# Discount Checking: Transparent, Low-Overhead Recovery for General Applications

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:** Checkpointing is a general technique for recovering applications. Unfortunately, current checkpointing systems add many seconds of overhead per checkpoint. Their high overhead prevents them from making failures transparent to users and other external entities, so failures lose visible state. This paper presents a checkpointing system called Discount Checking that is built on reliable main memory and high-speed transactions. Discount Checking can be used to make general-purpose applications recoverable easily and with low overhead. The checkpoints taken by Discount Checking are extremely fast, ranging for our target applications from 50 µs to a few milliseconds. Discount Checking's low overhead makes it possible to provide ideal failure transparency by checkpointing each externally visible event. Yet even with this high rate of checkpointing, Discount Checking slows real applications down by less than 0.6%.

## 1. Introduction

On today's computer systems, failures such as operating system and process crashes are a fact of life. Persistent state refers to data on a computer system (such as user files) that must survive such failures. Many applications running on a typical computer manipulate persistent state. Examples of such applications include e-mail programs, word processors, spreadsheets, CAD programs, databases, and file systems. These applications must be recoverable; that is, the user must be able to restart the application after a crash without losing persistent state. Most programs enable recovery by adding application-specific code to save and recover user data. This code may be invoked by the user (e.g. an editor's save command) or by the application (e.g. an editor's autosave). This code may be complex and slow, and it usually does not save the complete state of the process. For example, the undo log in most editors is not preserved across program invocations.

This research was supported in part by NSF grant MIP-9521386 and Intel Technology for Education 2000. Peter Chen was also supported by an NSF CAREER Award (MIP-9624869).

Checkpointing offers a general way to recover a process. Checkpointing is a form of backward error recovery that periodically saves the complete state of a running process to stable storage. Checkpointing with rollback recovery is used most often as a fault-recovery technique, but it can be used in other areas as well. Process migration can use checkpointing to move a running process to a new computer [Litzkow92]. Debuggers can use checkpoints to examine the state of a process before a crash. Distributed simulation systems may allow a process to speculatively execute a path rather than wait to synchronize with other processes. If the speculation is incorrect, the system can use checkpointing to recover the process to the point in time before the incorrect speculation [Fujimoto90].

Checkpointing systems strive to be transparent in terms of overhead during failure-free operation, handling of failures, and modifications of the program needed to support recovery.

- **Performance transparency**: The checkpointing system should not increase significantly the time it takes to execute a program, and it should not increase significantly the disk or memory space required to run the program. Users should be unaware of any slowdown during normal operation.
- **Failure transparency**: The checkpointing system should recover the process without losing work that is visible to external entities (such as users). External entities are those whose state can not be undone easily. For example, it is difficult to undo actions caused by sending commands or output to humans, printers, and missile launchers. As far as the outside world can tell, the process should recover back to the same state it was in before the crash. For example, a process should not print a message indicating that a backup was taken, then fail and recover to a point before the backup was taken. In this scenario, a user would be misled if the process executed differently after recovery and did not actually take a backup. To achieve failure transparency, the system must take a checkpoint to ensure that the state from

which the system outputs a message to the user will never be rolled back. This checkpoint is a form of "output commit" [Strom85, Elnozahy93].

- **Programmer transparency**: Since checkpointing is a general recovery technique, it offers the potential for easily making many applications recoverable. A programmer should be able to use the checkpointing system to make a program recoverable with minimal programming effort. It should be much simpler to handle recovery using a checkpointing library than by writing custom recovery code for each new application.

Current checkpointing systems fail to attain one or more of the above three goals for general-purpose programs. In particular, current checkpointing systems incur a high overhead (many seconds) per checkpoint. Their high overhead prevents them from providing failure transparency for interactive applications, because it is infeasible for them to take a checkpoint for every user-visible event.

In this paper, we present a checkpointing system, Discount Checking, that is built on reliable main memory and high-speed transactions. Discount Checking meets all three of the above goals. Overhead per checkpoint for our target applications ranges from 50 µs to 2 milliseconds. This low overhead makes it possible to provide ideal failure transparency by checkpointing each externally visible event. As far as the user and other external entities can tell, a process recovers back to exactly the same state it was in before the crash. Even with this high rate of checkpointing, Discount Checking slows real applications down by less than 0.6%. Furthermore, Discount Checking is easy to use. For most programs, the programmer simply links the program with the Discount Checking library, adds an `#include` file, and adds a call to `dc_init` at the beginning of the program. Discount Checking automatically takes checkpoints and recovers the process during restart.

