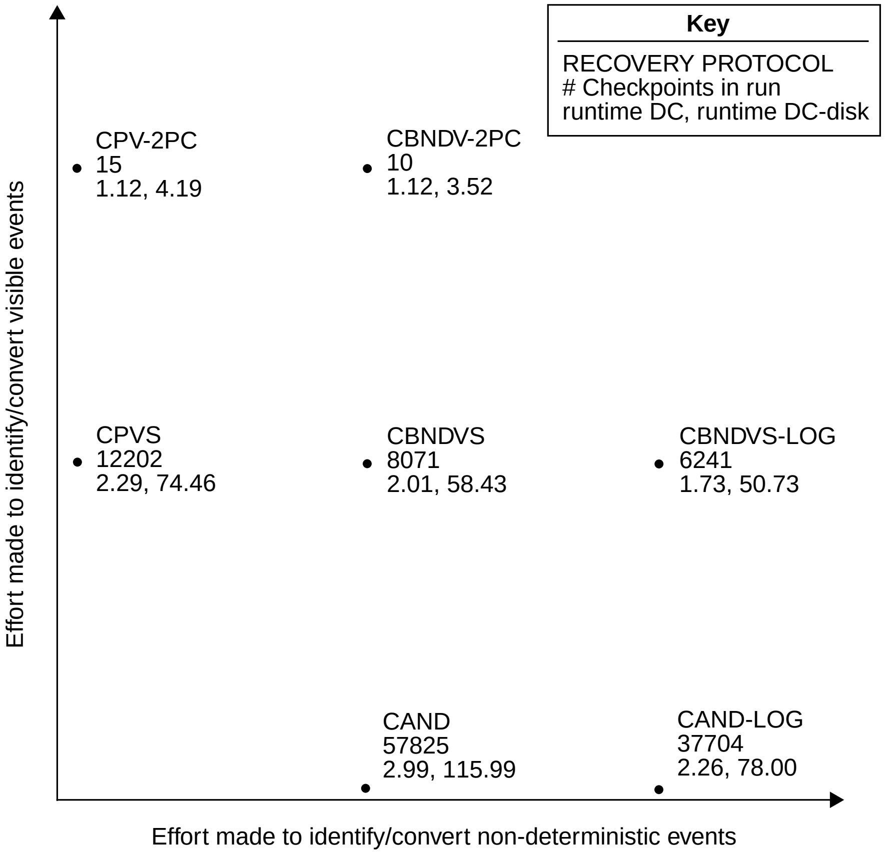


Key to Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CAND-LOG: commit after non-deterministic with keyboard input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with keyboard input logged.

Figure 6.5: Performance of recovery protocols for *magic*

This plot shows the performance of *magic* on five of the recovery protocols from Section 6.2.2. *magic* benefits both from converting non-determinism with logging, and tracking true visible events. Logging reduces the number of checkpoints in *magic*'s run by roughly half. Treating only writes to the screen as visible however reduces them by almost 80%. This reduction has little effect on the overhead of the Rio-based runs, but the disk-based runs are helped significantly, reducing overhead from 89% to 27%.

