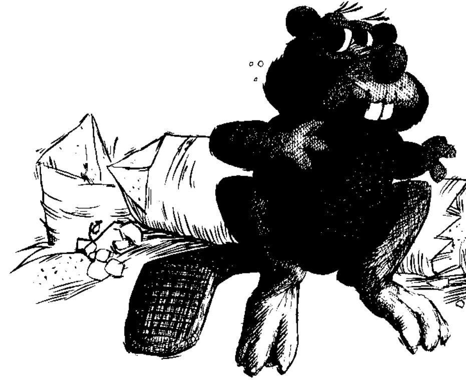
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Backward Error Recovery in a UNIX Environment

D.J. Taylor
M.L. Wright

CS-85-54

December, 1985

Backward Error Recovery in a UNIX Environment

David J. Taylor

Michael L. Wright

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Performing backward recovery efficiently usually requires special hardware, for main storage data, and facilities in the operating system, for data on disk. This paper describes how we have used facilities in UNIX, and in a simple file system built on top of the UNIX file system, to achieve reasonably efficient backward error recovery. The implementation of recovery is simplified because our simple file system is record-oriented, in contrast to the unformatted stream of bytes making up a UNIX file. The system is not intended for production use, but is sufficiently general and efficient to support experimentation with backward recovery. Because the facility is intended to support experimentation, it is quite flexible: in particular, it supports both inclusive and disjoint recovery. As well, a program using the system can specify its own routines to perform disjoint recovery.

1. Introduction

In a fault tolerant system, when an error is detected, continued operation requires that the existing erroneous state be replaced by an error-free state. There are two basic possibilities, forward and backward error recovery [1]. In the case of forward error recovery, redundancy within the state is used to correct errors. In the case of backward error recovery, the current state is replaced by a previous, presumably correct state.

Forward error recovery can be efficient, but must rely on detailed knowledge of the system state. Backward error recovery has the important advantage that the mechanism can be completely independent of the organisation of data used by the system. Unfortunately, the overhead associated with backward recovery can be large. Thus, many implementations of, and proposals for, backward recovery use special hardware to reduce overhead. The use of special hardware limits the applicability of the techniques, and makes it difficult to experiment with backward recovery without a very significant investment.

This paper describes our attempt to implement backward recovery facilities in a UNIX* environment, without the use of special hardware. The efficiency is

*UNIX is a trademark of Bell Telephone Laboratories.

limited by the lack of such special hardware, but we have achieved a system which is sufficiently efficient for experimentation. It was not intended to construct a system which would be useful in a production environment, although the efficiency in some cases may be appropriate even for such use.

The recovery mechanism we have implemented is quite general, but it is our intention to use it in the context of empirical study of robust storage structures. Thus, the mechanism was implemented as part of the IOSYS file system [7] which we have been using to support such experimentation. A particular advantage is that IOSYS files contain fixed-length records rather than the unformatted stream of bytes found in a UNIX file. The file recovery techniques described here are oriented to IOSYS files. We believe that an efficient recovery mechanism for UNIX files would be much more difficult.

We follow the terminology for backward recovery used in [1]. A *recovery point* is a system state which can be returned to by backward recovery. It is conceptually an instant at which the system state was recorded for possible future use. Making the current state a recovery point is referred to as *establishing* a recovery point. Returning to an established recovery point is referred to as *restoring* the recovery point. Disposing of the information associated with a recovery point, so that it may no longer be restored, is referred to as *discarding* the recovery point. A recovery point is said to be *active* from the time it is established until the time it is discarded. The interval during which a recovery point is active is a *recovery region.*

In many cases, recovery regions must be properly nested. For example, this is the case if backward recovery is being used to support a recovery block facility [3]. Since our objective was to build a general tool for experimentation, it was not appropriate to include such restrictions. Thus, we allow any active recovery point to be restored or discarded. If a recovery point other than the most recent active recovery point is restored, it does not make sense to retain more recent recovery points, so they are implicitly discarded. The implementation discards recovery points most efficiently if recovery regions are properly nested, but no other restrictions exist, except implementation restrictions which place an upper limit on the number of simultaneously active recovery points.

