The Theory and Practice of Failure Transparency

David E. Lowell and Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{dlowell,pmchen}@eecs.umich.edu
http://www.eecs.umich.edu/Rio

Abstract: System and application failures are all too common. In this paper, we argue that operating systems should provide the abstraction of failure transparency—the illusion that systems and applications do not fail. We construct a theory of consistent recovery that is useful for systems that want to provide failure transparency. The theory defines precisely what constitutes consistent recovery and provides a simple invariant that all applications must uphold to guarantee they get it. The theory unifies the various recovery protocols for achieving consistent recovery; existing protocols can be viewed as different ways of upholding our theory's invariant. We use the theory to design new recovery protocols and analyze existing protocols. We conclude by evaluating performance for a suite of recovery protocols. We focus our study on interactive programs, an application domain for which it is challenging to provide failure transparency. Our results indicate that it is possible to provide failure transparency for general applications with overhead of 1-2% for systems with reliable main memory, and 10-40% for disk-based systems.

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), and non-file state in the system is lost completely during a crash. In particular, the state of running processes is lost 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 such as auto-saves. 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. Application recovery often loses recent changes to a user's file and also loses 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 paper.

We define *consistent recovery* as recovering from crashes in a way that makes failures transparent to the user. For an operating system to provide failure transparency, it must provide consistent recovery on behalf of applications, and do so without requiring programmer assistance.

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 possible recovery protocols has not been explored systematically, and it is difficult to discern the relationship between existing protocols.

Our goal in this paper 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 local 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, and exposing new ones. We will explore all these applications of the theory in this paper. Finally, we will show the theory in action by examining the performance tradeoffs 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. Theory of Consistent Recovery

This section describes informally our theory of consistent recovery. For a formal version of the theory and its proof, please see [Lowell99].

2.1. Definition of consistent recovery

For our purposes, a *computation* consists of one or more processes working together on a task. Each process computes by executing a sequence of events. *Visible events* are events whose results are observable to someone outside the computation (e.g. the user). Visible events have also been called “output events” or “output messages” [Elnozahy92]. Examples of visible events are writes to the screen and messages sent to printers.

Given a computation in which some subset of processes has failed, the goal of consistent recovery is to reconstruct the computation so it can continue to execute in a way that hides the failure from the user. Recovery is consistent if the user sees output from the computation corresponding to a correct execution of the program, despite its failure.

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 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 computation 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.

Let us now make the definition of consistent recovery more precise.

Definition: Consistent Recovery

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 equivalent to the sequence of visible events actually seen by the external observer.

By this definition, recovery is consistent as long as the sequence of outputs from the failed and recovered computation is *equivalent* to those that would be output by some legal (i.e. failure-free) execution of the process [Strom85].

This definition of consistent recovery establishes two constraints on how computations recover: a constraint of *safety* and a constraint of *liveness*. In order to meet the definition’s safety constraint, a computation must be sure not to execute a visible event after a failure that could appear in no legal sequence with the computation’s pre-failure visible events. The liveness constraint follows from the observation that the definition evaluates the consistency of recovery by comparing the output of a recovered computation with those of complete executions. If a computation recovers but is unable to output a complete sequence of visible events, its recovery cannot be consistent. In other words, a single failure cannot prevent a consistently recovered computation from executing to completion. Of course, continuous failures can always prevent a computation from executing to completion.

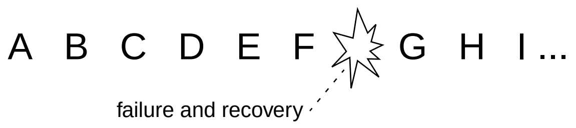


Figure 1: Recovery consistent assuming an *identity* equivalence function.

The equivalence of output sequences is governed by an *equivalence function*. This function takes as a parameter the sequence of visible events generated by an execution of the computation and returns *true* if they are equivalent to some failure-free execution, and *false* if not. This function is an application-specific encapsulation 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. Or instead, the application may allow recovery to cause the duplication of just the last visible event—an *okay to duplicate last visible* equivalence function (see Figure 2). An application may even be happy with an *any permutation* equivalence function that accepts a visible event sequence if it is some permutation of a legal sequence.

Consider an application that outputs the alphabet and defines and uses the *identity* equivalence function. Figure 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.

What would a computation have to do to guarantee consistent recovery according to some equivalence function? Processes can execute *commit events* to aid later recovery. By executing a commit event, a process guarantees that it can recover the state of the process at the time of the commit. 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.

A wide variety of equivalence functions are possible. Each varies in tractability, the constraints it places on recovery, and the usefulness of its recovered computations. For example, the *identity* equivalence function requires a protocol that commits the application atomically with each visible event. Since making the commit and visible events atomic requires special purpose hardware however, we conclude that the *identity* equivalence function is not tractable on mainstream systems. On the other hand, the *okay to duplicate last visible* equivalence function is tractable, but restrictive: it requires a commit immediately before every visible event. Much less restrictive would be the *any permutation* equivalence function, although it is almost useless in practice: self-respecting applications are typically not content with failures causing scrambled output.

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].