## 2. Related Work

Recovering a failed process means reconstructing the state of the process, then restarting it. The process may be restarted on the same machine (perhaps after a reboot) or on a different machine (as is done with process pairs) [Bartlett81, Gray86]. There are two main techniques for reconstructing the state of a failed process: checkpointing and log-and-replay [Elnozahy96].

### 2.1. Checkpointing

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 `libckp` 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 (at each user-visible event) to mask failures from users.

Researchers have developed many optimizations to lower the overhead of checkpointing. Incremental checkpointing [Elnozahy92] 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—this guarantees that no visible work is lost during a failure. 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 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.

### 2.2. Log-and-Replay

Log-and-replay is another general-purpose recovery technique [Borg89]. Whereas checkpointing *saves* the state of the failed process, log-and-replay *recomputes* the state of the failed process. Log-and-replay starts from a prior state and rolls the process forward by re-executing the instructions. Re-executing the process must use the same inputs that were used the first time; otherwise the process will not recover back to the same state it was in when it crashed.

These inputs are logged before a crash and used during recovery. Unfortunately, there is a wide variety of inputs that must be logged to recover the process back to the same state, and many of these are difficult to log and replay. In particular, all events that may cause non-deterministic execution must be logged and replayed carefully to ensure repeatability. The following are examples of these events:

- Message-logging systems focus on logging and replaying messages in the original order [Strom85, Koo87, Johnson87, Borg89, Lomet98]. Input from the user can be considered a form of messages.
- Signals and other asynchronous events are difficult to log and replay, because the effect of these events may depend on the exact processor cycle the process received the signal [Slye96]. For this reason, Targon/32 chose to checkpoint before each signal rather than log [Borg89]. In general, timing dependencies are a difficult input to log and replay.
- Thread scheduling events must be logged and replayed in the same order to ensure repeatability during the recovery of multi-threaded applications. Multi-threaded applications may also need to log shared-memory accesses between cooperating threads.
- The results of many system calls must also be logged and replayed [Elnozahy93, Russinovich93]. For example, the application could execute code based on the time of day returned by a system call. During recovery, the system call must return the same time of day to enable to process to execute the same code.

With sufficient effort, many of these inputs can be logged and replayed [Elnozahy93]. As evident from the above list, however, it is no simple matter to track down, log, and replay repeatably all inputs that affect the roll-forward phase. For example, [Slye96] required a custom thread library and object-code instrumentation to successfully track thread scheduling events and signals. The complexity of dealing with these and other sources of non-determinism has prevented the widespread use of log-and-replay in recovering general applications [Birman96, Huang95].

### 2.3. Comparison of Recovery Techniques

Checkpointing is a more general recovery technique than log-and-replay, because checkpointing saves the crashed state and so obviates the need for reconstructing state using repeatable re-execution. In other words, checkpointing works for non-deterministic processes, whereas log-and-replay must turn non-deterministic processes into deterministic ones. The main motive for using log-and-replay instead of checkpointing is its speed for output commit. Logging inputs is faster than current checkpointing systems, unless the checkpoint interval is very long [Elnozahy94].

In summary, prior work has provided two general-purpose recovery techniques: checkpointing and log-and-replay. Prior checkpointing systems add many seconds of overhead per checkpoint, which prevents them from providing failure transparency for general applications. Log-and-replay offers good failure transparency by reconstructing the state of a failed process to the exact point of the crash. However, it is very difficult to use log-and-replay for general, non-deterministic programs.

The next section describes the design and implementation of a fast checkpointing library. Discount Checking provides fast, synchronous checkpoints, allowing checkpointing to provide complete failure transparency for general, non-deterministic programs.

## 3. Design and Implementation of Discount Checking

### 3.1. Reliable Main Memory and Fast Transactions

The key to fast checkpointing is reliable main memory and fast transactions. Reliable main memory is a form of fast, stable storage that can be mapped directly into a process's address space. In our project, we use the reliable main memory provided by the Rio file cache [Chen96] and the fast transactions provided by the Vista transaction library [Lowell97].