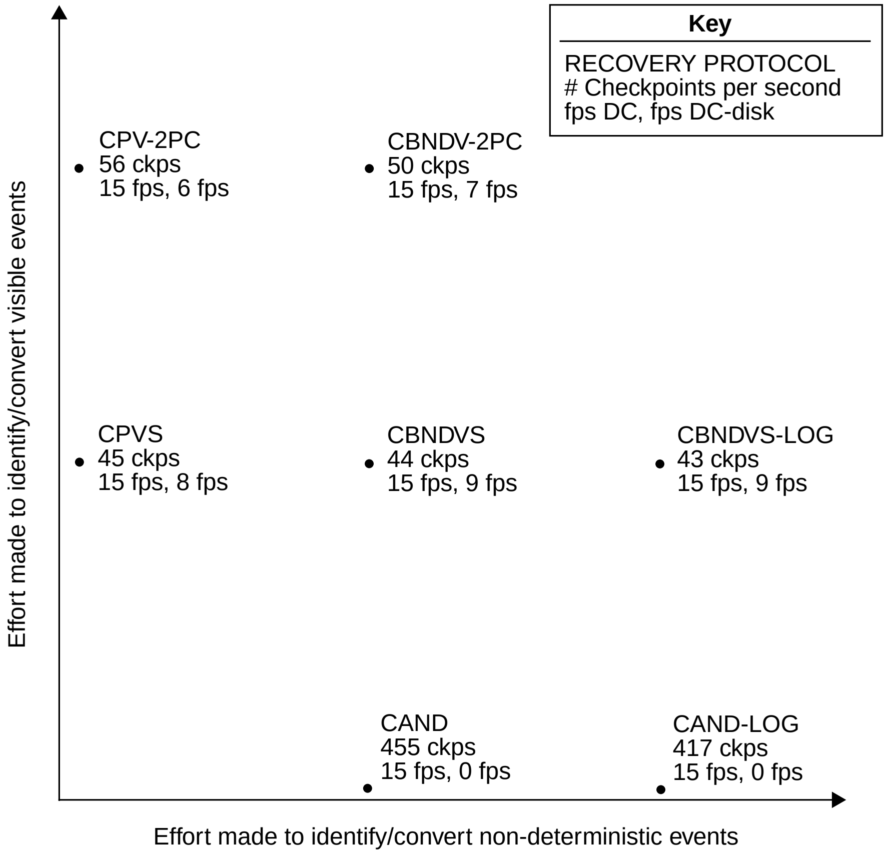


Key to Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CAND-LOG: commit after non-deterministic with messages and keyboard input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with messages and keyboard input logged.
CPV-2PC: commit prior to visible, using two-phase commit to commit multiple processes together.
CBNDV-2PC: commit between non-deterministic and visible, using two-phase commit to commit multiple processes together.

Figure 6.6: Performance of recovery protocols for *TreadMarks*

This plot shows the performance of *TreadMarks Barnes-Hut* on seven of the recovery protocols from Section 6.2.2. TreadMarks executes many non-deterministic and send events, but few visible events. The best protocols for TreadMarks use treat only writes to the screen as visible using two-phase commit. Doing so reduces the overhead on Rio from almost 200% to just 12%. Unfortunately, disk-based checkpoints cannot keep up with this application.



Key to Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CAND-LOG: commit after non-deterministic with messages and keyboard input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with messages and keyboard input logged.
CPV-2PC: commit prior to visible, using two-phase commit to commit multiple processes together.
CBNDV-2PC: commit between non-deterministic and visible, using two-phase commit to commit multiple processes together.

Figure 6.7: Performance of recovery protocols for *xpilot*

This plot shows the performance of *xpilot* on seven of the recovery protocols from Section 6.2.2. Because *xpilot* is a real-time application, we report its performance as sustainable frames per second (fps), where full speed is 15 fps. We report the number of checkpoints for *xpilot* as the checkpoints per second (ckps) rate amongst all the clients and the server. All protocols with Discount Checking on Rio are able to sustain the full 15 fps. With Discount Checking on disk, CBNDVS and CBNDVS-LOG fare best at 9 fps.

Higher frame rates indicate better interactivity, with full speed being 15 frames per second. Checkpoints for *xpilot* are given as the largest checkpoints per second rate executed by any client or server, measured at 15 frames per second.

We can make a number of interesting observations from the results in Figures 6.3 through 6.7. As expected, the number of commits generally decreases for protocols that are farther from the origin. As a designer expends more effort to treat as few events as possible as visible, or convert non-deterministic events into deterministic ones, he or she usually is rewarded with fewer commits and better performance. The sole exception to this rule is *xpilot*. In *xpilot*, using two-phase commit *increases* the number of checkpoints. This increase is because all processes must checkpoint whenever any of them executes a visible event (i.e. sending output to the X server). This increase in checkpoint frequency is greater than the decrease in checkpoint frequency that results from not needing to checkpoint before sending a message.