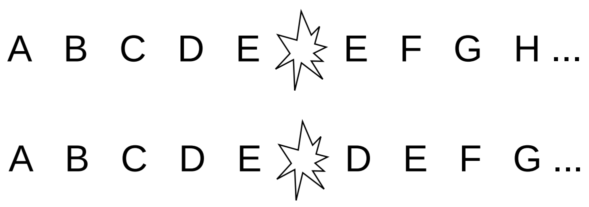


Figure 2: Analysis of two recoveries of the alphabet program. The first recovery is consistent assuming an *okay to duplicate last visible* equivalence function. The second recovery is not consistent by this equivalence function.

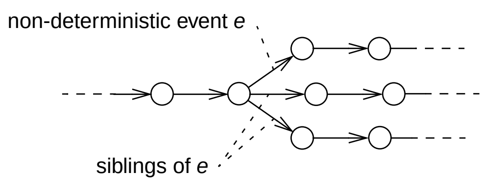


Figure 3: A portion of a state machine showing non-deterministic event *e* and its siblings.

All discussion of consistent recovery occurs either implicitly or explicitly in the context of a specific equivalence function. Although in general equivalence functions are 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 rest of this paper, we will assume that a sequence of visible events V output by a computation is equivalent to a sequence V' output by a failure-free execution of the computation if either V and V' are identical, or those events in V that differ from events in V' are repeats of earlier events from V. Such an equivalence function that allows duplicate visible events gives a great deal of flexibility in how we guarantee consistent recovery and is closely related to the one implicitly assumed by existing recovery protocols. Most importantly, it is a reasonable one to use in practice: typical users can overlook the duplication of earlier visible events while the system is recovering from a failure.

For a more detailed discussion of equivalence functions, please see [Lowell99].

2.2. Assumptions for general application recovery

Providing failure transparency as an operating system abstraction implies the ability to recover general applications without application-specific recovery code. This section describes 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.

The constraint of event undoability for rollback recovery is not too challenging to meet since most application events are changes to local state that are easily undoable. Other events, such as visible events, can be hard to undo. However, applications that strive for consistent recovery will not have to undo a visible event without re-executing it as doing so would likely make recovery inconsistent.

The redoability constraint of re-execution recovery is more challenging to meet, because it can take significant 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 after a failure. 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 re-delivered 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 it fails to do so, recovery will start from a buggy state. 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.

This paper 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 (non-deterministic), or a backup system must be used 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 3.2 unifies these two constraints and explores the interaction of different recovery protocols with the fail-stop model.

2.3. Achieving consistent recovery

We next turn our attention to the problem of achieving consistent recovery. In this paper we are concerned with the actions needed to ensure consistent recovery and the effect of those actions on failure-free performance. We do not address secondary (though still important) issues such as recovery time and stable storage space.

We first define some terminology. We model processes as state machines. Each state transition is called an *event*. A *non-deterministic event* is a transition from a state that has multiple next states possible during recovery. The other possible events out of that state are called *sibling* events. Figure

3 depicts a portion of a process's state machine in which a non-deterministic event *e* and its siblings appear. In real systems, non-deterministic events correspond to such acts as reading user input, receiving a message, taking a signal, or executing certain system calls such as `gettimeofday`.

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,

or e_1 is the send of a message and e_2 is its corresponding receive by another process,

or e_1 causally precedes *e* and *e* causally precedes e_2.

We may also say event e_2 *causally depends on* e_1. This notion of causality corresponds to Lamport's "happened before" relation [Lamport78]. Under our model, computation proceeds *asynchronously.* That is, there exist no bounds on either the relative speeds of processes or message delivery time. In an asynchronous system, the only way to order events is through causal precedence [Lamport78].

Computations recover from failures by restarting failed processes from their most recently committed state. If processes commit judiciously, they can ensure consistent recovery no matter when crashes happen. The following theorem defines when and how computations should commit to guarantee consistent recovery.

Theorem

Recovery of a computation is guaranteed to be consistent if and only if each process that executes a non-deterministic event that causally precedes a visible or commit event *e* later executes a commit event that itself causally precedes *e*.

The theorem states that to guarantee consistent recovery, a computation 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, the commit preserving that non-deterministic event must causally precede the visible event in question. Finally, the computation must ensure that no commit causally depends on an uncommitted non-deterministic event. We call these requirements the theorem's *commit invariant*.

Recall that our definition of consistent recovery has both safety and liveness components. The theorem as written guarantees both safety and liveness, although it can be decomposed into separate theorems for each.When event *e* in the theorem is a visible event, the theorem guarantees only safety. When *e* is a commit event, the theorem guarantees only liveness.

For the purposes of this paper, we make two assumptions about the behavior of general computations to assure the necessity of the theorem.

Assumption 1

Each non-deterministic event that causally precedes a visible event e_1 may have a sibling that causally precedes a different visible event e_2, and that e_1 and e_2 are not both contained in any failure-free execution.

This assumption states that if a process executes a non-deterministic event that leads to a visible event, executing a sibling of that non-deterministic event during recovery will cause inconsistency. This assumption is reasonable since in real systems, paths of execution through non-deterministic events do not usually join up with paths through siblings of those events.

Assumption 2

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

Let's call a process whose events causally depend on an aborted non-deterministic event an *orphan*. Assumption 2 implies that orphans will always cause problems for recovery by attempting to execute visible events that causally follow lost non-deterministic events.

The alternative to making Assumptions 1 and 2 is to exhaustively analyze which non-deterministic and commit events will causally precede future visible events, and which will not. Although we can conceive of state machines in which this analysis is not hard, most real state machines are far too complex to analyze completely. If one can perform this analysis, then the theorem is merely a sufficient guarantor of consistent recovery. Applications that uphold the commit invariant are still guaranteed consistent recovery, although they may commit more often than is strictly necessary. We explore the implications of not making these assumptions more fully in [Lowell99].

