NASA Contractor Report 172122

Development of Software Fault-Tolerance Techniques

Peter Michael Melliar-Smith

SRI International
Menlo Park, California 94025

Contract NAS1-15480

March 1983

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report 172122

Development of Software Fault-Tolerance Techniques

Peter Michael Melliar-Smith

SRI International
Menlo Park, California 94025

Contract NAS1-15480

March 1983

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

List of Figures

1 INTRODUCTION

As computers become more widely used, and in particular as they become used in more safety critical applications, the reliability of the computer system becomes more important. NASA and FAA have established stringent reliability requirements for computer systems performing critical functions in aircraft control applications, and similar reliability requirements exist in nuclear power plant control, medical intensive care units, air traffic control, and certain military applications. We should also note a need for high levels of reliability in applications involving the distribution of very large numbers of relatively inexpensive units, where repair or replacement of defective systems could become expensive. The most immediate example of such an application is the use of small computers to control automobile engines.

Reliable operation depends on both correct design and on fault free manufacture and operation of that design. Traditionally, because designs were simple and hardware components were of only modest reliability, the term "reliability" has been associated with hardware reliability and the absence of failed hardware components. The Flight Electronics Division of NASA Langley Research Center has sponsored the development of two highly reliable computer systems, SIFT [5] and FTMP [3], designed to achieve a very high degree of immunity to hardware component failures. From these designs, it is clear that the "hardware" reliability problem is soluble. But as systems become more complex, and as hardware components become more reliable, it is clear that residual defects are becoming a significant, even a limiting, factor in the reliability of systems. Here design encompasses the design of the hardware, the design and programming of the software, and the overall system design, requirements analysis, and specification. Because the most complex portions of our systems are implemented in software, it is in the area of the software that most attention has been given to design defects and to techniques of fault tolerance that can reduce the effect of design errors on system reliability. However, as the hardware of our systems becomes increasingly complex, design fault tolerance techniques currently proposed for software may become applicable within the hardware design as well.

In this report, we first consider the nature of faults, errors and failures, fault tolerance and intolerance, and the assumptions made by different possible approaches to correct operation. In Section 3 we describe the recovery block techniques upon which our research has been based. Section 4 contains a probabilistic analysis of the reliability of systems containing recovery blocks, and a discussion of the effects of correlated design errors on that reliability. In Section 4 the importance of very accurate acceptance tests is demonstrated. Section 5 discusses the derivation of acceptance tests from software specifications, showing that appropriate specification styles permit the construction of rigorous acceptance tests without undue costs. Section 6 describes possible recovery block cache mechanisms for complex type and process structures such as are found in many modern systems. We show that recovery blocks are quite consistent with these modern program structuring techniques. Section 7 investigates the recovery of systems of asynchronous interacting processes, still a major problem area for the recovery block approach.

Sections 8 and 9 discuss and describe a design for the implementation of recovery blocks, using microprogram, on the Bendix BDX930 processor. Sections 8 and 9 show how to take advantage can be taken of the characteristics of flight control programs to simplify the recovery block mechanisms and thus obtain an efficient implementation with only limited hardware cost. Section 10 contains an example of the use of recovery blocks in a

simple operating system, and illustrates the use of many of the techniques discussed in sections 4 through 7. Finally, Section 11 contains our conclusions.

2 THE RELATIONSHIP BETWEEN SOFTWARE FAULT TOLERANCE AND OTHER APPROACHES TO SOFTWARE RELIABILITY

2.1 Fault Tolerance and Fault Intolerance

Avizienis [1] has distinguished two classes of approaches to reliability, those based on fault tolerance, and those based on fault intolerance. Approaches based on fault tolerance are invoked after the occurrence of the fault and are intended to ensure the continued successful operation of the system despite the occurrence of the fault. Approaches based on fault intolerance depend on action taken in advance to ensure that the fault does not occur. The nature of the fault hazard being considered will determine which approach is most likely to be effective. The two approaches are not, of course, mutually exclusive and any real highly reliable system will contain elements of both approaches.

Traditionally hardware designers have used fault-tolerance techniques to protect their systems, because hardware component faults have been the primary cause of unreliability and cannot be entirely precluded. Software designers, in contrast, have used fault-intolerance techniques to eliminate, or at least reduce, the impact of software design faults on their systems. Software does not suffer from the kind of component degradation in service that makes a fault-tolerance approach essential, and the level of reliability obtained from fault-intolerance methods has sufficed. However, as hardware has become more complex, increasing problems of hardware design faults have occurred, and methods of fault intolerance are being borrowed from software development. Conversely as software is being required to achieve higher levels of reliability than can be obtained from current methods of fault intolerance, so methods of fault tolerance are being considered for software.

It is inevitable that a complex system will contain, at all times, a number of potential faults. There may be faults in the hardware and in the operating system, in the logic design and in the hardware components, in the application system design and specification and in the application programming, in the actions of the operations staff and of the maintenance engineers, and in the system's environment.

It is particularly important to note that the information content of the system may not be correct. Incorrect information in a system is particularly serious, because that information may disrupt subsequent processing, resulting in further incorrect information and yet more disrupted processing. Fault-tolerance techniques may be able to protect a system against incorrect information.

It is also appropriate to recognize that design faults may have significantly more serious consequences than component failures. A component failure is usually confined to that component and possibly a few other closely associated components. It may even be possible to enumerate the consequences of component failures. A design error, however, demonstrates that the design is not understood, and, of course, specific design errors cannot be anticipated. The possible failure modes are much wider for design errors and

cannot be enumerated. In particular a design error in one part of a system might damage the processing of some other part of the system apparently quite unconnected with it. The safe assumption is that the behavior of a subsystem affected by a design error can be quite arbitrary, including the generation of plausible but subtly misleading results.

2.2 Systems and their Failures

Before proceeding further, it is appropriate to consider more carefully what we mean by faults, errors, and failures.

We define a system as a set of components together with their interrelationships, which system has been designed to provide a specified service. The components of the system are themselves systems, and we term their interrelationships the algorithm of the system. There is no requirement that a component provide service to a single system; it may be a component of several distinct systems. The algorithm of the system is, however, specific to each system individually.

This definition of 'system', with its insistence that the service provided must be specified (but not necessarily prespecified), is intended to exclude systems which are "intelligent" in the sense of being capable of determining their own goals and algorithms. At present, intelligent systems are not understood sufficiently to permit consideration of their reliability.

The internal state of a system is the aggregation of the external states of all its components. The external state of a system is the result of a conceptual abstraction function applied to its internal state. During a transition from one external state to another external state, the system may pass through a number of internal states for which the abstraction function, and hence the external state, is not defined. The specification defines only the external states of the system, the results of these operations, and the transitions between external states caused by these operations, the internal states being inaccessible from outside the system.

The service provided by a system is regarded as being provided to one or more environments. Within a particular system, the environment of a given component consists of those other components with which it is directly interrelated.

A failure of a system occurs when that system does not perform its service in the manner specified, whether because it is unable to perform the service at all, or because the results and the external state are not in accordance with the specifications. A failure is thus an event. There is, however, no implication that the event is actually recognized as having occurred. For example, if an environment does not make full use of that particular result of the system then the failure to provide it will have no effect.

2.3 Error and Faults

In contrast to the simple, albeit very broad, definition of 'failure' given above, the definitions we now present of 'error' and 'fault' are not so straightforward. This is because they aim to capture the element of subjective judgment which we believe is a necessary aspect of these concepts, particularly when they relate to problems which could have been caused by design inadequacies in the algorithm of a system.

We term an internal state of a system as an erroneous state when that state is such that there exist circumstances, within the specification of the use of the system, in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault. The subjective judgment that we wish to associate with the classification of a state as being an erroneous one derives from the use of the phrases "normal algorithms" and "which we do not attribute" in this definition - however further definitions are required before these matters can be discussed properly.

The term error is used to designate that part of the state which is "incorrect." An error is thus an item of information, and the terms error, error detection, and error recovery are used as casual equivalents for erroneous state, erroneous state detection, and erroneous state recovery.

A fault is the mechanical or algorithmic cause of an error, while a potential fault is a mechanical or algorithmic construction within a system such that, under some circumstances within the specification of the use of the system, that construction will cause the system to assume an erroneous state. It is evident that the failure of a component of a system is (or rather, may be) a mechanical fault from the point of view of the system as a whole.

A system can be designed to be fault-tolerant by incorporating into it abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures. The degree of fault-tolerance will depend on the success with which erroneous states corresponding to likely faults are identified and detected, and the success with which such states are repaired.

It should now be clear that the generality of our definitions of failure and fault has the intended effect that the notion of fault encompasses such design inadequacies as a mistaken choice of component, a misunderstood or inadequate specification (of either the component, or the service required from the system) or an incorrect interrelationship amongst components (such as a wrong or missing interconnection, in the use of hardware systems, or a program bug in software systems), as well as, say, hardware component failure due to aging.

Note that the definition of an erroneous state depends on the subdivision of the algorithm of the system into normal algorithms and abnormal algorithms. These abnormal algorithms will typically be the error-recovery algorithms. There are many systems in which that subdivision, and hence the designation of states as erroneous, is a matter of judgment.