Second, note that Discount Checking adds negligible (< 1-2%) overhead to all the applications but TreadMarks, even for recovery protocols that generate many commits (such as CAND and CPVS). Because Discount Checking takes advantage of reliable main memory and fast transactions, it is able to take checkpoints very quickly: under 2 milliseconds per checkpoint for these applications. Discount Checking's strong performance substantiates our claim that with a fast commit, simple recovery protocols can be efficient.

While Discount Checking-disk cannot match Discount Checking's performance, it is nonetheless able to provide failure transparency with acceptable overhead for some applications. For example, Discount Checking-disk expands the running time of *nvi* by as little as 12%, *oleo* by 39%, and *magic* by 27%. *xpilot* can sustain a rate of 9 frames per second, which is 40% lower than the full frame rate. As expected, Discount Checking-disk is much more sensitive to the number of commits generated by a recovery protocol than Discount Checking. This sensitivity results from its writing to disk rather than memory. We conclude that with a slower commit, using more complex protocols that reduce commit frequency (such as those that use logging and two-phase commit) is essential.

Third, note that different recovery protocols benefit the various applications in different ways. For *nvi*, logging is the most effective way to reduce commit frequency. Logging keyboard input is sufficient to eliminate most non-determinism in *nvi*, and thus most checkpoints.

For *oleo* and *magic*, however, logging message receives and user input does not help appreciably, because these applications have other sources of non-determinism that are not logged. The system call `gettimeofday` is the major source of non-determinism in *oleo*, and signals are the major source of non-determinism in *magic*. For these applications, our results show it is most helpful to uphold the commit invariant by tracking the true visible events, and committing before them rather than after each of the more numerous non-deterministic events.

Overhead for *TreadMarks Barnes-Hut* is higher than other applications because *TreadMarks Barnes-Hut* has a large working set and is compute-bound rather than user-bound. Our results show that the best recovery protocols for *TreadMarks Barnes-Hut* are those that do not treat send events as visible. *TreadMarks* sends messages and executes non-deterministic events (message receives and signals) very frequently, which results in a high checkpoint frequency for most protocols. However, *TreadMarks Barnes-Hut* executes very few true visible events (just a handful of writes to the screen), so using 2PC to let it treat only these events as visible, and not sends, gets rid of most checkpoints.

Note that different applications achieve the lowest overhead with different protocols. Thus, no one protocol is appropriate for all workloads. Each protocol that provides low overhead for some application does so by exploiting a class of events that is rare for that application. For example, *TreadMarks* performs best with the two protocols that use two-phase commit. These protocols force *TreadMarks* to commit only once process executes a rare visible event. Similarly, in *magic,* writes to the screen (or X server) are much less common than non-deterministic events, and its best protocols exploit this fact. However, it is conceivable that some applications will not have a class of events that are rare and thus might have poor performance on all our protocols. Consider an interactive graphical simulation that uses a system like *TreadMarks* to perform a scientific simulation in real time while driving a graphical front end that executes copious visible events. Unlike TreadMarks Barnes-Hut, visible events in this application would be common and using two-phase commit may not help. In general, protocols that both minimize and track visible events, and identify and convert non-deterministic events yield more robust performance across a range of applications.

6.3 Related Work

A number of researchers have built systems that attempt to provide some flavor of failure transparency. For example, the Tandem NonStop [Bartlett81], Publishing [Powell83], Targon/32 [Borg89], and Hypervisor [Bressoud95] systems all use special hardware to implement recovery protocols to allow processes to survive hardware failures. In comparison, Discount Checking is designed to provide failure transparency for software faults, which are more common than hardware faults. In fact, Discount Checking will recover applications from any fault—hardware, software, or application—as long as the fault is transient and fail-stop. Furthermore, it does so without requiring special hardware.