This theorem has a corollary concerning the possibility of recovery in the presence of events that are both non-deterministic and visible.

Corollary

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 executed atomically with a commit.

This result follows naturally from the theorem, as the theorem's invariant is impossible to uphold for these events.

For the proof of the theorem, please see the Appendix.

2.4. Upholding the commit invariant

As mentioned above, the commit invariant can be broken up into three separate requirements:

1. Every non-deterministic event that causally precedes a visible event *e* is committed by some commit event e^c.
2. e^c causally precedes *e*.
3. No commit event causally depends on an 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 each process to execute 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: if a non-deterministic event causally precedes a visible event, the commit event immediately following must as well (since the visible event cannot come between the non-deterministic event and its commit). 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, we can be assured it guarantees consistent recovery. We call this protocol *commit after non-deterministic* (CAND).

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. We call this protocol *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. Under this protocol, a process commits immediately before a visible or send event if that process has executed a non-deterministic event since its last commit. In so doing, this protocol ensures a commit after a 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 local send event, in case that send leads to the execution of a visible event by another process. Thus this protocol meets requirement 1. It also meets requirement 2 since each commit executed before a visible event causally precedes that visible event, and each commit before a send event must causally precede any downstream visible events. Finally, this protocol meets requirement 3 following the same reasoning as in CPVS. We call this protocol *commit between non-deterministic and visible or send* (CBNDVS). Note that it 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 4.4. 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 treat only the truly visible events as visible. 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, meeting 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, and 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 under the CAND or CBNDVS protocols by endeavoring to convert many non-deterministic events into deterministic ones. 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 the 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.

3. Applying the Theory

The theory has a number of practical uses. As we have seen, it helps make evident new recovery protocols. To our knowledge, CPVS and CBNDVS (and two protocols described in Section 4.2, CBNDVS-LOG, CBNDV-2PC) have never been proposed or implemented. We next turn our attention to discuss how the theory can be used to unify existing recovery protocols, and explore interactions with the fail-stop model.

3.1. Unifying existing recovery protocols

All existing recovery protocol 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.

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 also 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.

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.

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 [Alvisi93, 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).

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.

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 committed state.

3.1.6. Process pairs

Process pair systems, such as Tandem NonStop, uphold the commit invariant by committing before each vis-

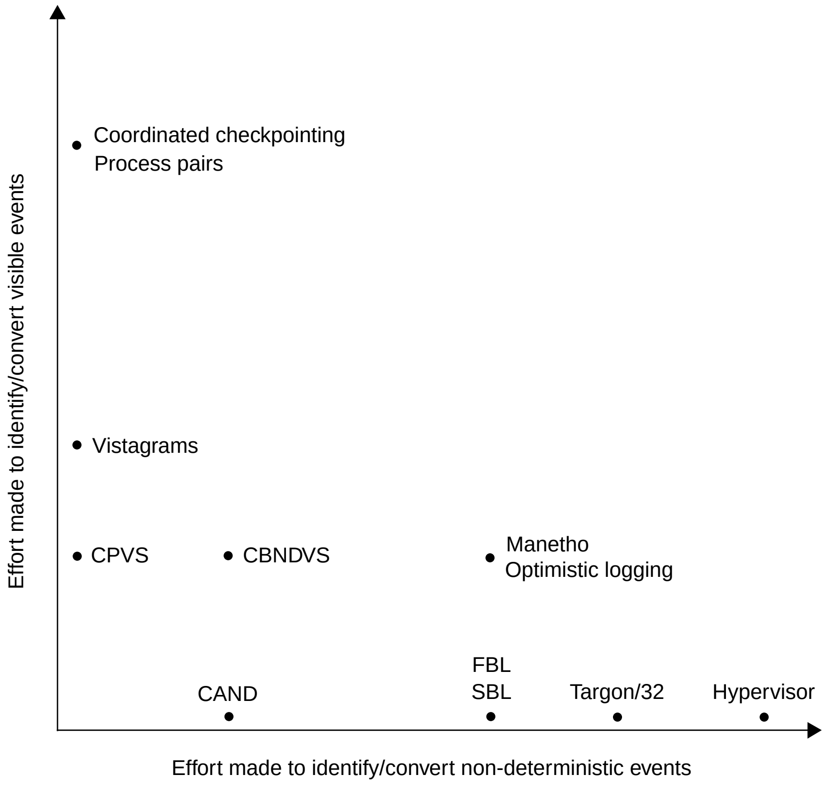


Figure 4: Protocol space. We can parameterize the space of all recovery protocols by effort made to identify and convert non-deterministic events and effort made to identify and convert visible events. This plot shows the point in such a space each protocol from Section 3.1 might occupy. For reference, we also show the CPVS, CBNDVS, and CAND protocols described in Section 2.4.

ible 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.

3.1.7. Vistagrams

As mentioned in Section 2.4, applications can uphold the commit invariant by committing before every message send or visible event. Vistagrams takes a similar approach to achieving consistent recovery [Lowell98]. Rather than committing state before every send, however, Vistagrams upholds the commit invariant by deferring send events until after the next naturally occurring commit executed by the application. Applications are responsible for committing their current transaction before doing a visible event.