For example, in a storage system utilizing a Hamming Code, one may regard the correction circuits as error recovery mechanisms and a single incorrect bit as an error. Alternatively the correction circuits may be regarded as a normal mechanism, and thus a single incorrect bit would not be regarded as an error, though two incorrect bits would be.

Note also that a demonstration that further processing can lead to a failure of the system indicates the presence of an error, but does not suffice to locate a specific item of information as the error. Consider a system affected by an algorithmic fault. The sequence of internal states adopted by this system will diverge from that of the "correct" system at some point, the algorithmic fault being the cause of this transition into an erroneous state. But there can be no unique correct algorithm. It may be that any one of several changes to the algorithms of the system could have precluded the failure. A

subjective judgment as to which of these algorithms is the intended algorithm determines the fault, the items of information in error, and the moment at which the state becomes erroneous. Some such judgements may, of course, be more useful than others.

The significance of the distinction between faults and errors may be seen by considering the repair of a data-base system. Repair of a fault may consist of the replacement of a failing program (or hardware) component by a correctly functioning one. Repair of an error requires that the information in the data base be changed from its currently erroneous state to a state which will permit the correct operation of the system. In most systems, recovery from errors is required, but repair of the faults which cause these errors, although very desirable, is not necessarily essential for continued operation.

2.4 Approaches to Software Reliability

It is appropriate to compare three approaches that can be used to obtain improved software reliability. Program testing and program proof are examples of fault intolerance while fault-tolerant software using recovery blocks is, of course, an example of fault tolerance. Each of these approaches is effective but each of them has its limitations. Each of these approaches depends on certain assumptions. Were these assumptions justified, then each approach would be able to assure any desired level of reliability. However, the assumptions are seldom, perhaps never, entirely justified, and flaws in those assumptions leave opportunities for system unreliability. When one seeks to meet an exceptional reliability requirement, one must consider carefully the validity of the assumptions one is making.

When we seek to validate a program by testing, we assume:

- A correct specification of the system to be implemented
- The environment, simulated or real, is correct
- The set of test cases is sufficient, and
- Errors will be recognized.

Clearly the weakest of these assumptions is that the set of test cases is sufficient. A very large number of tests, far beyond any feasible test program, would be required. Work has been done on the generation of comparatively small test sets, by analysis of the program. These methods involve techniques, and assumptions, very similar to those required for program proof. The assumption that errors will be recognized is also weak, for, in many cases of system failure, it is found that test cases had also triggered the program fault but that the resulting error had not been recognized.

When we seek to validate a program by proof, by the closely related techniques of formal program development from its specification, we assume:

- A correct specification of the system to be implemented
- A correct specification of the environment and the controlled equipment

- A correct specification and implementation of the programming language and of the computer
- An implementation of the proof is possible, and
- The proof is sound.

Here, a very great amount of work on the specifications and the proof may significantly increase our confidence that the assumptions are justified. However these assumptions are much further removed from the actual system, and that distance must itself introduce some uncertainty.

When we seek to enhance the reliability of a program by use of software fault tolerance, such as recovery blocks, we assume:

- A correct specification of the system to be implemented
- Recognition of errors (by the acceptance tests)
- Some aspects of the specification of the computer concerning the recovery mechanism, and
- Independence between alternate blocks, and independence of the alternate blocks from the acceptance tests.

Here the primary concern must be that the assumption of independence is unjustified.

Comparison of the assumptions, for each of these three approaches to improving program reliability, shows that each pair of the approaches has assumptions in common and thus has common vulnerabilities. But only one assumption is common to all three approaches, that of the correct specification of the system to be implemented. Any errors in that specification will be faithfully implemented in the programs, enforced by our validation methods, and foiled by failure of the actual system in service. Our concern over the assumption that the specification is correct is increased because the specifications are long, complex, and highly detailed, and are written in a language quite unfamiliar to those best able to determine what the behavior of the system should be.

It is possible to write much more abstract and general statements about the behavior of the system, i. e., the formal requirement definitions. These requirement definitions can be expressed in a form that much more closely matches the user's perception of his needs from the system, but it is difficult to ensure that all of his needs have been expressed as formal requirements. Further, the very abstract nature of these requirements causes technical difficulties in expressing them and in validating them. In principle the three approaches still apply though testing is less likely to be effective. The proof approach now involves a proof that the system specifications satisfy the requirement definitions, a quite lengthy and difficult proof for which mechanical assistance is at present unavailable. The software fault-tolerance approach is directly applicable, using the user requirements as a source of acceptance tests, but may involve difficulties because many requirement definitions encompass the entire system, and recovery will be expensive.

Once we are satisfied with the specifications of the system, it is clear that the highest

reliability is obtained through the use of all three approaches, testing, proof, and software fault tolerance. Only by the use of all three approaches can we avoid common assumptions whose invalidity might be a source of unreliability.

2.5 The Use of Recovery Blocks to Allow Evolution without Loss of Reliability

Let us envisage that the flight-control system is initially developed and validated using very stringent methods at great expense, for instance using formal-proof techniques. Both a primary alternate and an acceptance test would be so validated. These initial versions of the flight-control programs are likely to be relatively simple, having been constrained by the limitations of the development and validation methods. Thus, while they are certain to be safe, they are liable to be deficient in other properties of importance to the aircraft manufacturer and the airlines, properties such as ride and fuel economy. It is therefore inevitable that modifications and elaborations to these programs will be required, and there is a risk that the modified versions of the programs may not be as reliable as the original, simpler and more highly validated, programs.

However, as suggested by NASA personnel, the use of Recovery Blocks, as described in the following section, could allow highly validated acceptance tests and alternates to be retained at all times, and any improved version permitted only as an alternate, say the primary one. Such a use of Recovery Blocks would insure that the reliability of the system is always protected by a fully validated acceptance test and alternate, while allowing some opportunity to develop improved versions of the flight control programs.

3 A BRIEF DESCRIPTION OF THE RECOVERY BLOCK TECHNIQUE

A recovery block [4] consists of a conventional block which is provided with error detection (an acceptance test) and zero or more stand-by spares (the additional alternates). A possible syntax for recovery blocks is as follows:

<recovery block>::=ensure<acceptance test> by
<primary alternate>
<other alternates> else error

<primary alternate>::=<alternate>

<other alternates>::=<empty>|<other alternates>
else by <alternate>

<alternate>::=<statement list>

<acceptance test>::=<logical expression>

The primary alternate corresponds exactly to the block of the equivalent conventional program, and is entered to perform the desired operation. The acceptance test, which is a logical expression without side-effects, is evaluated on exit from any alternate to determine whether the alternate has performed acceptably. A further alternate, if one

exists, is entered if the preceding alternate fails to complete (e.g., because it attempts to divide by zero, or exceeds a time limit), or fails the acceptance test. However, before an alternate is so entered the state of the process is restored to that current just before entry to the primary alternate. If the acceptance test is passed, any further alternates are ignored and the statement following the recovery block is the next to be executed. However, if the last alternate fails to pass the acceptance test then the entire recovery block is regarded as having failed, so that the block in which it is embedded fails to complete, and recovery is then attempted at that level.

```
A:  ensure AT
    by      AP : begin
                     <program text>
                 end
    else by  AQ : begin
                     <program text>
                 end
    else error
```

Figure 1: A simple recovery block

In the illustration of a recovery block structure in Figure 1, double vertical lines define the extents of recovery blocks, while single vertical lines define the extents of alternate blocks, primary or otherwise. Figure 2 show that the alternate blocks can contain, nested within themselves, further recovery blocks.

Consider the recovery block structure shown in Figure 2. The acceptance test BT will be invoked on completion of primary alternate BP. If the test succeeds, the recovery block B is left and the program text immediately following is reached. Otherwise, the state of the system is reset and alternate BQ is entered. If BQ, and then BR, do not succeed in passing the acceptance test, the recovery block B as a whole, and therefore primary alternate AP, are regarded as having failed. Therefore, the state of the system is reset to that current just before entry to AP and alternate AQ is attempted.

Deferring for the moment questions as to how the state of the system is reset when necessary, the recovery block structure can be seen as providing a very general framework for the use of stand-by sparing. There is no need for, indeed no possibility of, attempts at automated error diagnosis because the system state is reset after an error, deleting all effects of the faulty alternate. Once the system state is reset, switching to the use of an alternate is merely a matter of a simple transfer of control.

```
A:ensure AT
  by      AP:begin declare Y
                <program text>
             B:ensure BT
                by      BP:begin declare U
                              <program text>
                           end
                else by  BQ:begin declare V
                              <program text>
                else by  BR:begin declare W
                              <program text>
                           end
                else error
                <program text>
           end
else by    AQ:begin declare Z
                <program text>
             C:ensure CT
                by      CP:begin
                              <program text>
                           end
                else by  CQ:begin
                              <program text>
                           end
                else error
             D:ensure DT
                by      DP:begin
                              <program text>
                           end
                else error
           end
else error
```

Figure 2: A more complex recovery block

The concept of a recovery block in fact has much in common with that of a sphere of control, as described by Davies [2]. However, we have limited ourselves to preplanned error recovery facilities and base all error recovery on automatic reversal to a previously reached recovery point. Once a process has "committed" itself by accepting the results of a recovery block, the only available form of recovery from an error within that block involves a more global process reversal, to the beginning of an enclosing recovery block whose results have not yet been accepted. In contrast, Davies is prepared to allow for the possibility of recovery following commitment, by means of programmer-supplied "error compensation algorithms".