Traditional checkpointing systems provide a tool that applications can use to help them survive software and hardware failures [Plank95]. These checkpointing systems are targeted for scientific applications with little kernel state, and that rarely execute visible events. In contrast, Discount Checking is targeted for general, interactive applications with copious kernel state.

6.4 Conclusion

Computer users know that system and application failures are all too common. Providing failure transparency as a fundamental abstraction of the operating system has the potential to make computers far more pleasant to use. Our goal with this chapter has been to show that doing so is feasible.

Our study has shown that providing failure transparency is feasible for a difficult class of applications without modifying those applications and without significantly degrading performance. For some applications and recovery protocols, we find it is possible to provide failure transparency using disk instead of reliable main memory. This result suggests the importance of new research into providing disk-based, full-process checkpointers that are optimized for small checkpoints, and that provide timeliness guarantees.

Interestingly, we find that since every application presents a different mix of non-deterministic events, send and receive events, and visible events, no one protocol works well for all applications. A real system that attempted to provide failure transparency for all applications would most likely want to adaptively select a recovery protocol based on workload observation.

Finally, we were pleasantly surprised to find that our user-level checkpointing library was able to capture sufficient kernel state to recover a large class of complex, real applications.

Our hope is that this work will encourage new research into the problem of providing failure transparency as a fundamental abstraction of modern operating systems. We also hope it will direct recovery research toward interactive applications. In this domain, recovery research can do a great deal to improve the relationship people have with their computers.

CHAPTER 7

CONCLUSION AND FUTURE WORK

We conclude by looking back over our progress, and examining where we have come. We then consider what remains to be done, and what new avenues of investigation have opened up out of our inquiry.

7.1 Conclusion

It is hard to hold your head up high as a computer scientist these days. Yes, people are happy with their web and their Duke Nukem 3D, and all the other spoils of 50 years of computer research. But computer users are increasingly, vociferously unhappy with the reliability of their computer systems. And rightly so: the most common computer systems in use today crash frequently.

What does computer science have to say about dealing with these failures? As successful as 30 years of fault tolerance research has been, it has focused its efforts on surviving hardware failures in high end business, factory, and scientific systems. The fault tolerance literature is less helpful for those who aim to build commodity systems that survive software failures. Our goal with this dissertation is to fill some of this void.

We propose that operating systems should provide the illusion they do not fail. We call this illusion the abstraction of failure transparency. Throughout this dissertation, we inquire into whether failure transparency is feasible, and how systems can attain it. The approaches to failure transparency we develop and examine all deal with recovering a system after a failure has occurred.

Despite the countless recovery protocols proposed in the canon of fault tolerance research, there have been no fundamental rules for recovery that are relevant across all protocols. The starting point for our inquiry, and one of our major contributions, is our theory of consistent recovery. The theory provides these fundamental rules. We find that guaranteeing consistent recovery is as simple as always making sure that at all times all non-deterministic events that causally precede visible events are committed. Furthermore, we find that if an application does not follow this rule, inconsistent recovery is possible. Interestingly, we show that all existing recovery protocols can be viewed as different approaches to upholding this rule. Our theory essentially unifies the many separate areas of recovery research, such as message logging and distributed checkpointing, and shows them to all be part of the same fundamental space of protocols. Furthermore, the theory provides the roadmap for attaining failure transparency.