3.1.8. 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 defer commits as late as possible. Still others do some of each. 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 4 shows how the protocols of Section 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 4 most existing protocols fall very close to the axes (Manetho and optimistic logging being exceptions). 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 4.2, and many more are possible.

3.2. 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 2.4) is guaranteed to commit buggy application state, and all application failures will violate fail-stop. Therefore, systems that use eager, complete logging (e.g. Targon/32 and 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, while still guaranteeing consistent recovery. Committing or logging can only be deferred up until the next causally dependent visible event. Protocols that defer these writes to stable storage, such as CPVS, CBNDVS, lazy logging, and coordinated checkpointing, all increase the likelihood that failures will be fail-stop.

4. Practice of Failure Transparency

The first three sections of this paper have been primarily theoretical. We developed a definition of consistent recovery, proved the invariant that a program must maintain to perform consistent recovery, and showed how existing recovery protocols maintain the invariant.

This next section of the paper illustrates the practical application of the theory and demonstrates the feasibility of providing failure transparency on a variety of challenging applications. We design seven recovery protocols that maintain the commit invariant in different ways and measure the performance of these recovery protocols on a variety of real applications. We have verified that all applications used in this section recover consistently from induced operating system and application crashes using all seven protocols.

4.1. Discount Checking

We built a 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 [Chen96] and the fast transactions provided by the Vista lightweight transaction library [Lowell97]. In this paper, we also use a variant called Discount Checking-disk that does not assume the presence of Rio's reliable main memory.

While the design of Discount Checking is not the focus of this paper, we do want to point out several unique aspects of Discount Checking (see [Lowell99] for more details).

First, Discount Checking's architecture differs from other checkpointing systems because it is optimized for reliable main memory and is layered on top of fast transactions. Conventional checkpointing systems assume memory is lost during operating system crashes and must copy the process state to a separate stable store [Plank95]. In contrast, Discount Checking loads the process address space into Vista's recoverable memory and executes the process directly from this space. Discount Checking uses Vista's atomic transactions to transition the committed process state atomically from one checkpoint to the next. Discount Checking is, to our knowledge, the first system to use reliable main memory in this direct manner and is the first checkpointing system built on top of a general-purpose transaction system.

Second, Discount Checking saves a more complete state of the process than most other checkpointing systems. Checkpointing systems have focused traditionally on recovering long-running, scientific computations that have little kernel state associated with the process. In contrast, we seek to provide failure transparency for interactive applications (e.g. editors, spreadsheets, CAD tools, games) to hide failures from ordinary users. These interactive applications have significant kernel state that must be committed before the crash and recovered after the crash. Discount Checking saves the process address space by mapping it into Vista's memory, as described above. Discount Checking saves the processor's register contents by copying the registers (e.g. stack pointer, program counter, general-purpose registers) into Vista's memory at each checkpoint. Our basic strategy for saving kernel state is to intercept system calls that modify kernel state (by wrapping `libc` calls), 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 saved and recovered by Discount Checking: open files and file positions, open sockets, bound and connected sockets, signal handlers, timers, TCP protocol stack, process IDs, and user-level page protections (e.g. those used in copy-on-write systems). Discount Checking also undoes uncommitted file modifications [Wang95]. In general, saving all relevant kernel state requires that the checkpointing system be part of the kernel. However, we were surprised that it was possible for a user-level checkpointing system to be made complete enough to recover a large class of complex, real applications.

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. Discount Checking-disk does not yet have the code needed to recover applications after failures (unlike Discount Checking, which successfully recovers all applications in this paper).

4.2. Recovery Protocols

There are many ways to maintain the commit invariant, as described in Section 2.4. 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 using Discount Checking.

As mentioned above, Discount Checking traps system calls in order to preserve kernel state. In the same manner, it can also trap system calls corresponding to visible, non-deterministic, or send events for special handling by the recovery protocol in use. By trapping these events, each protocol can be sure to uphold the commit invariant on behalf of the application.

Several protocols need to specially handle an application's non-deterministic events, such as reading user input, checking the real-time clock, taking a signal, or receiving a message. To support these protocols, Discount Checking traps calls to `recv`, `recvfrom`, `recvmsg`, `read`, `gettimeofday`, `select`, `bind`, and the application's signal handlers. For the protocols that need to handle an application's visible events, Discount Checking traps the `write` system call. Finally, some protocols want to specially handle message send events. For these protocols, Discount Checking traps calls to `send`, `sendmsg`, `sendto`, and `write`.

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 or send event by trapping the appropriate system calls. As described in Section 2.4, doing so upholds the commit invariant and guarantees consistent recovery.

Commit after non-deterministic event (CAND): As described in Section 2.4, Discount Checking forces each process to checkpoint immediately after executing one of the above non-deterministic events.

Commit between non-deterministic event and visible event or send (CBNDVS): As described in Section 2.4, each process commits between executing a non-deterministic event and executing a visible event or sending a message. To implement this protocol, Discount Checking sets a flag when a process executes one of the above non-deterministic events. When that process later attempts to execute a visible or send event while the flag is set, it first takes a checkpoint and clears the flag. This protocol will always execute the same or fewer commits as CPVS or CAND.