The utility of the recovery block scheme for stand-by sparing in software rests on the

practicability of producing useful acceptance tests and alternates, and on the cost of resetting the system state. We will discuss each of these points in turn.

3.1 Acceptance Tests

The function of the acceptance test is to ensure that the operation performed by the recovery block is satisfactory to the program which invoked the block. The acceptance test is therefore performed by reference to the variables accessible to that program, rather than variables local to the recovery block since these can have no effect or significance after exit from the block. Indeed the different alternates will probably have different sets of local variables. There are no separate acceptance tests for the different alternates. The surrounding program may be capable of continuing with any of a number of possible results of the operation, and the acceptance test must establish that the results are within this range of acceptability, without regard for which alternate can generate them.

There is no requirement that the test be, in any formal sense, a check on the absolute 'correctness' of the operation performed by the recovery block. Rather it is for the designer to decide upon the appropriate level of rigour of the test. Ideally, the test ensures that the recovery block meets all aspects of its specification that are depended on by the program text that calls it.

When an acceptance test fails, all the evidence is hidden from the alternate that is then called, but a detailed log can be kept of such incidents for off-line analysis. Some failures to pass the acceptance test may be spurious, because a design inadequacy in the acceptance test itself has caused an unnecessary rejection of the operation of an alternate. The execution of the program of the acceptance test itself might suffer an error and fail to complete. Such occurrences, will be rare, since the aim is to have acceptance tests which are much simpler than the alternates they check, and are treated as failures in the enclosing block. Like all other failures they can also be recorded in the error log. Thus, the log provides a means of finding these two forms of inadequacy in the design of the acceptance test. The remaining form of inadequacy, that which causes the acceptance of an incorrect set of results, is, of course, more difficult to locate.

When an acceptance test is being evaluated, any non-local variables that have been modified must be available in their original as well as their modified form because of the possible need to reset the system state. For convenience and increased rigour, the acceptance test is allowed to access variables in either their modified value or their original (prior) value. A further facility available inside an acceptance test could be a means of checking whether any of the variables that have been modified have not yet been accessed within the acceptance test, to assist in detecting sins of commission, as well as omission, on the part of the alternate.

Figure 3 shows a recovery block whose intent is to sort the elements of the vector S. The acceptance test incorporates a check that the set of items in S after operation of an alternate are indeed in order. However, rather than incur the cost of checking that these elements are a permutation of the original items, it merely requires the sum of the elements to remain the same.

```
ensure sorted (S) /\ (sum(S) = sum(prior S))
by quickersort (S)
else by quicksort (S)
else by buddlesort (S)
else error
```

Figure 3: A fault-tolerant sort program

3.2 Alternates

The primary alternate is intended to be used normally to perform the desired operation. Other alternates perform the desired operation in some different manner, presumably less economically, and preferably more simply. As long as one of these alternates succeeds, the desired operation will have been completed and only the error log will reveal any troubles that occurred.

In many cases one might have an alternate which performs a less desirable operation but one which is still acceptable to the enclosing block, in that it will allow the block to continue properly. (One plentiful source of both these kinds of alternates might be earlier releases of the primary alternate! See Section 2.5.)

Figure 4 shows a recovery block consisting of a variety of alternates. The aim of the recovery block is to extend the sequence S of items by a further item i, but the enclosing program will be able to continue even if afterwards S is merely "consistent". The first two alternates try, by different methods, to join the item i onto the sequence S. The other alternates make increasingly disparate attempts to produce at least some sort of consistent sequence, providing appropriate warnings as they do so.

```
ensure consistent sequence (S)
by extend S with (i)
else by concatenate to S (construct sequence (i))
else by warning ("lost item")
else by S:= construct sequence (i); warning ("correction, lost sequence")
else by S:= empty sequence; warning ("lost sequence and item")
else error
```

Figure 4: A recovery block with alternates which achieve different, but still acceptable, though less desirable results.

3.3 Restoring the System State

By making the resetting of the system state completely automatic, the programmers responsible for designing acceptance tests and alternates are shielded from the problems of this aspect of error recovery. No special restrictions are placed on the operations which are performed within the alternates, on the calling of procedures or the

modification of global variables, and no special programming conventions have to be adhered to. In particular the error-prone task of explicit preservation of restart information is avoided. It is thus that the recovery block structure provides a framework which enables additional program text to be added to a conventional program, for purposes of specifying error detection and recovery actions, with good reason to believe that despite the increase in the total size of the program its overall reliability will be increased.

All this depends on automating the resetting of the system state whose overheads are tolerable. Clearly, taking a copy of the entire system state on entry to each recovery block, though in theory satisfactory, would in normal practice be far too inefficient. Any method involving the saving of sufficient information during program execution for the program to be executable in reverse, instruction by instruction, would be similarly impractical.

Whenever a process has to be backed up, it is to the state it had reached just before entry to the primary alternate. Therefore the only values that have to be reset are those of non-local variables that have been modified. Since no explicit restart information is given, it is not known beforehand which non-local variables should be saved. Thus, recovery blocks require a mechanism to save non-local variables in a 'cache' when necessary, i.e., just before they are modified. This mechanism detects, at run time, assignments to non-local variables, recognizing that an assignment to a non-local variable is the first assignment to that variable within the current alternate. Thus, precisely sufficient information can be preserved.

The cache is divided into regions, one for each nested recovery level, i.e., for each recovery block that has been entered and not yet left. The entries in the current cache region will contain the prior values of any variables that have been modified within the current recovery block. In case of failure, these cache entries can be used to back up the process to its most recent recovery point. The region can be discarded in its entirety after it has been used for backing up a process. However, if the recovery block is completed successfully, some cache entries can be discarded, but those that relate to variables which are non-local to the enclosing environment must be consolidated with those in the underlying region of the cache.

4 A PROBABILISTIC ANALYSIS OF THE RELIABILITY OF RECOVERY BLOCKS

We need to be able to evaluate the effectiveness of a recovery block structure for improving the reliability of a program, so as to be able to deploy our efforts and costs in the manner that gives the greatest increase in reliability. However, even if we can construct a model that is useful in comparing alternative strategies for their effect in increasing reliability, we must not believe that the model can predict the actual reliability of a program; the model makes assumptions that are at least open to doubt, and the input data are purely conjectural.

In this analysis, we assume complete independence between the alternates and the acceptance tests and between the various alternates. It would be very desirable to be able to include the correlations between those programs in the model, but at present we do not know how to measure, to express, or to analyze the correlations between programs. We will consider the problem further, but a conceptual breakthrough is required and there can be no expectation of success.

The analysis below is for a single sequential program. We would like to extend the analysis to interacting asynchronous programs. The principal problem in any such extension is the much wider range of possible structures for the interacting asynchronous programs. However we do intend to extend this analysis to at least a few of the more obvious structures.

The program structure we consider here is of a set of nested recovery blocks in a sequential program. Each recovery block contains an acceptance test and N alternate blocks, while each alternate contains a section of code and M embedded recovery blocks (except at the deepest level at which the alternate contains only its section of code). The recovery blocks are nested L levels deep. The analysis considers that each execution of a section of code has a probability (f) of creating an error (f being very small), and that each execution of an acceptance test has a probability (f) of not detecting any error that has been submitted to it (g being small but not necessarily very small). Let us denote by E_i and F_i, respectively, the probabilities of obtaining an erroneous result, or an error return, from the recovery block at the i-th level, the objective of the analysis being to calculate E_1 and F_1. The general structure of the analysis is shown in Figure 5.