In the next section, the interface to the recovery mechanism is described. The following section contains a brief description of the implementation. Then, we present the results of some experience in using the recovery mechanism, including performance characteristics. The last section provides a summary and conclusions, together with some possibilities for future work.

2. User Interface

The central core of the user interface consists of functions for establishing, restoring, and discarding recovery points. Some additional complexity exists because the mechanism is intended to offer various alternatives for experimentation.

The major option offered is inclusive versus disjoint recovery for main storage. Without special hardware, checkpointing is the only reasonably efficient, general mechanism for recovery of main storage. This is expensive, so an alternative is offered: a program may provide specialised facilities for recovery of a particular main storage object. If all relevant objects have such recovery facilities, recovery may be performed on an object-by-object basis. Following the terminology of [1], the checkpoint alternative is referred to as *inclusive* recovery and the object-by-object alternative as *disjoint* recovery.

The user thus needs the ability to select inclusive or disjoint recovery, and if disjoint recovery is selected, the ability to specify functions to perform recovery on a main storage object. To specify type of recovery, a program simply invokes _Recov_type(INCLUSIVE) or _Recov_type(DISJOINT). Specifying disjoint recovery for main storage objects is described at the end of this section.

To establish a recovery point, a program invokes

id = _Establish(&restore_count);

The value returned, assuming success, is a recovery point identifier, and *restore_count* is set to zero. As explained in the next section, when using the inclusive mechanism, the restore function does not return, instead return is from the _Establish which established the recovery point. In this case, the recovery point identifier is returned as usual, and *restore_count* is set to indicate the number of restores which have taken place. Thus, when implementing a recovery block, the *restore_count* may be used to select the appropriate primary or alternate to be executed next.

To discard a recovery point, a program simply invokes

ret = _Discard(id);

where *id* identifies an active recovery point. To restore a recovery point, a program invokes

ret = _Restore(id, flag);

where *id* identifies an active recovery point. (In each case the return value is simply an error/no error indicator.) The parameter *flag* is either KEEP or DISCARD, and indicates whether the recovery point is to be retained after it is restored. Since _Restore(id, DISCARD) is equivalent to _Restore(id, KEEP) followed by _Discard(id), the *flag* is not essential for proper functionality. Unfortunately, in some cases the inefficiency of separate restore and discard operations is too great to be tolerated, so this small complication was felt to be essential.

To specify disjoint recovery for a main storage object, a program invokes

_Recovery_object(object, establish, restore, discard, flag);

The parameter *object* identifies the object in some way, and is passed to the various functions when they are invoked. The parameters *establish, restore,* and *discard,* are the object-specific functions for the recovery mechanism. The parameter *flag* is not strictly necessary, but is used to improve performance. When a restore to other than the most recent recovery point is performed, it may be more efficient to use the restore function than the discard function, to remove intervening recovery points. The *flag* simply indicates which should be used: each should provide the same result, but not necessarily as efficiently.

The object establish function is passed *object* and the identifier of the recovery point being established. It returns a value which is later passed to the restore and discard functions, along with *object* and the recovery point identifier. In addition, the restore function is passed a KEEP/DISCARD flag, and the discard function is passed the identifier of the recovery point preceding the one being discarded.

The above describes the interface as provided to a programmer writing a program which uses IOSYS. In addition, our software for experimentation with data structures (the Interchangeable Storage Structure System [7]) provides an interface to the terminal user for _Recov_type, _Establish, _Restore, and _Discard. This allows the terminal user (or a prepared script of terminal commands) to use the recovery facilities instead of, or in addition to, the usage generated by the program itself.

3. Implementation

This section first describes the implementation of the inclusive recovery mechanism for main storage, then the recovery mechanism for IOSYS files, and finally, the use of the object recovery mechanism for two main storage objects: mangle tables and bit vectors. Specific motivation for some design decisions is given below. In general, we wanted to implement a simple mechanism which would not slow execution significantly. We were less concerned with main storage overhead, although we tried to minimise such overhead, to the extent that it did not significantly conflict with the other two objectives.