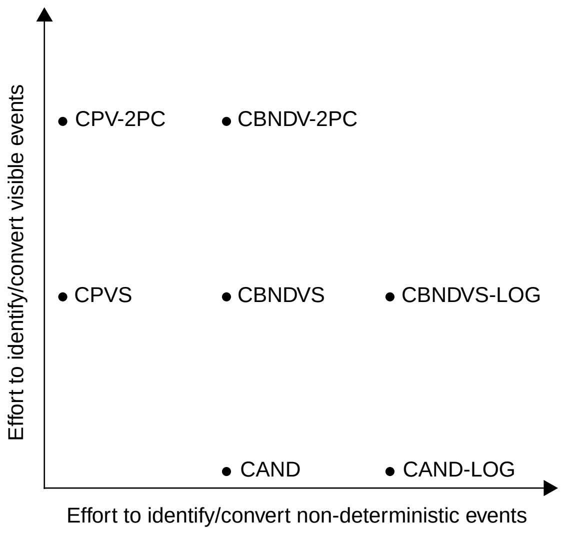

Effort to identify/convert visible events
• CPV-2PC
• CBNDV-2PC
• CPVS
• CBNDVS
• CBNDVS-LOG
• CAND
• CAND-LOG
Effort to identify/convert non-deterministic events

Figure 5: Recovery protocols measured in Section 4.

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 writing the results of `read`, `recv`, `recvfrom`, `recvmsg`, and `select` to a log in order to make message receive and user input events deterministic. After the remaining non-deterministic events (`gettimeofday` and signals), this protocol simply forces a checkpoint.

Commit between non-deterministic event and visible event or send, with logging (CBNDVS-LOG): This protocol is identical to CBNDVS, with the same logging logging used in CAND-LOG employed to convert some non-deterministic events into deterministic events.

Commit prior to visible event with two-phase commit (CPV-2PC): Under this protocol, any time a 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]. Doing so ensures a commit after any non-deterministic events that might causally precede the visible event.

Commit between non-deterministic event and visible event (CBNDV-2PC): In this protocol, if a process wants to execute a visible event that depends on an uncommitted non-deterministic event, it first asks all processes to commit using 2PC. Rather than tracking causal dependencies on non-deterministic events between processes, we implement this protocol using the simple flag-setting approach of CBNDVS by assuming all message receives cause a dependence on a non-deterministic event executed by the sender.

Figure 5 places these recovery protocols in the protocol space developed in Section 3.1.8. 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.

4.3. Applications

Recovery has focused traditionally on recovering long-running, scientific computations. For our study, we will focus instead on recovering interactive applications. We choose these applications because real users are demanding and important customers of recovery, and because recovering these applications is particularly challenging. Interactive 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. there are no message sends (*magic*'s messages to the X server are considered 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.

Applications need only two minor modifications to use Discount Checking and become fully recoverable (implementing recovery in the operating system would require no modifications to the application). First, we insert a call to `dc_init` at the beginning of each program's `main()` and include `dc.h` in this file. Second, we link the application with `libdc.a`, which contains the Discount Checking code, including code to intercept many `libc` functions. Discount Checking transparently inserts checkpoints and logs non-deterministic events as required by each recovery protocol. It also recovers a process when it is restarted after a crash.

4.4. Results

All experiments were conducted on 400 MHz Pentium-II computers each with 128 MB of memory (100 MHz SDRAM). Each machine runs FreeBSD 2.2.7 with Rio and is connected to a 100 Mb/s switched Ethernet. Rio was turned off when using Discount Checking-disk. Each computer has a single IBM Ultrastar DCAS-34330W ultra-wide SCSI 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.

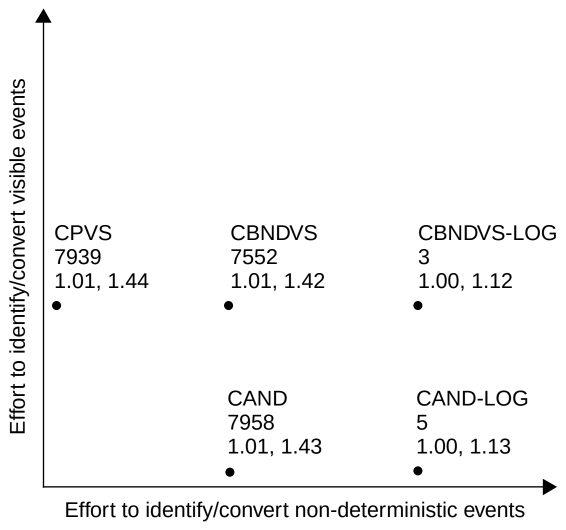
Figures 6 and 7 display the results of running the five applications using both Discount Checking and Discount Checking-disk to implement all seven recovery protocols. Since we use the two-phase commit protocol to coordinate commits between processes in message passing applications, we show the results for CPV-2PC and CBNDV-2PC only for applications that send messages. For each application we show a plot of the protocols in the protocol space described in Section 3.1.8. 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*. Because *xpilot* is a real-time, continuous program, we report its performance as the frame rate it can sustain rather than run-time expansion. Higher frame rates indicate better interactivity, with full speed being 15 frames per second. Checkpoints for *xpilot* are given as the largest number of checkpoints per second among all clients and the server, measured at 15 frames per second.