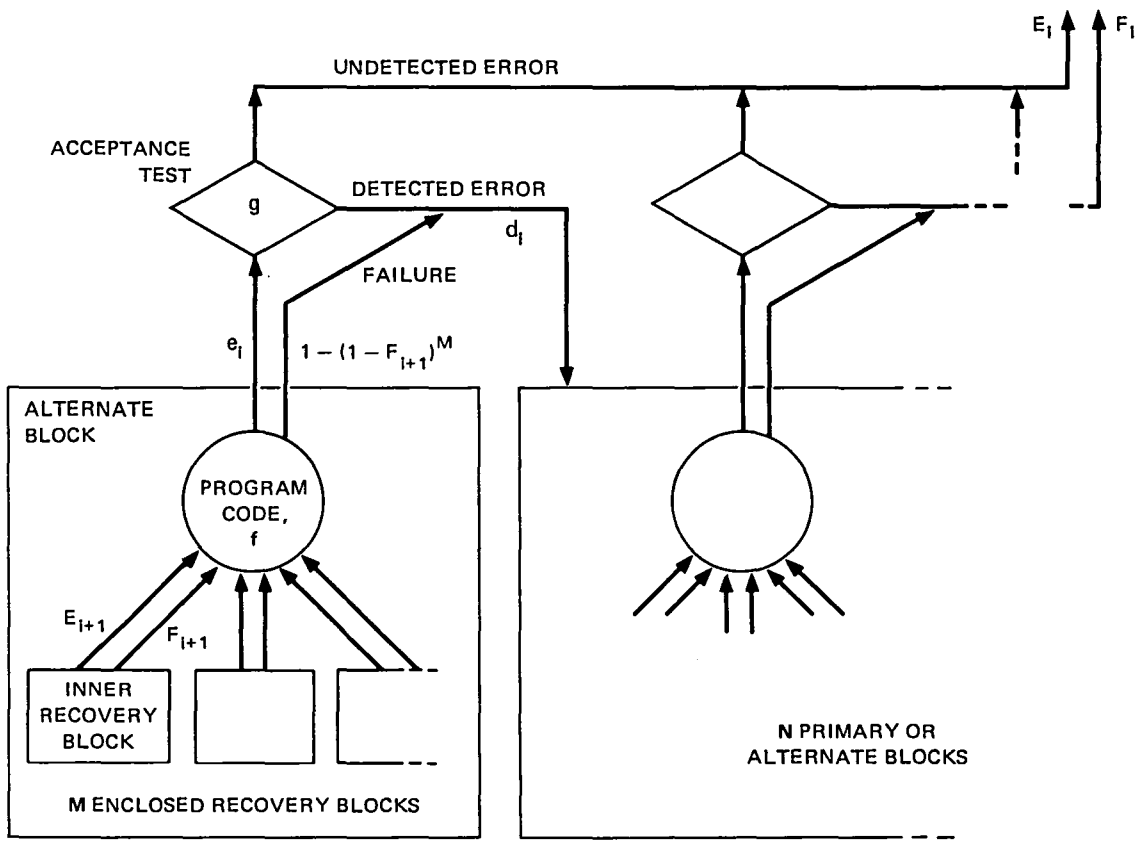

Figure 5: The Probabilistic Analysis of a Recovery Block

The probability of error on exit from an alternate block at level i (e_i) contains components from the probability of generating an error during the execution of the code of the block (f) and also components from the probability for an undetected error in one of the M enclosed recovery blocks, (E_{i+1}).

$$e_i = (1 - F_{i+1})^M - (1 - f)(1 - E_{i+1} - F_{i+1})^M$$

The probability of a detected error in an alternate block (d_i) includes components from the probability of detecting one of the errors above, and also components from the probability of an error return from one of the M enclosed recovery blocks (F_{i+1}).

$$d_i = 1 - (1 - g)(1 - f)(1 - E_{i+1} - F_{i+1})^M - g(1 - F_{i+1})^M$$

The probability of an error return (F_i) from a Recovery Block, through exhaustion of its list of alternates, is the probability of detecting an error for each of the N alternate blocks.

$$F_i = (d_i)^N$$

The probability of an undetected error on exit from the recovery block (E_i) contains components for the probability of undetected error from each of the alternates, weighted by the probability that the alternate will be invoked because all preceding alternates have been detected to be in error.

$$E_i = ge_i + ge_i.d_i + ge_i(d_i)^2 + \text{- - -}$$

$$= \frac{ge_i\ (\ 1 - (d_i)^N\ )}{(\ 1 - d_i\ )}$$

$$= \frac{ge_i\ (\ 1 - F_i\ )}{(\ 1 - d_i\ )}$$

One can now obtain a recurrence relation for E_i and F_i, but that relation is far too complex to be solved readily. Thus approximations must be made, based on f (and thus e_i. d_i, F_i and E_i) being very small.

Thus

$$e_i \simeq f + M E_{i+1}$$

and

$$d_i \simeq (1 - g)\ e_i + MF_{i+1}$$

giving

$$F_i = (d_i)^N \simeq 0 \qquad (\text{for } n \geq 2)$$

and

$$E_i \simeq ge_i \simeq gf + g M E_{i+1}$$

with solution

$$E_1 = gf(1 + gM + g^2M^2 + - - g^{L-1}M^{L-1}$$

or

$$E_1 = \frac{gf\ (1-(gM)^L)}{(1-gM)}$$

This should be compared with the probability of error in a similar system containing no redundancy and no acceptance test

$$E_o = \frac{f(1-M^L)}{(1-M)}$$

Examining these results we note that, to this approximation, the number of alternates N does not appear in the probability of error and is not significant provided that it is two or more.

Further, the importance of ensuring that gM is smaller than 1 is evident. In the interesting case of an already quite reliable system, with e_i and f_i already small, reliability is improved at each level only if the acceptance test detects more errors than are passed up undetected from the embedded recovery blocks. If gM is small enough, the importance of a deeply nested structure for improved reliability is evident.

Lastly, if we need very high reliability and very low values of E, then g must be kept small. Prior work has emphasized "acceptability" rather than "correctness" as the criterion for acceptance tests. One might doubt that a criterion of "acceptability" would reduce the value of g to the required level.

4.1 The Effect of Correlated Faults

Were all faults in the various alternates and acceptance tests entirely independent (and were the values f and g accurately known), this analysis would provide a reasonable estimate for the reliability of a program structure containing recovery blocks. Unfortunately the assumption of independence is unrealistic for some faults. Here we investigate the consequences of correlation, assuming the existence of faults of two different types:

- Random program coding errors, assumed to be completely uncorrelated and also uncorrelated with the various tests and other validation procedures.

- Systematic design errors, assumed to be completely correlated, thus affecting all alternates, acceptance tests, and validation procedures.

Designating the rate of errors due to random coding errors by f and the rate due to systematic design errors by F, the rate of undetected errors in a non-redundant system is:

$$E_0 = \frac{(f+F)(1-M^L)}{(1-M)}$$

and for a system with recovery blocks would be:

$$E_1 = \frac{gf\ (1-(gM)^L)}{(1-gM)} + \frac{F\ (1-M^L)}{(1-M)}$$

Clearly, since recovery blocks are ineffective against correlated faults, the hoped for substantial reliability increase is obtained only if F is very much smaller than f. We must therefore consider the probable magnitudes of f and F for the types of programs of interest and, in particular, for safety critical flight-control programs.

When a program is first written, it contains numerous faults, many of them random coding faults and some systematic design faults. For this newly constructed and little tested system (and many systems never get beyond this stage), recovery blocks should substantially increase reliability.

Safety critical flight-control programs, however, are subjected to very rigorous validation procedures that aim to detect and eliminate almost all of the faults from the programs. The random coding faults, by the assumption of independence, will be susceptible to location by validation tests, and to further continuing detection during normal operation. Some of the design faults will also be found during validation, but others will systematically evade detection during validation and early operation. The rate at which these residual faults cause errors will undoubtedly be low, but may still be much too high to be acceptable for potentially catastrophic failures. Thus we must expect that, in an operational flight-control program, a high proportion of the residual faults will be of the highly correlated, systematic design fault type.

Let us now consider faults intermediate between the random coding faults and the

systematic design faults, faults that are partially but not completely correlated with other alternates, acceptance tests, or validation procedures. If validation procedures are very thorough, many uncorrelated or only partially correlated faults should be located and removed; only faults that are quite highly correlated with the validation procedures should escape the validation procedure. Further, rigorous validation procedures are likely to include most, if not all, of the tests contained in the acceptance tests, and will be quite highly correlated to the acceptance tests.

Once in service, however, a fault that is undetected or has no effective alternate is potentially catastrophic. Since a typical program result will pass through several acceptance tests, and since several alternates may be available at various levels, a fault partially correlated to its acceptance tests or alternates may nevertheless be detected and recovered from. If recovery blocks are to substantially improve reliability, it must be because almost all of the residual faults left in the operational program are faults that are highly correlated to the validation procedures but uncorrelated or only partially correlated to the acceptance tests and alternates. Since the validation procedures and the acceptance tests are already highly correlated, it will be hard to argue that such faults predominate.

At the very high reliability levels required for safety critical flight-control functions, the existence of any faults in an operational program must be cause for the gravest concern. Moreover, it is almost impossible to estimate the failure rate from such faults. The recovery-block approach provides a possibility of recovery should such a fault generate an error, threatening system failure. However, this analysis shows that recovery blocks do not, by themselves, provide a reliability improvement sufficient to meet the stringent reliability requirements of flight-control applications. However, once an initial version of the programs (of adequate reliability) has been obtained, as suggested in Section 2.5 Recovery Blocks can be used to allow some modification and enhancement of programs without loss of reliability.

5 ACCEPTANCE TEST DESIGN

A problem that has been raised about recovery blocks is the cost of rigorous acceptance tests. Concern about the cost of acceptance tests has encouraged the view that acceptance tests should test for less than complete correctness, that "acceptability" might suffice. Since there is reason to believe that the quality of the acceptance tests is a primary determinant of system reliability, this inclination toward less rigorous acceptance tests is regrettable. But there must be a limit to the cost of the acceptance tests. That tests should cost only 10 percent of the main calculation, as has been suggested, is probably optimistic; it would be more realistic to expect that the test might, in general, be comparable in cost to the calculation it is to test.

There are, however, examples where it has been suggested that the tests would be unreasonably expensive relative to the calculation, thus justifying a compromise. For example, to test the insertion of an item into a sorted heap using the heap-sort algorithm costs on the order of the logarithm of the number of elements in the heap, and yet to check that the heap is still sorted, let alone that it still contains the appropriate elements, is much more expensive. This example is, we believe, representative of the cases in which the cost of the test is greater than the calculation, and exemplifies the characteristic common to such cases, that the test in question involves an invariant across an entire data structure, and thus requires the values of many more elements than

just those directly accessed in the calculation. We do not deny that invariant properties of data structures provide a powerful method of expressing the intent of the program and of checking for errors. However, repeated checking of the invariant is expensive, and we must seek some alternative that does not compromise the rigor of the acceptance test.