3.1. Inclusive recovery

It is fortunate that UNIX provides a kernel primitive which takes a complete checkpoint of main storage. The primitive is *fork* and was intended for concurrency rather than checkpointing. It has the property that after successful execution, the invoking process has been duplicated (except that read-only code is shared): the two processes can be distinguished only by the value returned by *fork*.

Thus, to checkpoint the contents of main storage it is only necessary to invoke *fork* and have the new process immediately stop itself. To discard a recovery point, the stopped process is killed. To restore a recovery point, the stopped process is started and the currently executing process terminates itself. If the recovery point is to be retained, immediately after the stopped process is

restarted it uses *fork* to make another copy, which also stops itself. Since processes stop in a routine invoked by _Establish, a restore causes _Establish to return, rather than _Restore.

This provides a simple, reasonably efficient mechanism for checkpointing main storage. Unfortunately, many details need to be handled carefully. (Implementation details described here pertain to 4.2BSD UNIX. Some things would have to be done differently in other versions of UNIX.)

One problem is that the shell (command interpreter) believes that termination of the process it originally created means that all activity of the program has ceased. Thus, if behaviour is exactly as described above, as soon as a restore takes place, the shell will decide that the program has terminated and attempt to resume reading input from the terminal. To avoid this, it is necessary to keep the original process in existence until the entire program terminates. The solution adopted is elegant, but relies on another UNIX facility, pipes. A pipe is created for which the original process holds the read descriptor and all other processes hold a write descriptor. After a restore, the original process attempts to read from the pipe. This read blocks as long as any write descriptor for the pipe exists. When the program terminates, and hence all the other processes terminate, the UNIX kernel discovers that the pipe cannot be written to, and causes the pipe read to terminate with an error. This unblocks the original process and indicates that it can now safely terminate. The pipe used here has the rather unusual property that it is created with the intention of never writing any data in it, but simply using the existence of write descriptors for it to indicate the existence of a certain set of processes.

Another problem is that, in UNIX, a process does not completely go away until its parent has obtained status information about its termination. Thus, a discard operation could leave a terminated process in the system. (For a restore, the parent of the terminating process has already terminated, so no problem occurs.) If such processes are allowed to build up, the user's process limit will eventually be reached and it will not be possible to establish new recovery points. Thus, whenever a discard is performed, it is necessary to check whether the process just killed was a child of the process performing the kill, and if so, immediately ask the UNIX kernel for its status, so that the process will be completely removed from the system. If the process being killed is not a child of the process performing the kill, the parent has already exited, so no problem occurs.

The final problem we describe here is that the kernel insists on starting stopped processes when the parent process exits. Thus, when a stopped process is restarted it must determine whether it was started by the kernel, in which case it should immediately stop itself again, or started as part of a restore operation, in which case it should continue executing.

As an example, suppose that three recovery points have been established. There will be four processes, as shown in Figure 1(a), where each process is labelled with its recovery point identifier, the original process being labelled "0" since it is associated with no recovery point. The currently executing process (0) is marked with an asterisk. If recovery point 2 is restored, with the KEEP option, the situation becomes as in Figure 1(b). Process 3 has been discarded,

process 2 is now executing, and a new process (2a) has been created to continue holding the system state at recovery point 2. Process 0 is no longer executing, but continues to exist so that the shell will not resume execution. If recovery point 2 is restored again, but now with the DISCARD option, the situation becomes as in Figure 1(c). Process 2 has terminated, and process 2a is executing. There is no longer a parent-child connection between process 0 and the executing process, but a pipe connects processes 1 and 2a to process 0, so it continues to wait for all other processes to terminate.

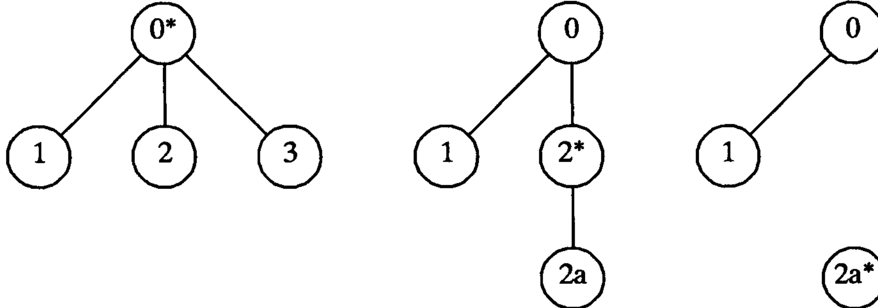


Figure 1(a) Figure 1(b) Figure 1(c)
Processes used for inclusive recovery.