We can make a number of interesting observations about the results in Figures 6 and 7. As expected, the number of commits generally decreases for protocols that are farther from the origin. As a designer expends more effort to defer commits 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 commits. This increase is because all processes must commit whenever any of them executes a visible event (i.e. sending output to the X server). This increase in commit frequency is greater than the decrease in commit frequency that results from not needing to commit 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. While Discount Checking-disk cannot match this level of performance, it is nonetheless able to provide failure transparency with acceptable overhead for some recovery protocols. 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 than Discount Checking to the number of commits generated by a recovery protocol, because it commits state by writing to disk rather than memory. Overhead for *TreadMarks* is higher than other applications because *TreadMarks* has a large working set and is compute-bound rather than user-bound.

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*.

For *oleo* and *magic*, however, logging does not help appreciably, because there are other sources of non-determinism that are not logged. `gettimeofday` is the major source of non-determinism in *oleo*, and signals are the major source of non-determinism in *magic*. For these applications, it is most helpful to uphold the commit invariant by treating as few events as possible as visible. By so doing, these applications can commit before every visible or send



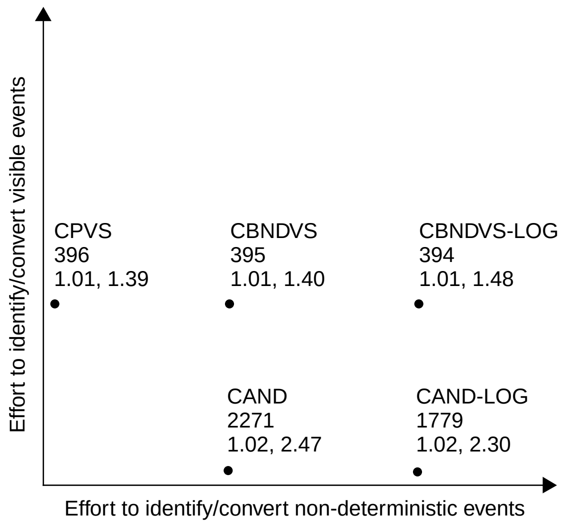
(a) nvi

Key

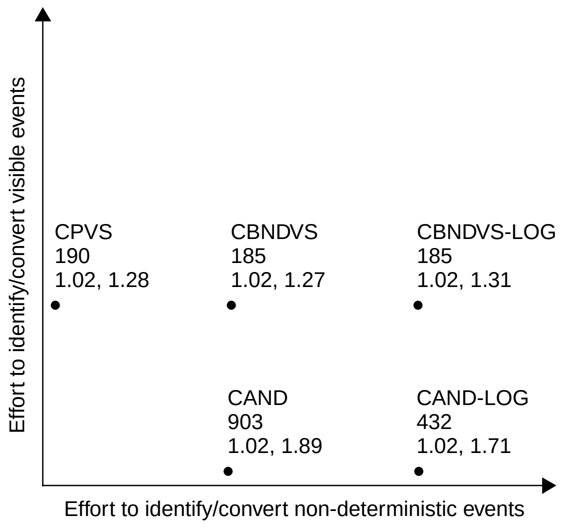
RECOVERY PROTOCOL
# Checkpoints in run
runtime DC, runtime DC-disk

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 user input logged.
CBNDVS-LOG: commit between non-deterministic and visible or send, with user input logged.



(b) oleo



(c) magic

Figure 6: Performance of different recovery protocols for local applications. The first line below each protocol label gives the number of commits in the run. The second line below each protocol label gives run-time ratios (running time of the recoverable program divided by running time of the baseline, non-recoverable application) for two checkpointing systems: Discount Checking and Discount Checking-disk.

event, rather than after each of the more numerous non-deterministic events.

The best recovery protocols for *TreadMarks* are those that use two-phase commit to defer commits beyond the point of sending a message. *TreadMarks* sends messages and executes non-deterministic events (signals) very frequently, which results in a high commit frequency for most protocols. However, *TreadMarks* executes very few visible events (just a handful of writes to the screen), so only a few coordinated commits are needed.

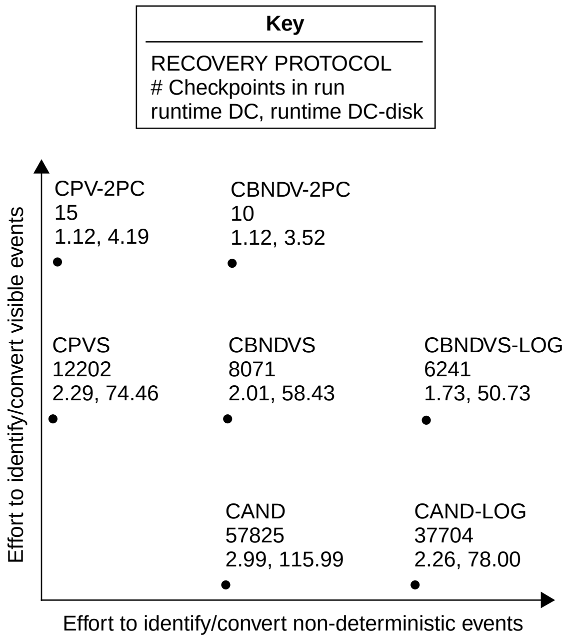
Note that different applications achieve the lowest overhead with different protocols. Thus no one protocol is appropriate for all workloads. In general, protocols that both defer commits and identify and convert non-deterministic events yield more robust performance across a range of applications.