In the attached report, "An Approach to Specification Analysis," Flon describes a method of establishing that invariant properties of data structures are maintained. Flon uses a formal algebraic style of specification for the individual operations of the program, a style that would yield simple low-cost acceptance tests derived directly from the specifications. Algebraic specifications, however, do not readily lend themselves to the expression of the invariant properties that are so useful to human intuition. Flon demonstrates that these invariants can still be derived from the specifications, and that, if every operation on the data structure satisfies the specifications for that operation, then the sequence of operations must maintain the invariant. Specifications written in other formal specification languages, such as Special, have the same general properties as the algebraic specifications used by Flon, and are just as appropriate for this type of analysis.

Consequently, we believe that there should be little reason for acceptance tests to be significantly more expensive than the calculations they check. Rigorous acceptance tests ensuring correct behavior of each individual operation should suffice to guarantee the properties that would have been checked in the unacceptably expensive tests. But this depends critically on a requirement that each operation not only does what we require of it, but also does nothing else. The acceptance tests must check both that elements of the structure that should be changed have been changed to the correct values, and also that no other element has been changed. Current implementations of the cache mechanism could, but do not, provide this facility. It involves noting in the cache every changed element as its value is checked during the acceptance test and confirming at the end of the test that there are no further entries in the cache whose values have not been checked.

6 ON CACHING MECHANISMS FOR ASYNCHRONOUS MULTILEVEL SYSTEMS

This section discusses two possible mechanisms, in outline, for implementing a recovery cache in an asynchronous environment based on two alternative kinds of multilevel system:

- Fully typed systems without resource mapping.
- Fully typed systems with resource mapping.

Systems with resource mapping but without typing, or with inadequate typing, appear to present severe problems and the possible mechanisms for such systems are less attractive.

6.1 Fully Typed Systems Without Resource Mapping

The simplest is applicable to fully typed systems without resource mapping between nested environments, for instance a Campbell-Wyeth system. The typing is assumed to be sufficiently rigorous to ensure protection. Within any environment there will exist a

repertoire of types of type, objects and associated operations. One of these operations will be the accept operation, for that type. In the representation of the type, an additional field may be provided to contain a recovery-level number, as in our existing mechanisms.

During an operation on an object of the type, the interpretation procedure may decide that it is necessary to change the value of the object. It would then check the level number held in the additional field of the representation, and, if necessary, make a cache entry, containing at least a reference to the type (and hence to the accept and reverse procedures) and a reference to the object. It may also place there any additional information that may be needed by the accept or reverse procedures. The cache entry can be made into the cache of the process which requested the operation on the object. The cache is itself a typed object and thus assignments to the cache are made by appropriate cache operations which can ensure correct manipulation of the stack pointers, etc.

It should be noted that, if the interpretation procedure is itself a recovery block, then any manipulations it may perform on the object, including changing the level number and caching its value, are, of course, subject to an acceptance test, and are encached in the cache corresponding to the interpretation procedure while the operation is in progress. However, if the scope of the working space used for all subsequent nested environments is the same as the scope of the interpretation procedure, on exiting the recovery block of the interpretation procedure, all these manipulations will be manipulations of objects now local, and the corresponding cache entries can be discarded.

Because protection is assumed to be perfect, the interpretation procedure can place any information in the space of the user, and any information in the cache of the user, confident that it cannot be lost or corrupted. It is not obliged to, however, and could keep any information, including recovery information, in its own space. An important feature of this scheme concerns recovery following failure of an intermediate environment. When this intermediate environment launched a subsidiary environment, it did so by constricting an object of type "level", or "process", and performing upon it an operation "interpret". The "interpret" operation, being an operation which changes the value of the object on which it operates, will of course need to cache a reference to the level or process in the cache of that intermediate environment. Such references automatically ensure that, should the intermediate environment fail, then all subordinate environments are reversed out, and any privileged operations they may have invoked are undone.

This mechanism is elegant in that it uses exactly the same mechanism for all operations simple or complex, in a single environment or in many nested environments. The problem is that the initial precondition may not be satisfied. The typing mechanism may not be fully secure, and, if resource mapping between nested environments is provided, there is as yet no known method of making it secure.

6.2 Fully Typed Systems with Recursively Nested Resource Mapping

The second caching strategy is applicable to a fully typed system with resource mapping. Because of the resource mapping we must assume that all resources in environment i+1, including the cache, are exposed to its immediate ancestor, environment i, and must be discarded if environment i should fail. Consequently a highly privileged interpretive

procedure may keep no sensitive information in the space of a user or in the cache of a user. Indeed essentially the only information which can be allowed in a cache is the values of objects held in the working space of that environment, or any nested environment, and the names of types and objects namable by the user of that environment (although such objects might be represented only in a more privileged environment).

An essential feature of any scheme of this kind is the mapping table provided for each environment. This mapping table which is inaccessible from within the environment, maps the names of types and objects known within the environment into the corresponding names of these types and objects by which they are known to its immediate ancestor environment. If the environments are nested several levels deep, then the resolution of a name will require its translation through each of the intervening environments' mapping tables.

The caching mechanism appropriate under these conditions is similar to that above, but the interpretation procedures do not place sensitive information in the user's cache but must hide it in workspace of their own. Such sensitive information relates necessarily to some object for which a name has to be passed down to the user through the mapping tables. If the user requests an operation on such an object, provided to him by a more privileged environment, the recovery level number field, which is examined to determine whether caching is required, is located not in the representation of the object but in the descriptor for the object in the mapping table for his environment. Thus, if the object is changed by a request from within a particular environment, the mapping table for that environment will record the fact that the object has been cached, and the name of the object will be recorded in the cache.

If there are intermediate environments between the user's environment and the environment where the object is implemented, then any change to the object requested by the user is also implicitly a change requested by each of the intermediate environments. The appropriate caching and recovery-level recording must be performed for each of these intermediate environments. Thus recovery after failure of an intermediate environment is provided for. This feature is valuable; when a reverse operation is invoked, it is possible to confirm that the object is recorded as having been changed in the mapping tables of every intermediate environment.

Each interpretation procedure must have access to workspace within which to cache sensitive information, workspace unfortunately organized as a heap. Within this workspace the procedure maintains a stack, for each object, of the cached information about that object.

A complication arises about the caches of intermediate environments. If a user program satisfies its outermost acceptance test and terminates successfully, its own space and cache can be recovered. However the next enclosing environment is still left holding, in its cache, the changes to objects made by that user program (for instance, changes to files). This enclosing environment itself may not be able to terminate, for it may also be serving other users, but it would wish to remove from its cache, after appropriate checking and acceptance procedures, cache entries relating to objects required only for processes now terminated. Indeed it may be appropriate to remove many such objects from the mapping tables as well, and thus the operation for removing the object from the mapping table could also be required to remove it from the cache. Unfortunately this implies that the cache for environments must also necessarily become a heap.

This scheme is less elegant in that it imposes much greater burdens on the programmers of the interpretive procedures. But the scheme should be capable of being made rugged and is therefore probably more valuable than the first scheme.

7 RECOVERY IN THE PRESENCE OF ASYNCHRONOUS INTERACTION

The problem of error recovery in a context involving interacting asynchronous processes is one of the more striking problems of recovery blocks, or of any other error-recovery mechanism. The classic domino effect, whereby the failure of one process causes an endless chain of rollbacks, has been extensively investigated, and a number of possible approaches to its solution proposed. Without structure being imposed on the program, clearly the domino effect cannot be entirely precluded. The question becomes one of selecting a structure for the interaction between asynchronous processes that is not too onerous to the programmer, and that provides recovery without excessive costs for storage of recovery information or unreasonable amounts of rollbacks during recovery.

The alternative structures proposed include the conversation mechanism and various forms of recoverable monitors. Of these, the conversation technique has the great advantages of elegance and simplicity, but its severe effect on the structure of the program casts doubt on its use. It is certainly true that the structure would be radically changed by the constraints imposed by conversations and that the user would be required to think about the structure of his program quite differently; it is possible, however, that the resulting program would be just as well structured as before, and that its construction would be just as feasible.

The conversation concept is closely related to the atomic action concept much favored as a structure for the coordination of asynchronous processes. A conversation is an atomic action in which two or more processes concurrently participate, and in which a recovery block structure provides fault tolerance for its operations; similarly, a standard recovery block must itself be regarded as an atomic action of a single process. Its atomic nature is necessary to ensure that the results cannot be used before they have been checked by the acceptance test, and to ensure that the state restoration mechanism is not disrupted by, and does not disrupt, the action of some other task. The requirement that the conversation be an atomic action does not require simultaneity (though that is certainly the simplest implementation) nor does it require that all required data be locked immediately for the exclusive use of the action. Lomet describes some interesting possible variations; however, such variations are merely implementational optimizations and add nothing to the power of the approach.

A particular advantage of conversations over other structures is that conversations allow a greater degree of recovery from errors. This is because the various alternatives of conversations imply only that two or more processes should interact, and anticipate nothing about the form of that interaction. In contrast, a system based on buffered messages not only requires that the buffered message be correct so that it can be resubmitted to the process after recovery, but also that the format and intent of the message are implicitly appropriate. In many cases where some of the alternatives represent previous versions of the system, this will be a constraining limitation.