3.2. Recovery for files

For efficient execution of I/O-intensive experiments, IOSYS allows small files to be kept in main storage, thus file recovery must handle two types of files, those on disk and those in main storage. If inclusive recovery is used, main storage files are handled by the inclusive mechanism. Files on disk must be dealt with explicitly for both inclusive and disjoint recovery.

There are only minor differences in the handling of main storage and disk files. Recovery data is maintained on disk for disk files and in main storage for main storage files. It is convenient, and reasonable, to allocate buffers for individual records when recording recovery data in main storage. On disk, each recoverable file has a single recovery file associated with it: it is necessary to manage free space within this file explicitly.

For files, a cache mechanism is used to record changes outside the file itself. Then, it is necessary to redirect input and output to the cache rather than the file, as appropriate, and to move information from the cache to the file only when a recovery point is discarded. Restore operations may be carried out simply by discarding part of the cache. Our technique differs from cache techniques as they are usually applied to main storage [2, 3, 4], in storing new values in the cache rather than old values. Because each read operation is a function call, the cost advantage of allowing reads to proceed unaffected, which is important in main storage, is not significant here.

Our implementation also differs from those cited in that we allow any active recovery point to be discarded, and hence do not enforce strict nesting of recovery regions. As a consequence, it is not possible for us to use an implementation based on a stack of recovery data.

A vector of pairs of integers is maintained for each file, one pair for each record in the file. One integer gives the recovery point identifier for the most recent write to that record, the other gives the location of the cached record for that recovery point. If the record has not been written since the earliest still-active recovery point, the identifier is zero and the location is irrelevant. Each cached record contains a similar (identifier, location) pair for the next record on the chain. For example, if a record was written twice, once when recovery point 2 was the most recent recovery point, and once when recovery point 5 was most recent, there will be two cached copies of the record, as shown in Figure 2.

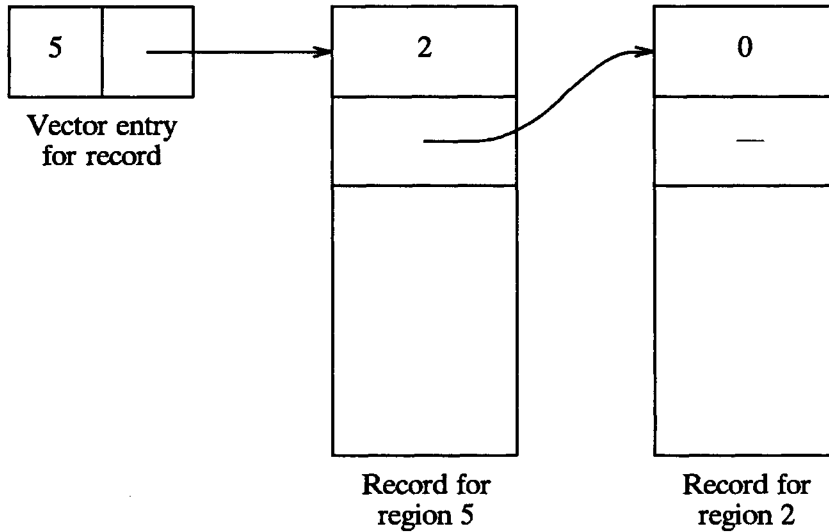


Figure 2. A chain of cached records.

The pair of words associated with the record has an identifier of 5 and points to the record written in recovery region 5. That record has an identifier of 2 and points to the record written in recovery region 2. That record has an identifier of 0; implicitly, the location is the original record in the file.