One of the implications of our theory is that fast commits allow for extreme simplicity and flexibility in constructing a recovery protocol. So that we can ourselves have this flexibility in our quest for failure transparency, we construct a system that provides a fast commit. Our Vista transaction library provides lightweight transactions that do as little as possible without being useless. It is built upon the reliable memory of the Rio File Cache. Programmers can use Vista to allocate persistent memory and then make atomic and durable updates to it. We eschew providing other transaction features such as isolation, distribution, and nesting on the grounds that they complicate and slow the transaction mechanism, are superfluous for many applications, and can be easily added on top of Vista. As a result of Vista's simplicity and Rio's speed, small Vista transactions take on the order of a microsecond. We compare Vista's performance with that of RVM, a similar disk-based transaction system that also provides atomic and durable transactions on memory. Since Vista operates directly on persistent memory, and RVM uses disks for persistence, we would expect that Vista would be significantly faster than RVM, and it is. Small Vista transactions are about 2000 times faster than similar RVM transactions. More surprisingly, of this factor of 2000, we find only a factor of 20 comes from using Rio's fast stable storage. The remaining factor of 100 results from the simplicity of using directly addressable persistent memory. To further underscore the differences in complexity between RVM and Vista, Vista's source code is well under 1/10th of RVM's in size. We conclude that systems builders pay a hefty price in complexity for disks' slow performance.

Armed with Vista's fast commit, we construct our first distributed recovery protocol, Vistagrams. With Vistagrams, we show how to use reliable memory and fast transactions to guarantee consistent distributed recovery and find that Vistagrams has almost no overhead for the consistent recovery it provides. However, Vista and Vistagrams are merely a tool for programmers that aim to build applications that guarantee consistent recovery. We would like to be able to guarantee consistent recovery for applications in which the programmer has given no consideration to recovery. The culprit in this shortcoming is actually the nature of transactions in general: programmers have to insert them into applications. To provide failure transparency, we will need a flavor of commit different from Vista's transactions, one that is amenable to automatic introduction into applications.

We next construct Discount Checking, a user-level, full-process checkpointing system that will provide the commit we need. Discount Checking is built on Rio's reliable memory and Vista's fast transactions. Although Discount Checking is a user-level library, it manages to commit sufficient kernel state to recover a large class of real applications. It does so by trapping system calls that manipulate kernel state and persistently noting the update. Then after a failure, it directly reconstructs that kernel state by redoing the appropriate system calls. We find that Discount Checking can take a checkpoint in as little as 50 µs. As importantly, Discount Checking requires almost no modification of the application: the programmer need merely add two lines to `main.c` and link with `libdc.a`. Discount Checking determines which portions of the address space the application has modified during each checkpoint interval, and it does so with no programmer intervention.

With Discount Checking's fast and transparent checkpoints, we are finally in a position to provide failure transparency. All that remains is to have Discount Checking automatically checkpoint running applications as needed to guarantee consistent recovery. There are many ways Discount Checking can uphold Theorem 2 to do so. We round out our inquiry by examining which recovery protocols that fall out of Theorem 2 work best for our target applications. We implement seven different recovery protocols within Discount Checking. We find that no one protocol works best for all applications, implying that careful selection of a protocol appropriate for a given application can be important in systems providing failure transparency. We also find that for all but one of the real applications in our experiment, Discount Checking can provide failure transparency with overhead in the 0-2% range. For the remaining application, TreadMarks, Discount Checking can provide failure transparency with 12% overhead. In order to find out whether we can provide failure transparency using disk for stable storage instead of reliable memory, we also run all our experiments using a variant of Discount Checking that uses disk for persistent data. We find that Discount Checking-disk cannot provide failure transparency with acceptable overhead for several of our target applications. However, for the remaining applications, Discount Checking-disk slows performance by only 12-39%.

We happily conclude that failure transparency is feasible for the challenging real applications we target, and with low overhead. We can make several other interesting observations from our results.

First of all, conventional wisdom has held that checkpointing the complete state of complex applications requires building your checkpointing system into the kernel. Only that way, the argument goes, can you be sure to preserve the application's copious kernel state. Our work has shown that checkpointers can capture sufficient kernel state at user level to allow full-process checkpointing for a large class of complex, real applications. Indeed, the applications in our experiments manipulate some of the more esoteric bits of kernel state that could conceivably be challenging to preserve, such as address space protections, TCP protocol state, file descriptor and socket options, and signal masks and handlers.