The feasibility of the atomic conversation, as a means of structuring recovery for interacting asynchronous processes, requires a demonstration of the technique applied to a realistic system. We are currently investigating the feasibility of such a demonstration.

Many designs for recovery in asynchronous systems require that the entire system stop for a period to establish the extent of the required recovery. In a fully distributed system, such a pause would be impossible, but, even in a single-processor system, it is not possible to stop all the high-priority real-time processes just because some background task requires recovery. However, it cannot be assumed that these high-priority tasks have not interacted with the task under recovery.

We have investigated two mechanisms for establishing what recovery is required in an asynchronous distributed system, keeping our assumptions about the constraints on the structure to a minimum. Each process must record the extent of its dependency on the results of other processes, noting the time (or some recovery-block number) of its interactions with each of the other processes, so that it can require the correct degree of rollback from other tasks and can calculate what rollbacks to perform should some other task fail. The resolution to which this information is recorded might be at the level of the process, or possibly it might be much finer, being associated with individual data structures.

The complications of rollback in asynchronous systems arise because of the propagation of rollbacks. This propagation can be regarded as being determined by the transitive closure of these dependencies; the two methods correspond either to maintaining that transitive closure continuously during the execution of the program, or to calculating it during recovery. Maintaining the transitive closure continuously involves, at each interaction with another process, taking note of not just the time of the interaction but rather all of the dependencies of the other process so they can be combined with the existing dependencies of the process. This could imply a rather substantial addition to each message between processes, but it allows a relatively rapid recovery from error.

The alternative approach involves three types of message between processes during recovery: Rollback, Ready, and Resume. The Rollback message is issued by the failing process, and by any process receiving a Rollback message with further dependencies that must be recovered. A process that has completed its rollback, and has not issued any Rollback messages of its own, can issue a Ready message back to each process that sent it a Rollback message, as can a process that has completed its rollback and has received a Ready message for every Rollback that it issued. When the process originating the recovery has received the requisite number of Ready messages, it can issue a Resume message to each process to which it sent a Rollback message, and those processes forward that Resume message to every process to which they sent a Rollback. On receipt of a Resume message, a process can resume processing. Clearly the price of not maintaining the full transitive closure of dependencies during normal operation is a much slower recovery when an error is detected.

8 PROVISION of RECOVERY BLOCKS on the BENDIX BDX930 PROCESSOR

Consideration has been given to the modification of the Bendix BDX930 processor microprogram to provide recovery blocks as an integral part of the instruction set. The objectives of this modification would be:

- To investigate the problems of recovery-block implementation in a realistic environment

- To investigate the magnitude of the recovery-block overhead attainable by relatively simple modification of existing hardware

- To demonstrate recovery blocks operating on an avionic computer with relatively little overhead

- To provide a vehicle for subsequent studies of the application of recovery blocks to aircraft flight control.

Alternative approaches to the experimental provision of recovery blocks would involve either software simulation, with very high run-time overhead, or special purpose hardware, with low overhead but high construction costs.

8.1 Modification of the Bendix BDX930

The processor proposed for this microprogram modification is the spare BDX930 processor for the SIFT configuration, a processor intended for error insertion testing and thus equipped with sockets for each of its integrated circuits. The BDX930 processor is a 16 bit computer intended for military and avionic use. It is constructed on a single, 6 inch-square multilayer card, with further cards being required for memory, input/output, real time clock, etc. The processor is built around 2901 4-bit arithmetic units, and is controlled by a 56-bit-wide microprogram. The microprogram is held in 7 PROM chips (54s472), giving 512 words of 56 bits, of which some 400 words are required for the standard instruction set, leaving about 100 words for modifications. The 54s472 is a fusible link device for which no EPROM is available. However the cost of a full set of 7 PROMs, programmed, is only about $100, and thus the cost of new PROM sets is not impossible. The 100 words of microprogram still available will allow some microprogram support for recovery blocks, but will constrain the exuberance of that support. Precisely how much additional support can be provided cannot be determined without performing the actual microprogram modifications.

The instruction set of the BDX930 provides primarily a range of 16 bit arithmetic and logical operations. There are 16 general-purpose registers of which only a few can be involved directly in indexing or store access. Some register-to-register instructions can be completed in a fraction of a microsecond but typical store reference instructions require 1.5 to 2 microseconds.

Floating-point instructions have been added with some difficulty to the BDX930 by microprogram, because the processor does not provide sufficient hardware to support them. However, the SIFT processors do not have floating-point microprogram, and the microprogram modifications envisaged for the provision of recovery blocks would probably interact with the floating-point facility. Thus floating-point and recovery blocks would be mutually exclusive.

Within the constraint of the available microprogram space, we envisage the following modifications of the BDX930 microprogram:

- Addition of a new instruction to indicate recovery block entry

- Addition of a new instruction to obtain the 'prior' cached value of a word of store

- Addition of new instructions to indicate success or failure of an acceptance test, with corresponding cache processing and state restoration

- Modification of the instructions (very few on the BDX930) that modify storage to cause the encaching of the prior values of storage

- Provision for the encaching of procedure calls to recovery procedures, for the calling of such procedures, and for returning from such procedures.

It is not considered that microprogram support for conversations between interacting asynchronous processes is appropriate on the 930. Should there be sufficient microprogram space available when the proposed modifications have been made, the most desirable further extension would be a facility to allow the acceptance test to determine that no additional modifications to storage have been made beyond those that it has already checked.

Software support will be required in the Pascal compiler to recognize the recovery-block and conversation components and to generate the appropriate special instructions and flow of control, and also to allocate storage for the cache. If the modified microprogram reserves general-purpose registers for its own use, the compiler must be modified to reduce the general-purpose registers available to its register allocation and optimization functions. This is a simple modification, but could seriously impact the efficiency of the compiled code. Careful design should ensure that existing linkage mechanisms suffice.

8.2 Performance and Resource Implications of Recovery Blocks

The aspects of recovery blocks that have the most impact on resource requirements and performance are:

- The space occupied by the cache,

- The space occupied by the flags attached to each word or group of words to indicate that they are already cached,

- The processor registers required by the modified microprogram to administer the cache and perform the encacheing operations,

- The time required to test whether a word to be modified has been already cached and, if not, to cache it,

- The time required to consolidate the cache on recovery block exit.

Even though recovery is presumably quite rare, the time required to restore the prior state for recovery may also be an important factor in the presence of real-time performance constraints.

One of the most important design decisions affecting these space and time overhead concerns the implementation of the already-cached flags. Several alternative approaches are available, each making a different compromise between the various costs. The

simplest algorithm involves typically four flag bits per word, a high space overhead. This can be reduced in several ways. First, by restricting the nesting of recovery blocks to three deep, which might suffice for limited applications, the overhead becomes only two bits per word. Secondly, if the unit of information encached is increased from single words to, say, groups of four words, flags are needed only for every fourth word rather than every word. The price, of course, is that the encacheing operations must encache all four words, which takes extra time and cache space. A third approach is to eliminate the flags entirely. The mechanisms are then unable to determine that data items had been already encached and are required to write the prior value of memory into the cache for every store operation. Provided that the programs execute only relatively few operations before returning to the global level outside all recovery blocks, the increase in cache size might be acceptable. For programs with a long-lived recovery block, the cache could become very large. Further, where the flags are available, they are used to prevent local data from being cached, thus avoiding unnecessary cache activity. Since assignments to local variables and temporaries are probably a majority of all assignments, in the absence of the flags, some other mechanism must be provided to recognize locals and inhibit their caching.

The essential processor registers for administration of the cache are:

- The current cache pointer,
- The cache limit,
- The current recovery-block depth.

If already-cached flags are not used, the recovery-block depth need not be held in a register, but a register may still be needed to permit the recognition of local variables and temporaries.

These registers can be obtained only by reservation of some of the programmer's general-purpose registers for the exclusive use of the recovery-block mechanisms. It appears that the Pascal compiler could be modified to refrain from using a pair of registers, at the cost of some loss of optimization. The 930, however, has rather limited facilities for accessing the memory and for address computation, but good interregister arithmetic operations. It thus depends quite heavily on the availability of registers and optimization to obtain reasonably efficient code. Since many of the registers are already assigned specific functions, the reservation of a further three registers for recovery blocks could have a significant effect on the efficiency of programs. The Pascal compiler uses relatively complex optimization algorithms, and thus it is hard to estimate how much performance would be lost without actually modifying the compiler and recompiling and running some sample programs.

The Bendix BDX930 has very few registers available to the microprogram aside from the 16 general-purpose registers provided to the programmer. It is notable that the floating-point microprogram must store and restore general-purpose registers in order to obtain registers in which to hold intermediate values during the arithmetic operations. Further, the 2901-arithmetic-unit integrated circuit has only simple shifting facilities, from which relatively complex addressing and masking operations must be constructed to access the already-cached flags. It appears that this is possible within the available registers, but at the price of repeatedly recomputing storage addresses because there is nowhere to save them while flags are examined. Moreover, using the shifting facilities of the 2901 to

access and manipulate already-cached flags is tedious and adds several microseconds to the time required to cache a word or group of words. A caching mechanism without flags appears to be relatively straightforward to microprogram.