A read operation reads from the first record on such a chain, reading from the original file only if the chain is empty. A write operation adds a new record to the beginning of the chain, if the first record belongs to a different recovery point, and otherwise overwrites the most recent record. To establish a recovery point, normally no action is required: subsequent writes simply discover that the current recovery point does not match the stored recovery point identifier for any record. The first establish operation must allocate the vector of recovery data. To restore a recovery point, all cache records belonging to that recovery point and more recent recovery points must be discarded. To discard a recovery point, any records belonging to that recovery point must be given to the preceding active recovery point. If both recovery points contain a copy of the record, the copy belonging to the preceding recovery point must be discarded. If the recovery point being discarded is the oldest active recovery point, then records belonging to it must be copied into the file itself.

The technique described here is similar to the techniques described by Severance and Lohman [6] and Rappaport [5], although not as complex. Their methods assume that a very large file is being used, with relatively few changes. Rappaport indicates that less than one per cent of the records will change in a recovery region, for the application described. Severance and Lohman are not discussing a specific system, but also indicate that they anticipate a small fraction of changed records. In our situation, files are smaller and we anticipate that a significant fraction of the records may change within a recovery region. Another difference distinguishing our design from the two cited, is that we support multiple active recovery points, whereas they only support (directly) the notion of a single active recovery point.

3.3. Disjoint recovery for main storage objects

The general mechanism for recoverable main storage objects is presently used for two kinds of objects: mangle tables and bit vectors. Mangle tables are used in IOSYS to record damage artificially added to a file. Such information must not be used by error detection and correction routines, but is often needed to control and evaluate the execution of experiments. Clearly, it is important that mangle tables be restored when the associated file is restored. Bit vectors are used to indicate which records are in a "changed" file and which are still in an unchanging "master" file. Maintaining a file as a differential file from a master file is used to make execution of experiments more efficient. Again, it is important that the bit vector information be maintained appropriately by the recovery mechanism.

Since mangle tables are, in practice, always quite small, recovery simply relies on making copies. Thus, when a recovery point is established, a copy is made of the mangle table. When a recovery point is restored, the copy is copied back onto the original. When a recovery point is discarded, the copy is simply discarded. If recovery is desired for several mangle tables, it is simply necessary to specify each separately to the general recovery mechanism. The mangle table functions use *object* to point to the mangle table, and the establish function returns a pointer to the copy. Then, it is not necessary for any state information to be maintained internally by the mangle table functions.

For efficient access, a bit vector is implemented as a byte vector. To avoid excessive zeroing of the vector, when the changing file is reset to be equal to the master file, each byte stores a pass number. A bit is logically one if the byte contains the current pass number, and logically zero otherwise. For recovery, it is only necessary to maintain the bit values, not the pass numbers. It is possible to achieve efficiency in both space and execution time by storing data for several recovery points in each word of a vector. There is one word for each record in the file, and one recovery point can be stored for each bit in a word. Since, at any time, only one recovery point is being established, restored, or discarded, a bit mask can be calculated and used repeatedly, avoiding the usual inefficiency of accessing individual bits. Other than this compression, recovery for a bit vector is handled simply by making copies. When a recovery point is restored, the pass number is set to one and all entries in the byte vector set to zero or one, since this maximises the time before the entire byte vector must be zeroed.

4. Experience with the Recovery Mechanism

Implementation of the recovery facilities was completed only recently, so experience with them is largely limited to experiments intended to determine their performance characteristics. A number of experiments have been performed in order to determine the cost of establishing, restoring, and discarding recovery points, and the cost of performing I/O operations when recovery is in use.

The first experiment shows the fundamental cost of performing the establish, restore, and discard operations with no I/O performed in a recovery region. Times are given for each of the three operations, in eight different cases. The eight cases result from inclusive versus disjoint recovery and four file configurations. One file was used in each case: the file was either small (50 records) or large (5000 records) and was in main storage or on disk. The record length in each case was 21 words. The restore operations were performed with the DISCARD option. All times are averages over 100 executions of each operation. The times given here, and throughout this section, were obtained on a VAX-11/780.*