5. Related Work

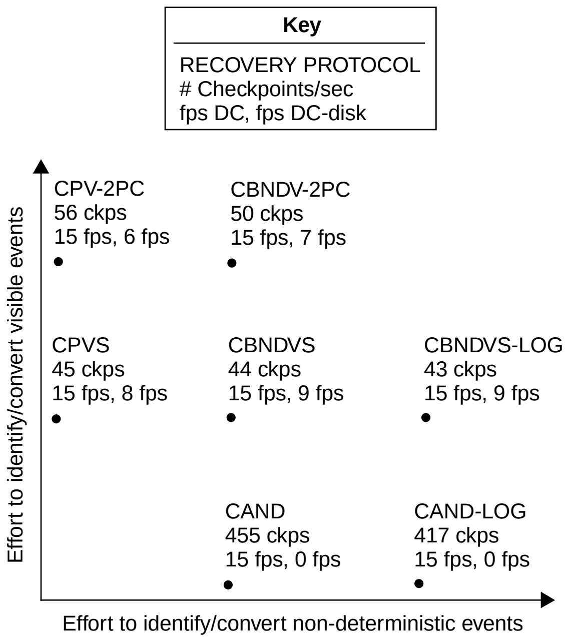
A few researchers have attempted to provide a general view of recovery research.

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.

Alvisi et al. provide a theory of recovery specific to causal logging protocols [Alvisi95]. Their theory is useful primarily in elucidating the relationship between different



(d) TreadMarks Barnes-Hut



(e) xpilot

Recovery Protocols

CPVS: commit prior to visible or send
CAND: commit after non-deterministic
CBNDVS: commit between non-deterministic and visible event or send.
CBNDVS-LOG: commit between non-deterministic and visible or send, with messages and keyboard input logged.
CAND-LOG: commit after non-deterministic 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 7: Performance of different recovery protocols for distributed applications. The first line below each protocol label gives the number of commits in the run. The second line below each protocol label gives run-time ratios (running time of the recoverable program divided by running time of the baseline, non-recoverable application) for two checkpointing systems: Discount Checking and Discount Checking-disk. Because *xpilot* is a real-time, continuous program, we report its performance as the frame rate it can sustain in frames-per-second (fps) rather than run-time expansion. Higher fps values indicate a better rate of interactivity, with full speed being 15 fps. Checkpoints for *xpilot* are given as the highest number of checkpoints per second (ckps) among all clients and the server, measured at 15 fps.

causal logging protocols; it is less useful for other recovery techniques.

A number of researchers have pointed out components of our theory. Our work builds on 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 [Chandy85], one rule defines when an application must preserve such a global state [Strom85, Elnozahy92], and a final rule relates non-determinism to commits [Johnson88, Elnozahy92]. In comparison, we provide a single invariant that captures the exact relationship between the non-deterministic, commit, and visible events at the heart of consistent recovery. Although our single invariant is equivalent to the confluence of the existing rules, viewing recovery through the lens of our invariant has several advantages. First of all, we are able to analyze all existing recovery protocols in light of this single invariant, unifying the various approaches to achieving consistent recovery, and elucidating the relationship between historically unrelated protocols. Second, we are able to expose the existence of a protocol space in which all recovery protocols fall. Third, we 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 must make assumptions equivalent to Assumptions 1 and 2, although these assumptions are usually not stated. Fourth, we are able to explore the implications of our invariant for the fail-stop assumptions made by most recovery systems. Finally, we are able to prove the sufficiency and necessity of our single invariant for guaranteeing consistent recovery, something that, to our knowledge, no researcher has ever done.

Several 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 provide recovery services that allow processes to survive hardware failures.

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.

6. Contributions

This paper makes a number of contributions:

- Provides and proves a theory of consistent recovery that is relevant across all recovery protocols, and shows that upholding a simple invariant is necessary and sufficient to guarantee consistent recovery.
- Unifies the various approaches to consistent recovery and clarifies how they uphold the invariant.
- Suggests a number of new recovery protocols (CPVS, CBNDVS, CBNDVS-LOG, CBNDV-2PC) and exposes the potential for more protocols that both defer commits and convert non-determinism.
- Argues for failure transparency as an OS abstraction and demonstrates its feasibility.

7. Conclusion

Computer users know that system and application failures are all too common. With this paper, we argue that if we have to live with failures, the least the computer could do is try to keep them a secret.

Providing failure transparency as a fundamental abstraction of the operating system has the potential to make computers far more pleasant to use. Doing so involves guaranteeing consistent recovery on behalf of local and distributed applications, which can be tricky business, particularly for complex, distributed applications.

With this paper, 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. Exposing the commit invariant behind recovery protocols enables system designers to think more systematically about what recovery protocol to use for each application.

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, it is even 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.

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.

8. Acknowledgements

We are very grateful to Paul Resnick for his help formulating the proof of our theory. Thanks also to George Dunlap for adding support to the XFree86 Server for our user-level TCP transport.

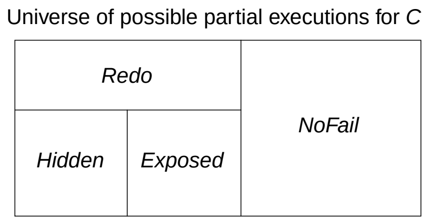


Figure 8: Partitioned execution universe.

9. Appendix

We present here an informal proof of the theorem. For a more formal approach, please see [Lowell99].

Theorem

Recovery of a computation is guaranteed to be consistent if and only if each process that executes a non-deterministic event that causally precedes a visible or commit event *e* later executes a commit event that itself causally precedes *e*.