9 A PROPOSAL FOR RECOVERY BLOCKS FOR FLIGHT CONTROL PROGRAMS ON THE BENDIX BDX930 PROCESSOR

The characteristics of flight-control programs allow a simplified recovery-block structure, which should be able to provide the required recovery capability with reasonable implementation and runtime costs. These characteristics are:

- The programs are structured as a number of relatively short and independent programs,
- The programs are executed relatively frequently and run to completion within a short period of time even when preempted by higher priority tasks,
- The programs use few local variables, and make few assignments to global variables other than the assignments to each of their results,
- A deep and complex recovery-block structure in a flight-control program would be unexpected.

The recovery-block structure proposed here has been designed to be readily implemented, by modification to the microprogram and the Pascal compiler, on the Bendix BDX930 processor. It provides reasonably efficient encaching and recovery mechanisms for flight-control programs, but the omission of already-cached flags would make the proposed mechanism unsuitable for many other applications. The design includes recovery procedures to restore the abstract state of a process rather than its concrete state. It could readily be extended to include conversations to allow recovery of interacting asynchronous processes, except that the Pascal compiler does not support asynchronous processes or their interaction at present. Thus full implementation of asynchrony and recovery in its presence would represent a substantial elaboration of the project.

In this design, a cache stack is provided for each process, where a process may correspond to a priority level of the scheduling tables, or to some operating system process. It is envisaged that the program main stack and the cache stack will be allocated in the same area of the store and will move toward each other from opposite ends of that area of store, thus obviating the need for registers to hold the stack or cache limits while still allowing recognition of stack overflow. The cache stack is managed by a register containing the address of the top of the cache stack.

Since the Pascal compiler allocates all local variables and temporaries on the stack, it is possible to recognize locals (to prevent their being cached) by retaining the value of the stack pointer on block-entry. Then all addresses between this block-entry pointer and the cache pointer must correspond to local variables. The block-entry pointer will be referred to frequently and must be allocated a register. The main stack word pointed to by the block-entry pointer is used to hold the recovery-block depth count. The proposed register allocations are:

- 10 -- block-entry pointer (removed from general-purpose use),
- 11 -- cache-top pointer (removed from general-purpose use),
- 12 & 13 -- function result value (not affected by cache proposal),
- 14 -- heap pointer (not affected by cache proposal),
- 15 -- main-stack pointer.

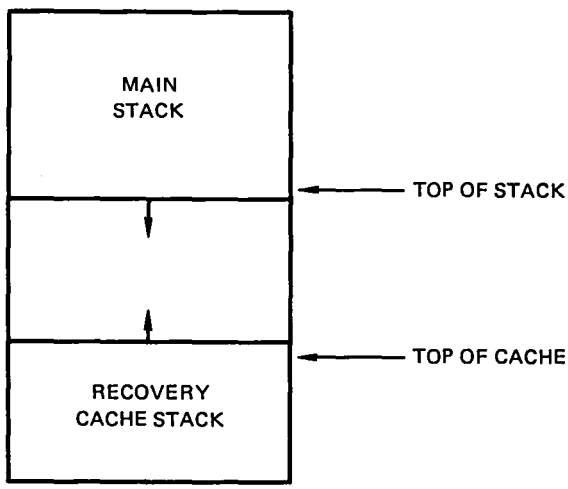

Figure 6: The Arrangement of the Main Stack and Cache Stack

9.1 Cache Stack Entries

Entries in the cache stack can be of three kinds:

- Assignment entries,
- Procedure-call entries,
- Block-marker entries.

There are two kinds of procedure-call entries:

- Recovery-procedure entries,
- Acceptance-procedure entries.

There are three kinds of block-marker entries:

- block-entry markers,
- null markers,
- a bottom-of-stack marker.

An assignment entry is two words: the address of the word and the prior value for that word. Since the BDX930 addresses only 32k of storage, the most significant bit of the address word must be zero; this is used to distinguish assignment entries from all other kinds of cache entries.

Recovery-procedure entries and acceptance-procedure entries have the same format, containing an arbitrary number of words. The first word has its most-significant-bit set and contains a flag field and a count field. The most-significant-bit set distinguishes these entries from assignment entries. The flag field distinguishes recovery- and acceptance-procedure entries from each other and from block-marker entries. The count field contains the number of parameters for the procedure and is two less than the number of words in the cache entry. Then follow the parameters to the procedure and, finally, the address of the procedure entry.

Block-entry markers, null markers, and the bottom-of-stack marker, also have very similar formats, each 18 words long. The first word has its most significant bit set and contains a flag field and a count field. The flag field distinguishes the entry from other kinds of cache entry, while the count field is always set to 16. The next 16 words are a set of 16 register values (by preserving all 16 registers we minimize the problems of interaction with the optimization algorithms of the compiler), while the last word is an instruction address. Only the first word of a null entry is significant, but all 18 words must be present.

The microprogram for each instruction that changes the value of a storage location must be modified to check the address of the word with the block-entry pointer (R10) and the cache pointer (R11). If the address does not lie between these two pointers, the word is not a local variable, and an assignment entry must be created in the cache. The assignment entry contains the address of the word and prior value of the word, and it is also necessary to increment the top-of-cache pointer checking for stack overflow.

9.2 Special Recovery Block Instructions

Five new instructions must be provided:

- Prior_value,
- Recovery_block_entry,
- Acceptance_test_passed,
- Acceptance_test_failed,
- Acceptance_procedure_return.

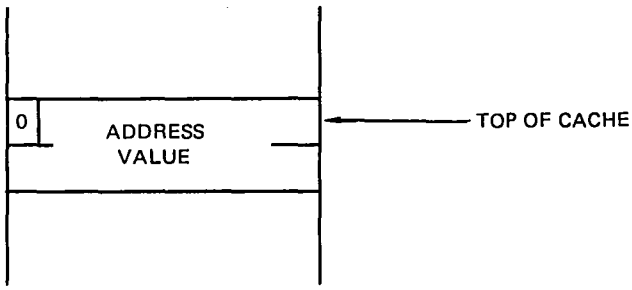

Figure 7: The Effect of a Store Assignment on the Cache

The prior_value instruction is intended for use by acceptance tests and allows the test to depend on the 'prior' value of a variable on entry to the recovery block as well as the current value at acceptance test time. The current value of the word is loaded into register 0 and a scan of the cache is started at the cache pointer (R11). As each cache entry is examined, if it is an assignment entry for the desired word, the prior value recorded in the entry is placed in R0. The scan stops when a block-entry marker is encountered and the instruction terminates. No register or storage location other than R0 is modified. The place at which the address of the desired word will be left for this instruction has still to be finally resolved, on the basis of possible compiler modifications, but it appears that the top-but-one word of the main stack is the most probable location.

The recovery_block_entry instruction places a block-entry marker on the cache. The contents of the 16 registers are copied into the cache entry and the address in the word immediately following the instruction is copied into the last word of the entry (this will be the address of the next alternate or, for the last alternate, of an acceptance_test_failed instruction). The recovery-block depth, pointed to by the block-entry pointer in R10, is incremented and stored on the top of the main stack, the new value of the main stack pointer being copied into the R10 as the new block-entry pointer. It is also necessary to increment the top-of-cache register and to check for stack overflow. Execution resumes at the location following the address of the alternate. This instruction is relatively simple, and it may be adequate for the Pascal compiler to provide this function by a sequence of ordinary instructions.

The acceptance_test_passed instruction scans down the cache stack looking for the first block-entry marker or for the bottom-of-stack marker. If it finds an acceptance-procedure entry, it invokes that procedure as described below. If it finds a block-entry marker, it sets that marker to a null, copies into the main-stack pointer and the block-entry pointer the corresponding values from the entry, but leaves the top of cache unchanged. If it finds the bottom-of-stack marker (which should not occur), the action is similar but the marker is left unchanged and the cache pointer is set to point to that marker. Other kinds of cache entry are ignored but the scan must, of course, allow for the various entries in the cache being of different lengths. Execution of the program resumes at the location following the instruction.

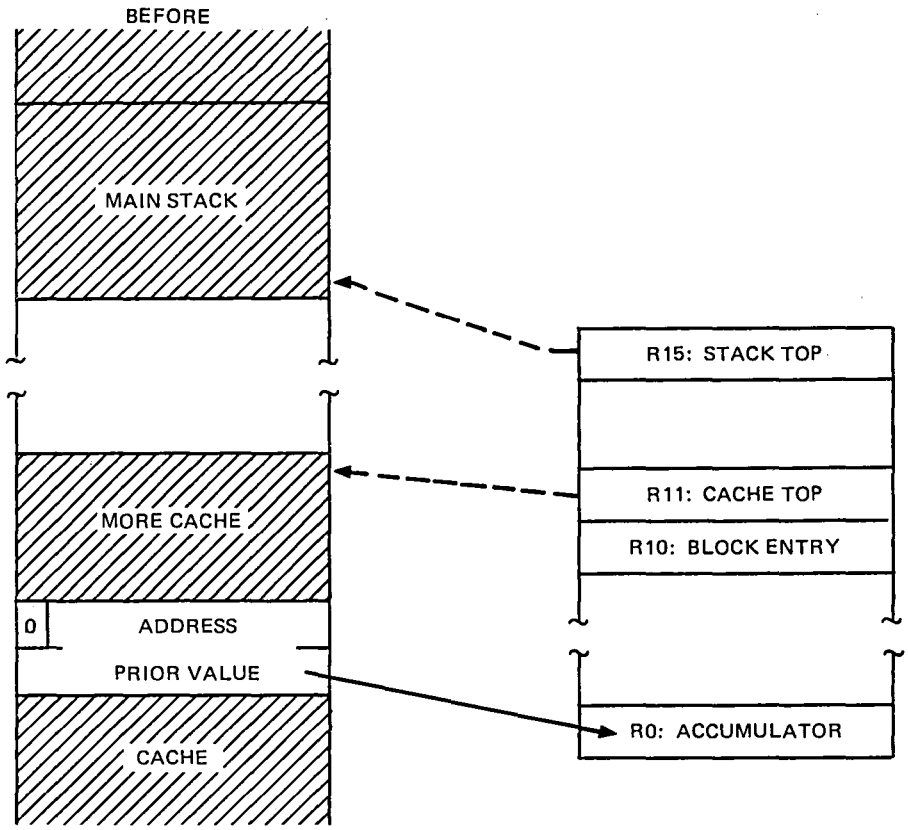

Figure 8: The Effect of the Prior_Value Instruction

If the acceptance-test-passed scan encounters an acceptance-procedure entry, the parameters in the entry are pushed into the main stack, as is the address of the cache entry beyond the procedure entry (the compiler will be allowing for a return link here, but acceptance procedures never exit normally) incrementing the main-stack pointer. The procedure addressed by the last word of the cache entry is executed.