		Establish	Discard	Restore
Inclusive Recovery	Small file in main storage	82.8	27.7	45.8
	Small file on disk	86.3	26.8	53.7
	Large file in main storage	619.8	79.5	121.7
	Large file on disk	142.5	89.2	123.8
Disjoint Recovery	Small file in main storage	19.8	5.0	18.3
	Small file on disk	22.2	5.3	18.3
	Large file in main storage	23.7	59.5	81.3
	Large file on disk	25.3	59.5	82.5

Table I: Fundamental operation times in milliseconds, no I/O

From the table, we can observe that inclusive recovery is expensive if a large main storage file is involved, and to a certain extent even if a large disk file is involved. Because the recovery mechanism for disk files is not particularly space-efficient, a large disk file also adds significantly to the amount of main storage data which must be duplicated. The program being used in these tests also has a substantial amount of main storage data in addition to the files (approximately 60K). A very small program, using a small main storage file would have an even better performance than that shown for small files above.

When disjoint recovery is used, the times are all only a few tens of milliseconds, even for a file of 5000 records. Since most files which we use are much smaller than this, the recovery mechanism is able to establish, discard, and

*VAX is a registered trademark of Digital Equipment Corporation.

restore recovery points with acceptable efficiency for our use. Clearly, for extremely large files, a different file recovery mechanism would be needed, to decrease both main storage use and execution time.

In order to provide a source of realistic I/O activity, the linked B-tree storage structure was used in an experiment to determine the I/O overhead in using the recovery facility. The linked B-tree is a robust implementation of a B-tree [8]. It is of special interest to us in respect of recovery, since there was a previously unsolved problem which is handled easily by the recovery mechanism.

The problem is that space in a file may be exhausted during a node split which propagates. If the insert simply stops when no free record can be obtained, an erroneous, unusable B-tree may result. It is possible to determine in advance how many additional records will be required for an insert, but this would add further complexity to an already complex insertion routine. It is also possible to make a pessimistic estimate (one more than the current height of the tree) and refuse to perform insertions if this number of records is not available, but that approach is unfortunate because most insertions require no new records at all. It is also possible to undo a partially completed insert, but that is extremely complex, if done on the basis of examining the partially modified B-tree. Given a recovery mechanism, the last possibility is trivial, so we have added an option to the B-tree implementation to establish a recovery point at the start of each insertion operation, discard the recovery point on successful insertion, and restore the recovery point on failing insertion.

Because exhausting the free list in the file causes strongly divergent behaviour between the recovery and non-recovery cases, the results shown below are all for the situation in which the file was sufficiently large. They are simply intended to show the total overhead of establishing and discarding recovery points, combined with the added expense of performing I/O through the recovery mechanism.

Five cases were tested, each being executed using a file in main storage and using a file on disk; as well, both inclusive and disjoint recovery were tested. In each case, a B-tree of order 2 containing 500 keys was built by successive pseudo-random insertions into an empty tree. (Such a linked B-tree has a record length of 21 words, for consistency this record length was also used in the first experiment, reported above.) The times shown are averages of the time to build a B-tree, over five repetitions of the experiment. The cases tested were (1) using a version of IOSYS in which the recovery code does not exist, (2) using a version of IOSYS in which the recovery code exists but no recovery is requested, (3) establishing and discarding a single recovery point, with the recovery region including the construction of the entire B-tree, (4) establishing and discarding a recovery point for each insertion, and (5) a combination of the two preceding cases: one recovery region enclosing all insertions and a nested recovery region for each individual insertion. Note that for cases (1) and (2), type of recovery is irrelevant, so the same figures are reported under both "Inclusive recovery" and "Disjoint recovery."

Clearly, using one recovery region for each insert operation introduces significant additional overhead, particularly if inclusive recovery is used. For disjoint recovery, the overhead is roughly 100% (somewhat greater in the case of

	Inclusive recovery		Disjoint recovery	
	File in main storage	File on disk	File in main storage	File on disk
No recovery code	5.92	24.77	5.92	24.77
Recovery code not in use	6.15	24.80	6.15	24.80
One recovery region	6.60	35.65	7.20	35.88
One recovery region per insert	83.76	103.95	11.20	53.98
Nested recovery regions	86.54	107.61	13.41	54.06

Table II: Times for building an LB-tree, in seconds

disk files), which could be tolerated in some circumstances but is clearly undesirable. The overhead for inclusive recovery is much worse, and would almost always be intolerable.