Like most file caches, Rio caches recently used file data in main memory to speed up future accesses. Rio seeks to protect this area of memory from its two common modes of failure: power loss and system crashes. While systems can protect against power loss in a straightforward manner (by using a $100 uninterruptible power supply, for example), protecting against software errors is trickier. 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 scheme does not affect performance significantly. After a crash, Rio writes the file cache data in memory to disk, a process called warm reboot. In essence, Rio makes the memory in the file cache persistent. Chen et al. verified experimentally that the Rio file cache was as safe as a disk from operating system crashes. The version of Rio used in this paper is a modification of FreeBSD 2.2.7. FreeBSD-Rio runs on standard PCs without modifications to the hardware, firmware, or processor configuration.

Vista builds on the persistent memory provided by Rio to provide fast transactions [Lowell97]. Applications use Vista to allocate areas of persistent memory and perform atomic, durable transactions on that memory. Vista uses several optimizations to lower transaction overhead. First, all data is stored or logged in Rio's reliable memory, thus eliminating all disk I/O for working sets that fit in main memory. Second, Vista uses a "force" policy [Haerder83] to update the transactional memory eagerly, thus eliminating the redo log and its associated complexity. Third, Vista maps the transactional memory directly into the address space. This style of mapping eliminates all systems calls and all-but-one memory-to-memory copy, while not hurting reliability [Ng97]. Fourth, Vista's simplicity and small code size (700

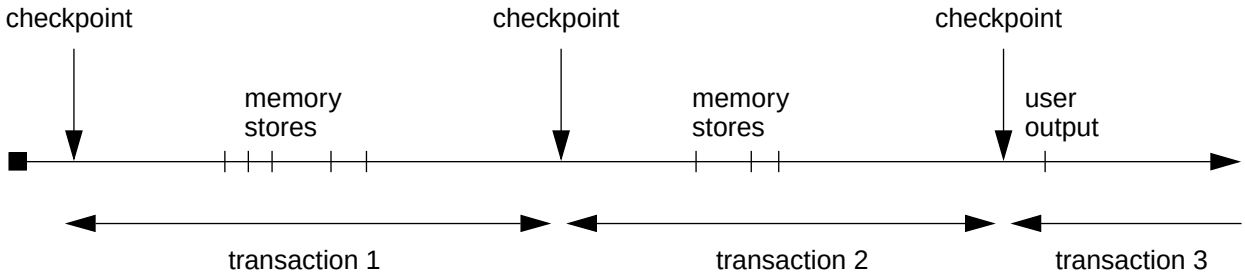



**Figure 1: Equivalence of Checkpointing and Transactions.** Transactions and checkpointing are very similar concepts. This figure shows a process taking checkpoints as it executes. The interval between checkpoints is equivalent to the body of a transaction. Taking a checkpoint is equivalent to committing the current transaction. After a crash, the state of the process is rolled back to the last checkpoint; this is equivalent to aborting the current transaction. A checkpoint is taken before user output to ensure the process recovers to a state consistent with that seen by external, non-abortable entities.

lines of C) lead to very short code paths. As a result of these optimizations, Vista's transactions are extremely fast: small transactions can complete in under 2 μs.

### 3.2. Transactions and Checkpointing

Although they are rarely discussed together in the literature, transactions and application checkpointing are very similar concepts. Figure 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. After a crash, the state of the process is rolled back to the last checkpoint; this is 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.

### 3.3. Saving Process State

Building a checkpointing system on a transaction system is conceptually quite simple: map the process state into the transactional memory and insert transaction_begin and transaction_end calls to make the interval between checkpoints atomic. Discount Checking is a library that can be linked with the application to perform these functions. There are three main types of process state that must be saved in the transactional memory: 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 transactional 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 `mmap`. To minimize the number of Vista segments, Discount Checking moves the stack into a static 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 at each checkpoint. To rollback after a process crash, Discount Checking must undo the memory modifications made during the current 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 transactional memory. However, a process's state also includes register contents, which can not be mapped into memory. Discount Checking copies the register contents (stack pointer, program counter, general-purpose registers) 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 general processes recoverable requires saving miscellaneous state stored in the kernel. Discount Checking saves the pieces of this state that are required most often. We occasionally need to add other pieces as we use Discount Checking for new applications. Our basic strategy for saving these pieces of state is to intercept system calls that manipulate the state, save the values in Vista's memory, and restore the state during recovery. 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 and re-positions these files. Discount Checking also inter-

cepts 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 the transactional 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, and uses this address when sending messages.