For most of our inquiry, the systems we build use reliable memory as stable storage. As a result, a subplot of our study is to examine the implications of reliable memory for recovery. The performance implications of the Rio File Cache for our work should be clear: Vista transactions are three orders of magnitude faster than their disk-based counterparts, Vistagrams has almost no overhead, Discount Checking checkpoints can complete in tens of microseconds. However, more important than speedup resulting from using reliable memory are the new ways it lets applications recover. Vistagrams and many of the protocols we develop in Chapter 6 represent new recovery techniques that are simply not be practical when using disk instead of reliable memory. Furthermore, reliable memory allows amazing simplicity of design—consider Vista's 720 lines of code, or the simplicity of the CAND or CPVS protocols.

Our decision to use reliable memory for our work has also had a notable, but unseen benefit. The performance of reliable memory allowed us to explore avenues of recovery that no self-respecting systems person would, avenues such as executing a full process checkpoint for every send event in a message bound application. It was once we had started down that path that we wondered how those simple but foreign recovery protocols related to all others. That inquiry led to the development of our theory of consistent recovery.

Hopefully the need for failure transparency is clear: failures are all too common. Luckily, years of computer science research has provided plenty of spare processor cycles that could well be harnessed to improve reliability. Our hope is that this work will encourage new research into providing failure transparency as a fundamental abstraction of modern operating systems. We have demonstrated the feasibility of doing so.

7.2 Future Work

Our experiments show that no one protocol out of the seven we implement performs best for all the applications in our study. Since different applications benefit from starkly different recovery protocols, an adaptive approach that dynamically selected a workload-appropriate protocol might fare well over a wide range of applications. Ideally, the system would be able to change recovery protocols as an application’s needs change.

Our experiments showed that using disk for stable storage is not a counterindication for providing failure transparency for some real applications. This result suggests the importance of new research into providing disk-based, lightweight, full-process checkpointing systems, that are optimized for small checkpoints, and that provide real-time guarantees.

The recovery approaches we explore all work for fail-stop (and transient) hardware failures, fail-stop operating system failures, and fail-stop application failures. However, application failures are often not fail-stop [Chandra98]. Recovering from non-fail-stop application failures is a challenging problem, and deserving of study. Furthermore, the problem of finding general techniques for recovering from application failures is largely unresearched.

In our work, we argue that future operating systems should provide failure transparency. However, current operating systems do not provide it, and OS developers may choose not to incorporate failure transparency into future systems. The holy grail of failure transparency research is therefore to provide failure transparency for unmodified, commodity operating systems, a challenging problem indeed.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Alvisi94] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 145–154, 1994.

[Alvisi95] Lorenzo Alvisi and Keith Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. In *Proceedings of the 1995 International Conference on Distributed Computing Systems*, pages 229–236, June 1995.

[Anderson91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPL OS-IV)*, pages 108–120, April 1991.

[APC96] The Power Protection Handbook. Technical report, American Power Conversion, 1996.

[Appel91] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 96–107, April 1991.

[Atkinson83] Malcolm Atkinson, Ken Chisholm, Paul Cockshott, and Richard Marshall. Algorithms for a Persistent Heap. *Software–Practice and Experience*, 13(3):259–271, March 1983.

[Baker92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile Memory for Fast Reliable File Systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22, October 1992.

[Bartlett81] Joel F. Bartlett. A NonStop Kernel. In *Proceedings of the 1981 Symposium on Operating System Principles*, pages 22–29, December 1981.

[Bensoussan72] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[Birman96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications, 1996.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrman, and Wolfgang Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Bressoud95] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[Chandra98] Subhachandra Chandra and Peter M. Chen. How Fail-Stop are Faulty Programs? In *Proceedings of the 1998 Symposium on Fault-Tolerant Computing (FTCS)*, pages 240–249, June 1998.

[Chandy72] K. M. Chandy and C. V. Ramamoorthy. Rollback and Recovery Strategies for Computer Programs. *IEEE Transactions on Computers*, C-21(6):546–556, June 1972.