The figures for using only one recovery region indicate that the recovery mechanism adds little overhead to I/O operations. Indeed, for files in main storage the overhead is likely negligible.

Two other observations can be made from the data in the table. First, there is some static overhead simply from the presence of the recovery code, even when it is not used, but the overhead is small. Second, for disjoint recovery, when using a disk file, nested recovery regions have almost exactly the same cost as one recovery region per insert. This seems odd since the cost of discarding the outer recovery region is clearly high: it involves copying most of the file, record by record. This cost is overcome by a cost saving in discarding the recovery points for the individual insert operations. It is more efficient to give recovery data to an enclosing recovery region (a relatively minor manipulation of the recovery file) than to copy recovery data into the main file (which involves copying an entire record).

The times reported in this section were obtained after some modest performance tuning of the recovery mechanism. It is likely that they could be improved by further tuning, but in the case of inclusive recovery there is a significant overhead in making a complete copy of the data area of the program, which cannot be decreased without major changes. If the UNIX kernel dynamically created copies of pages as they were changed, rather than copying all pages during a *fork* (a possibility allowed by the hardware) then the inclusive recovery option would be much less expensive. Changing the kernel is not a possibility for us, and this change would undoubtedly be a complex one to make.

5. Conclusions and Further Work

In this paper, we have described a recovery mechanism with a reasonably simple user interface and a reasonably simple implementation. Although the mechanism is simple, it is also flexible, allowing various alternatives to be tested experimentally. The mechanism is sufficiently efficient for experimentation althought not, in most cases, for a production environment.

Implementation of the mechanism was reasonably easy for two reasons. One is that UNIX provides very flexible and general facilities. Although some details of these facilities can be frustrating, they form a powerful base for building above them. The second reason is that the file system affected was our own code and could be modified as necessary.

Various extensions and improvements remain to be explored. For the implementation of recovery blocks, it would be useful to have a "prior" feature, allowing access to values which existed when the most recent recovery point was established. For inclusive recovery, this presents serious problems, but should be possible for disjoint recovery.

The experiments reported in the preceding section have shown that large files not only use much space for recovery vectors (as expected), but also that significant time is lost scanning the vectors during discard and restore operations. The use of hashing to reduce the size of the vectors will also be explored.

Acknowledgement

The work described in this paper was supported by the Natural Sciences and Engineering Research Council of Canada, under grant A3078 and a Postgraduate Scholarship.

References

1. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*, Prentice-Hall, Englewood Cliffs, N. J. (1981).
2. T. Anderson and R. Kerr, Recovery blocks in action: A system supporting high reliability, *Proceedings, 2nd International Conference on Software Engineering*, pp. 447-457 (October 13-15, 1976).
3. J. J. Horning *et al*, A program structure for error detection and recovery, pp. 171-187 in *Lecture Notes in Computer Science*, ed. E. Gelenbe and C. Kaiser, Springer Verlag, Berlin (1974).
4. P. A. Lee, N. Ghani, and K. Heron, A recovery cache for the PDP-11, *IEEE Transactions on Computers* **C-29**(6) pp. 546-549 (June 1980).
5. R. L. Rappaport, File structure design to facilitate on-line instantaneous updating, *Proceedings, ACM-SIGMOD International Conference on Management of Data*, pp. 1-14 (May 14-16, 1975).
6. D. G. Severance and G. M. Lohman, Differential files: Their application to the maintenance of large databases, *ACM Transactions on Database Systems* **1**(3) pp. 256-267 (September 1976).

7. D. J. Taylor and J. P. Black, Experimentation with data structures, CS-84-52, Dept. of Computer Science, University of Waterloo (December 1984). Accepted for publication in *Software–Practice and Experience*.

8. D. J. Taylor and J. P. Black, A locally correctable B-tree implementation, CS-84-51, Dept. of Computer Science, University of Waterloo (December 1984). To appear in *Computer Journal*, vol. 29, no. 1, February 1986.