**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 re-installs 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).

### 3.4. Failure Transparency

In general, providing complete failure transparency requires a checkpoint just before executing a non-abortable event. An event that is visible to an external entity (such as printing to the screen) is an example of a non-abortable event. Taking a checkpoint right before such an event guarantees that the event is not forgotten in a crash. From the point of view of an external entity (such as people), the recovered process returns to the same state it was in before the crash. In the worst case, the recovered process will re-execute the non-abortable event. This duplicate event is often harmless. For example, applications must already cope with duplicate messages when using today's unreliable networks. The duplicate event can be eliminated if the event is testable or can be made atomic with the checkpoint [Gray93]. Otherwise the probability of the event being duplicated may be minimized by taking another checkpoint right after the event.

Without the checkpoint before a non-abortable event, the process might (1) execute the non-abortable event, (2) crash and recover to an earlier state, (3) take a different execution path due to non-determinism in the program (Section 2.2), then (4) not re-execute the non-abortable event. Under this scenario, the process would recover to a state that is inconsistent with the state seen by external, non-abortable entities.

By taking a checkpoint before each non-abortable event, Discount Checking guarantees failure transparency for deterministic and non-deterministic programs. It is far easier to identify non-abortable events, such as printing to the screen, than to identify and make repeatable all inputs and non-deterministic events, as is required by log-and-replay (Section 2.2).

One aspect of designing Discount Checking is classifying events as abortable or non-abortable. Events such as printing to the screen must be classified as non-abortable, as we know of no easy way to abort a user's memory. Other events may be considered abortable or non-abortable. For example, we could consider writing to a file a non-abortable event and preserve failure transparency by taking a checkpoint before each file write. However, a faster way to preserve failure transparency when files are not shared concurrently is to make file writes abortable with an undo log (Section 3.3) and eliminate the checkpoint.

Sending a message to a non-abortable entity (such as a display server) must be considered a non-abortable event and hence must induce a checkpoint. Taking a checkpoint for each message send guarantees consistent, distributed recovery [Lamport78, Koo87]. If the receiver's state can be rolled back, we can consider message sends abortable events. This requires an atomic commitment protocol (such as two-phase commit) between the sender and receiver.

### 3.5. 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 5.1, 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 is copy-on-write. Instead of copying the address space at a checkpoint, Vista uses copy-on-write to lazily copy only those pages that are modified during the ensuing interval. Copy-on-write is implemented using the virtual memory's write protection. On some sys-

tems, system calls fail when asked to store information in a protected page. Discount Checking intercepts these system calls and pre-faults 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.

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 (1) the signal stack contains data only when the checkpoint occurs in a signal handler, and (2) 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.

### 3.6. Vista-Specific Issues

Discount Checking benefits substantially by building on Vista's transactional 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 suddenly change when the undo log was played back during Vista's recovery procedure. Instead, we want Vista to manually recover its own variables, as it does when not running with Discount Checking. One way to view this is that a recovery system may recover client variables automatically, but it cannot use itself to recover its own variables. To fix the 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.

## 4. Using Discount Checking

Checkpointing libraries provide a substrate for easily 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 transactional memory, moves the stack, and starts the first checkpoint interval. `dc_init` 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.

As discussed in Section 3.4, Discount Checking inserts checkpoints automatically before non-abortable events such as printing to the screen. Discount Checking also restarts with little user intervention: the user simply re-invokes the program. When `dc_init` is called, it notices that there is an existing checkpoint file and initiates recovery. The recovery procedure restores the registers, memory, and kernel state at the time of the last checkpoint, then resumes execution from the last checkpoint. From the user's point of view, the program has simply paused—no visible state is lost during a crash. 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.

| 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 1: Experimental Platform.**

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. Performance Evaluation

Our goal is to use checkpointing to recover general applications with complete failure transparency. The feasibility of this goal hinges on the speed of Discount Checking, particularly during failure-free execution. In this section, we measure the overhead added by Discount Checking for a variety of applications.