[Chandy85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States in Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Chang88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Chen96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[Chen99] Peter M. Chen and David E. Lowell. Reliability Hierarchies. In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS)*, March 1999.

[Costa97] Diamanto Costa, Joao Carrierra, and Joao Gabriel Silva. WinFT: Using Off-the-shelf Computers on Industrial Environments. In *Proceedings of the 6th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '97)*, pages 39–44, 1997.

[DeWitt84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[Elnozahy92] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.

[Elnozahy96] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU TR 96-181, Carnegie Mellon University, 1996.

[GM92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[Golub90] David Golub, Randall Dean, Allessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[Gray78] J. N. Gray. *Operating Systems: An Advanced Course*. Springer-Verlag, 1978. Notes on Database Operating Systems.

[Gray86] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[Gray90] Jim Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Haerder83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the 1988 Symposium on Principles of Distributed Computing (PODC)*, pages 171–181, August 1988.

[Keleher94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Technical Conference*, 1994.

[Koo87] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lamb91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[Lamport78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson83] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 1983 Symposium on Operating System Principles*, pages 33–48, 1983.

[Li90] C-C. J. Li and W. K. Fuchs. CATCH–Compiler-Assisted Techniques for Checkpointing. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 74–81, 1990.

[Li94] Kai Li, J. F. Naughton, and James S. Plank. Low-Latency, Concurrent Checkpointing for Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[Lomet98] David Lomet and Gerhard Weikum. Efficient Transparent Application Recovery in Client-Server Information Systems. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 460–471, June 1998.

[Lowell97] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, October 1997.

[Lowell98a] David E. Lowell and Peter M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. Technical report, University of Michigan, December 1998.

[Lowell98b] David E. Lowell and Peter M. Chen. Persistent Messages in Local Transactions. In *Proceedings of the 1998 Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, June 1998.

[Lowell99] David E. Lowell and Peter M. Chen. The Theory and Practice of Failure Transparency. Technical report, University of Michigan, May 1999.

[Mullender93] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, 1993. Chapter 10.

[Ng97] Wee Teck Ng and Peter M. Chen. Integrating Reliable Memory in Databases. In *Proceedings of the 1997 International Conference on Very Large Data Bases (VLDB)*, pages 76–85, August 1997.

[Ng99] Wee Teck Ng and Peter M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)*, pages 76–83, June 1999.

[O'Toole93] James O'Toole, Scott Nettles, and David Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the 1993 Symposium on Operating Systems Principles*, December 1993.

[Ousterhout90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings USENIX Summer Conference*, pages 247–256, June 1990.

[Plank95] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.

[Powell83] Michael L. Powell and David L. Presotto. PUBLISHING: A Reliable Broadcast Communication Mechanism. In *Proceedings of the 1983 Symposium on Operating Systems Principles*, October 1983.

[Randell75] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[Rosenblum95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[Satyanarayanan93] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 1993 Symposium on Operating System Principles*, pages 146–160, December 1993.

[Schneider84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Slye96] J. H. Slye and E. N. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 250–259, June 1996.

[Soparkar90] N. Soparkar and A. Silberschatz. Data-Value Partitioning and Virtual Messages. In *Proceedings of the 1990 ACM Symposium on Principles of Database Systems*, pages 357–364, April 1990.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Sullivan91] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.

[Tannenbaum95] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, pages 40–48, February 1995.

[TPC90] TPC Benchmark B Standard Specification. Technical report, Transaction Processing Performance Council, August 1990.

[TPC96] TPC Benchmark C Standard Specification, Revision 3.2. Technical report, Transaction Processing Performance Council, August 1996.

[Wahbe93] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical Data Breakpoints: Design and Implementation. In *Proceedings of the 1993 Conference on Programming Langauge Design and Implementation (PLDI)*, pages 1–12, June 1993.

[Wang95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and Its Applications. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing (FTCS)*, June 1995.

[Yarvin93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer 1993 USENIX Conference*, 1993.