As mentioned in Section 2, the definition of consistent recovery provides both safety and liveness constraints. We will split the theorem into two components: when event *e* in the theorem is a visible event the theorem guarantees safety, when *e* is a commit event the theorem guarantees liveness. We will prove the necessity and sufficiency of each component separately.

Proof of sufficiency for Safety component. We want to prove that all partial executions of a computation *C* will be equivalent to a failure-free partial execution of *C* despite failures, if each process that executes a non-deterministic event that causally precedes a visible event *e* later executes a commit event that itself causally precedes *e*.

Consider the universe of possible partial executions of *C*, including executions in which it has failed and recovered. We can think of each member of 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 fails. We call this region *NoFail* The region contains the executions of *C* in which at least one process fails. We call this region *Fail*. We can further divide the *Fail* region into two parts. The first part we call *Redo*. It contains the execution 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-failures 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 its pre-failure path by executing a sibling of a pre-failure non-deterministic event. Finally, we can divide *Deviate* into two sub-regions: *Hidden* and *Exposed*. *Exposed* contains those executions where the non-deterministic event at the point of recovery's deviation causally precedes a visible event. *Hidden* contains those where it does not. We show the relevant portions of this partitioning in Figure 8.

With the universe of partial executions thus partitioned, we can make a number of observations. First of all, each execution in *Redo* has an equivalent execution in *NoFail*: the execution from *Redo* is the same as an execution in *NoFail* except that the *Redo* execution re-executes the events between its last commit and failure which is allowed by our equivalence function. Thus all executions in *Redo* have consistent recovery.

Next, observe that each execution E in *Hidden* has an equivalent execution E' in *Redo*. E' is the execution in which the computation failed immediately before executing the deviating non-deterministic event in E, and executed the sibling of that event during recovery. Since the executions in *Redo* have consistent recovery, it follows the executions in *Hidden* do as well.

By process of elimination, it follows that all executions with inconsistent recovery must fall in the *Exposed* region of the universe. Note however that since C commits according to our premise, no execution of C can fall in the *Exposed* region. Thus, the sufficiency of the safety component of the theorem is proved.

Proof of necessity for the Safety component. We intend to prove there exists a partial execution of a computation C that is not equivalent to any failure-free partial execution of C if some process executes a non-deterministic event that causally precedes a visible event e and does not later execute a commit event that itself causally precedes e.

We'll call the non-deterministic event e^{ND} that causally precedes e. We know from Assumption 1 that e^{ND} has a sibling that causally precedes different a visible event e'. From our premise, we know that the process p that executed e^{ND} did not execute a commit event after it. Therefore, if process p fails immediately after the computation executes e, it could execute through the sibling of e^{ND} during recovery, causing e' to be executed. However, we also know from Assumption 1 that e and e' are not both in any legal execution of C. Thus we have described an inconsistent recovery that is possible as a result of C's not committing correctly, and the necessity of the safety component of the theorem is proved.

Proof of sufficiency for the Liveness component. We want to prove that a single failure cannot prevent a process from executing a visible event if each process that executes a non-deterministic event that causally precedes a commit event e later executes a commit event that itself causally precedes e.

We need to show that whenever a process wants to execute a visible event but determines the event would depend on an aborted non-deterministic event, it can abort to its last commit and be assured that all events it executes from that point on will not causally depend on the aborted non-deterministic event. By our premise, we know that no commit causally depends on an uncommitted non-deterministic event. Therefore, any process can always abort back to its last commit to abort a dependency on a lost non-deterministic event (an event that could not have been committed). To ensure that it never regains its dependency on the lost event, the aborting process may have to ask other processes to abort back to their last commits. If sufficient processes are aborted, all processes will permanently lose their dependency on the lost non-deterministic event, since no process's last commit causally depends on it. Thus we have proved the sufficiency of the liveness component of the theorem.

Proof of necessity for the Liveness component. We intend to prove that a single failure can prevent indefinitely some process p from executing a visible event if some process q that executes a non-deterministic event that causally precedes commit event e_p (which is executed by p) never executes a commit event that itself causally precedes e_p.

Imagine process q fails immediately after process p executes event e_p. We know e_p preserves a dependence on process q's now lost non-deterministic event. When process p later attempts to execute a visible event (which it will do by Assumption 2), it will not be able to do so without violating the safety component of the theorem since the visible event would causally follow the lost non-deterministic event. Even worse, it will never be able to abort its dependency on the lost non-deterministic event since its last commit preserves the dependency. Thus we have proved the necessity of the liveness component of our theorem.

Since we have proved the sufficiency and necessity of both the safety and liveness components, we have proved that upholding the theorem is both necessary and sufficient to guarantee consistent recovery under the assumptions we have made. *QED*

10. References

[Alvisi93] 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, June 1993.

[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.

[Baker92] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings USENIX Summer Conference*, pages 31–43, June 1992.

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

[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.

[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.

[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)*, pages 74–83, October 1996.

[Elnozahy92] E. 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.

[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.

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

[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*, 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*, pages 115–132, 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.

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

[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*, pages 92–101, October 1997.

[Lowell98] 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 219–226, June 1998.

[Lowell99] David Ellis Lowell. *Theory and Practice of Failure Transparency*. Ph.D. thesis, University of Michigan. August 1999.

[Plank95] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, pages 213–224, 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*, pages 100–109, October 1983.

[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.

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

[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)*, pages 22–31, June 1995.