We first use a microbenchmark to quantify the relationship between checkpoint overhead and working set size. We then describe our suite of test applications and the overhead needed to make them recoverable. Last, we explore the relationship between checkpoint interval and overhead for two programs from SPEC CINT95.

Table 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.

### 5.1. Microbenchmark

Copying memory pages to Vista's undo log comprises the dominant overhead of checkpointing. Figure 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.

The 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.

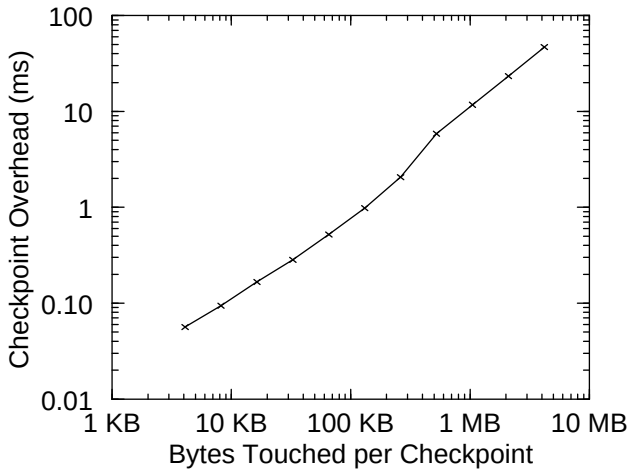



**Figure 2**: **Checkpoint Overhead for Microbenchmark.** Checkpoint overhead is proportional to the number of pages touched per checkpoint. Each 4 KB page takes approximately 45 µs to handle the write-protection signal and copy the page to Vista's in-memory undo log (32 µs for working sets that fit in the L2 cache).

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. Prior checkpointing libraries incur many seconds of overhead per checkpoint, so Discount Checking is thousands of times faster than these libraries even for programs that touch a large amount of data per interval.

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.

### 5.2. Applications

We use four benchmarks to measure how Discount Checking affects performance on real applications: `vi`, `magic`, `oleo`, and `rogue`. `vi` is one of the earlier text editors for Unix. The version we use is `nvi`. `magic` is a VLSI layout editor. `oleo` is a spreadsheet program. `rogue` is a game that simulates an adventure through a dungeon.

`vi`, `oleo`, and `rogue` are full-screen text programs that manipulate the terminal state through the `curses` library. To recover the terminal state after a crash, we spec-

| Program | | Running Time | | | # of Checkpoints | Size of Undo Log | |
|---|---|---|---|---|---|---|---|
| Name | Type | Original | Discount Checking | Overhead | | Average | Max |
| vi | editor | 881.90 sec | 882.76 sec | **0.1%** | 7940 | 74 KB | 144 KB |
| magic | CAD | 89.54 sec | 90.04 sec | **0.6%** | 208 | 126 KB | 734 KB |
| oleo | spreadsheet | 57.91 sec | 58.02 sec | **0.2%** | 396 | 94 KB | 123 KB |
| rogue | game | 16.51 sec | 16.58 sec | **0.4%** | 231 | 49 KB | 82 KB |

**Table 2: Application Overhead.** Discount Checking adds very little run-time or memory overhead in making programs recoverable, yet it provides completes failure transparency by not losing any visible state after a crash.

ify a user recovery function when calling `dc_init`. The recovery function is six lines of `curses` function calls. These six lines and the call to `dc_init` are the only changes required to make `vi`, `oleo`, and `rogue` recoverable with Discount Checking. `magic` is an X11-based application and communicates with the X server using messages. To provide failure transparency, Discount Checking intercepts message sends and checkpoints before communicating with the X server. Other than the call to `dc_init`, no changes were needed in `magic` to use Discount Checking.

Conducting performance measurements of interactive programs requires some care to achieve repeatable results. We make the runs repeatable by playing back input from a log file instead of the terminal. For `vi`, `oleo`, and `rogue`, we simulate a very fast typist by delaying 100 ms each time the programs asks for a character. For `magic`, we delay 1 second between mouse-generated commands. We use this type of delay instead of fixing the interarrival time between characters because it more faithfully simulates real interaction; typists often wait for the output from the last keystroke before typing the next keystroke. This type of delay also presents a pessimistic view of checkpointing overhead because checkpoint overhead is never overlapped with the delay. Fixing the interarrival time between characters would allow some of the checkpoint overhead (and other computation) to be hidden in the 100 ms interarrival time. Note that we measure performance using an input rate much faster than people can interact with the program. With real human interaction, the relative overhead added by Discount Checking would be even lower.

For `vi`, we replay the keystrokes used when writing the introduction to this paper. For `magic`, we replay the commands used to layout a simple inverter. For `oleo`, we replay the keystrokes used to create the budget for a grant proposal. For `rogue`, we replay the keystrokes used to navigate through one level of the dungeon.

Table 2 shows the overhead added by Discount Checking in terms of running time and memory usage. Discount Checking adds negligible run-time overhead for these applications (at most 0.6%). As expected, the bottleneck for interactive applications is the user. Even with interactive input rates, however, prior checkpointing libraries would have slowed these applications by a factor of 10-100 to achieve complete failure transparency. A unique strength of Discount Checking is its ability to recover general-purpose, interactive applications—this is one of the target classes of applications for our work.

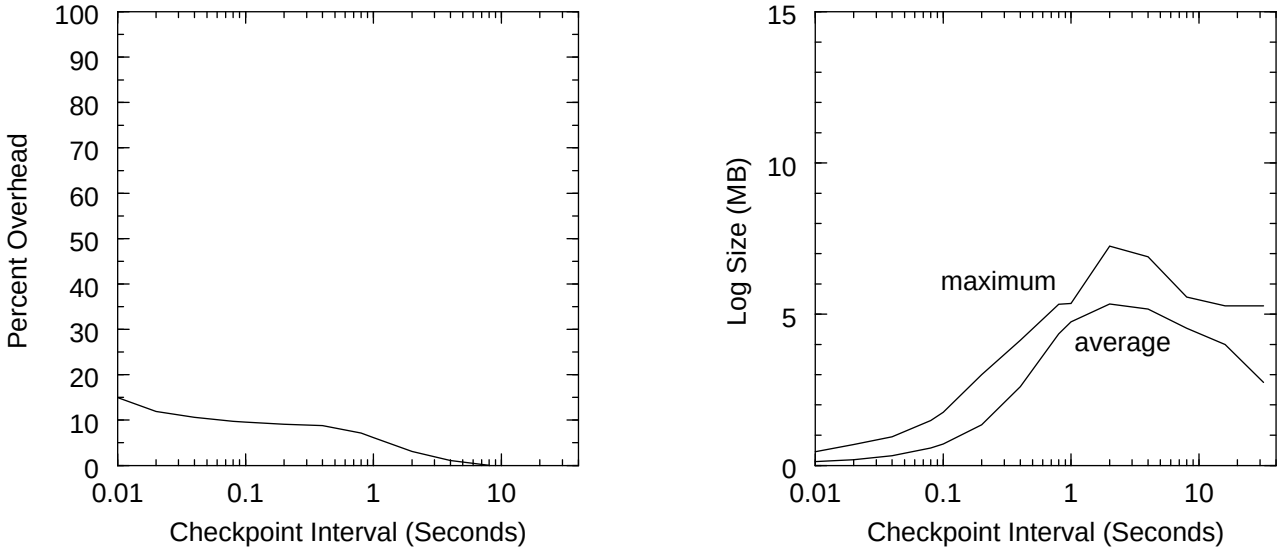
Discount Checking uses extra memory to store the undo log. Table 2 shows the average and maximum size used by the undo log, where average refers to the average size of the undo log at the end of a checkpoint interval. The extra memory used by Discount Checking for these applications is usually only 100 KB.

Besides adding little overhead to failure-free operation, Discount Checking also recovers very quickly. Because there is no roll-forward during recovery, Discount Checking is able to recover these programs in a fraction of a second.

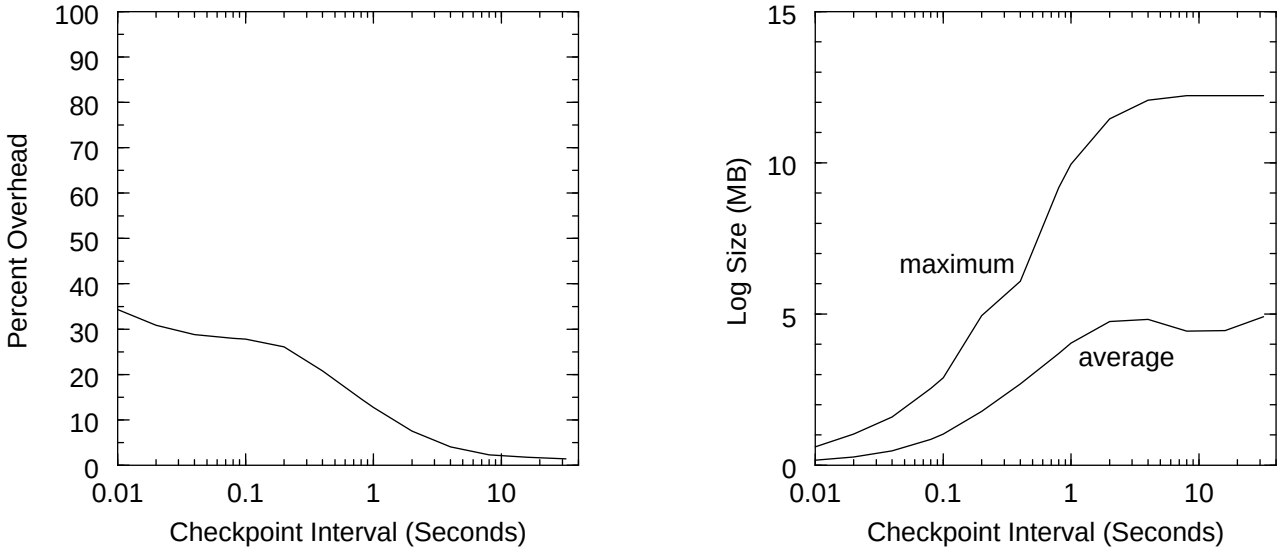
### 5.3. Varying the Checkpoint Interval

We next measure how Discount Checking performs for different checkpoint intervals. The benchmarks we use are non-interactive computations from SPEC CINT95. 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 these recoverable with Discount Checking was the call to `dc_init` (and the accompanying `#include`).

Figure 3 graphs the relative overhead incurred as a function of checkpoint interval. For these programs, few checkpoints are required for failure transparency because they produce very little output. Instead, 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.




(a) ijpeg




(b) m88ksim

**Figure 3: Checkpoint Overhead for Varying Intervals.** We use `ijpeg` and `m88ksim` to measure how the interval of checkpointing affects 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.

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 checkpoints is not needed for these applications, but it serves to demonstrate the speed of Discount Checking on a wider range of applications. Other SPEC benchmarks gave similar results, with overhead ranging from 0-10% with a 1 second checkpoint interval.

Figure 3 also shows how the average and maximum size of the undo log varies as a function of checkpoint inter-

val. 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 intervals. This occurs because of 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.

## 6. Contributions

This paper makes a number of contributions to recovery research.

First, we show how to build a fast checkpointing system from reliable memory and fast transactions. Discount Checking's checkpoints typically take between 50 µs and a few milliseconds to complete, much less time than the many seconds of overhead traditional checkpoints incur.

Micro and millisecond checkpoints are important not just because they speed up recoverable applications. Fast checkpoints enable recovery techniques that would be impractical with classical checkpoints. For example, with fast checkpoints it becomes feasible to recover distributed systems by taking a checkpoint before every message send.

Fast checkpoints also expand the domain of applications that can use checkpointing. For example, as we show in this paper, fast checkpoints can be used to make interactive applications transparently recoverable with low overhead. Programs with lots of human interaction are some of the most deserving of strong recovery properties, as human labor is difficult and painful to rebuild. Unfortunately, traditional checkpoints have been too slow to provide failure transparency for interactive applications.

Another of our contributions is to focus this research on checkpointing general applications, for which numerous system calls and plentiful kernel state are the norm. In contrast, classical checkpointing research has focused on taking checkpoints in scientific compuations. Scientific computations are concerned mainly with computing mathematical results and as such do few system calls, and have minimal kernel state. As a result, checkpointing systems targeted for such applications need not be very general.

Furthermore, we show it is possible to duplicate sufficient kernel state outside the kernel to enable full process checkpointing at user level—even for applications that execute a wide variety of system calls.

Finally, we illustrate an important use for reliable memory and fast transactions: recovering process state. Process state is a class of data that fits in memory, benefits from being made recoverable, and for which traditional stable storage and transactions are too slow.

## 7. Future Work

We are exploring more fully how fast checkpoints affect distributed recovery. For example, fast checkpoints remove the main bottleneck (writing to stable storage) in algorithms used in distributed transactions and coordinated checkpointing, such as two-phase commit [Gray78]. As mentioned in section 6, fast checkpoints also make practical new algorithms in distributed recovery. For example, taking a checkpoint before sending each message guarantees globally consistent recovery without sending extra messages [Koo87]. With fast checkpoints, these schemes provide low-overhead recovery for general-purpose, distributed applications.

We are also considering ways to recover from bugs that violate the fail-stop model [Schneider84, Chandra98, Chandra99]. Programs with such a bug may run for a long time after the bug is activated. We know of no current recovery system that can recover from such a bug, because the corruption caused by the bug may be preserved in the recovery data (checkpoint or log). One way to recover from these bugs is to keep a number of past checkpoints and roll back more than one checkpoint until before the bug was activated [Wang93]. Most bugs in production systems are triggered by non-deterministic events (so-called Heisenbugs) [Gray86] and may not occur again after recovery.

## 8. Conclusions

This paper has presented a checkpointing system that is built on reliable main memory and high-speed transactions. Discount Checking can be used to make general-purpose applications recoverable easily and with low overhead. The checkpoints taken by Discount Checking are extremely fast, taking between 50 µs and 2 milliseconds for our target applications.

Discount checking strives to be transparent in three important ways. Its performance is transparent—users are unaware of a 0.6% performance degradation. It makes failures transparent—every single update to the user's display is recovered. Finally, it is transparent to the programmer—applications need only two small source code modifications to be made recoverable.

## 9. Software Availability

Source code for Discount Checking, Vista, and the FreeBSD version of Rio will be made available at `http://www.eecs.umich.edu/Rio`.

## 10. Acknowledgments

We owe thanks to George Dunlap for adding support for our user-level TCP protocol to the X Windows library. The other members of the Rio team (Wee Teck Ng and Subhachandra Chandra) contributed in discussions and debugging sessions during the design and implementation of Discount Checking.

## 11. References


[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.

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

[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.

[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.

[Chandra99] Subhachandra Chandra and Peter M. Chen. Recovery in the Presence of Fail-Stop Violations. Submitted to the *1999 Symposium on Fault-Tolerant Computing (FTCS)*. June 1999.

[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.

[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.

[Elnozahy92] E. N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The Performance of Consistent Checkpointing. In *Proceedings of the 1992 Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[Elnozahy93] E. N. Elnozahy. Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication. Technical Report TR93-212, Rice University, October 1993. Ph.D. thesis.

[Elnozahy94] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proceedings of the 1994 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 298–307, June 1994.

[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.

[Fujimoto90] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 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.

[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.

[Huang95] Yennun Huang and Yi-Min Wang. Why Optimistic Message Logging Has Not Been Used in Telecommunications Systems. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 459–463, June 1995.

[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.

[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.

[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.

[Litzkow92] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration outside the Unix Kernel. In *Proceedings of the Winter 1992 USENIX Conference*, January 1992.

[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.

[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.

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

[Russinovich93] Mark Russinovich, Zary Segall, and Daniel P. Siewiorek. Application Transparent Fault Management in Fault Tolerant Mach. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pages 10–19, June 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.

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

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

[Wang93] Yi-Min Wang, Yennun Huang, and W. Kent Fuchs. Progressive Retry for Software Error Recovery in Distributed Systems. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS)*, 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.