The acceptance procedure returns by an acceptance_procedure_return instruction, which is identical to an acceptance_test_passed instruction except that the scan of the cache commences at the address held in the top of the main stack. Note that the parameters of the acceptance procedure are not properly removed from the main stack before the scan is resumed; this is not significant since they will remain there only until a block-entry marker is found and the main-stack pointer is reset.

The acceptance_test_failed instruction causes a recovery scan down the cache. If the scan encounters an assignment entry, the prior value recorded in the entry is assigned to the word addressed by the entry. If the scan encounters a null, it is ignored. In either case, the scan continues.

If the scan encounters a block-entry marker, the processor-stack control registers are

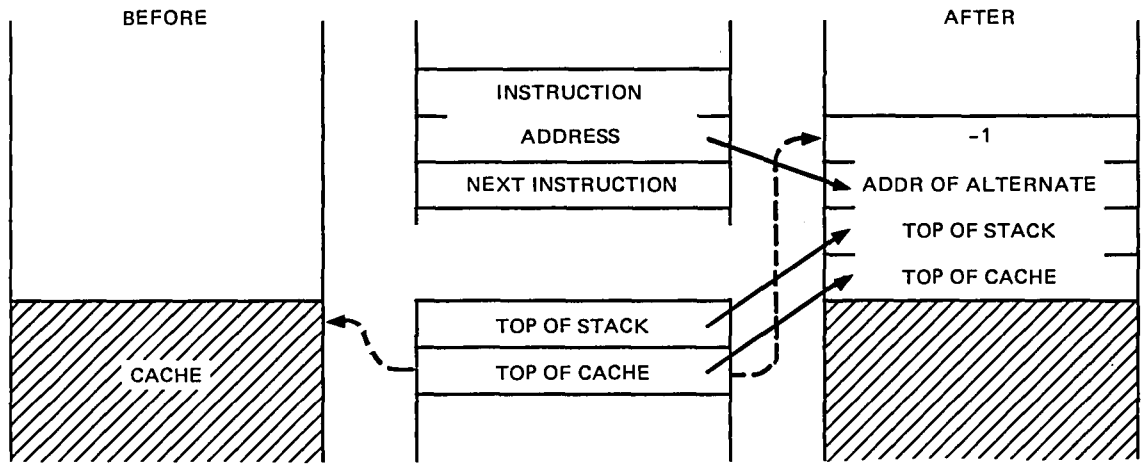

Figure 9: The Effect of a Recovery_Block_Entry Instruction

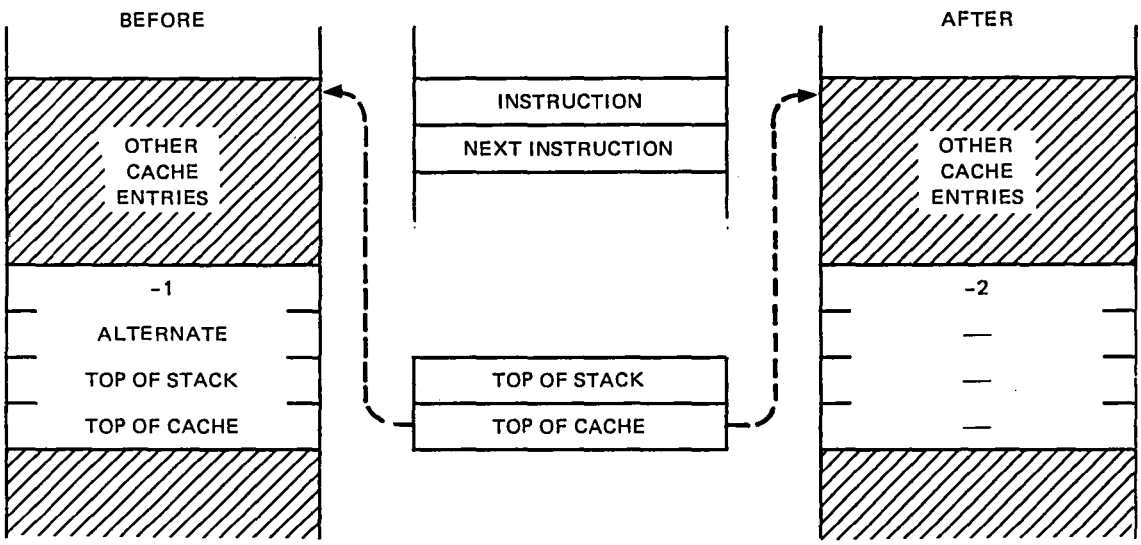

Figure 10: The Effect of an Acceptance_Test_Passed Instruction

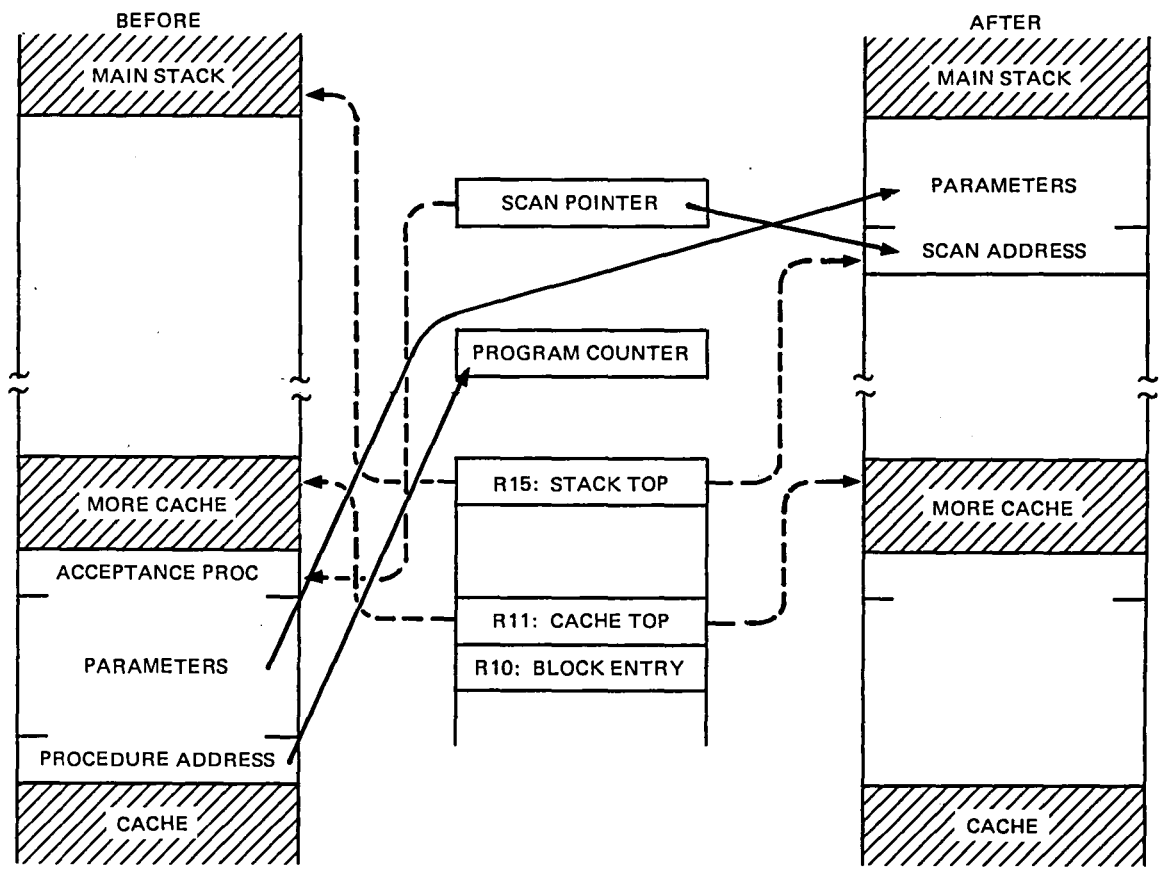

Figure 11: An Acceptance-Procedure Entry Operation

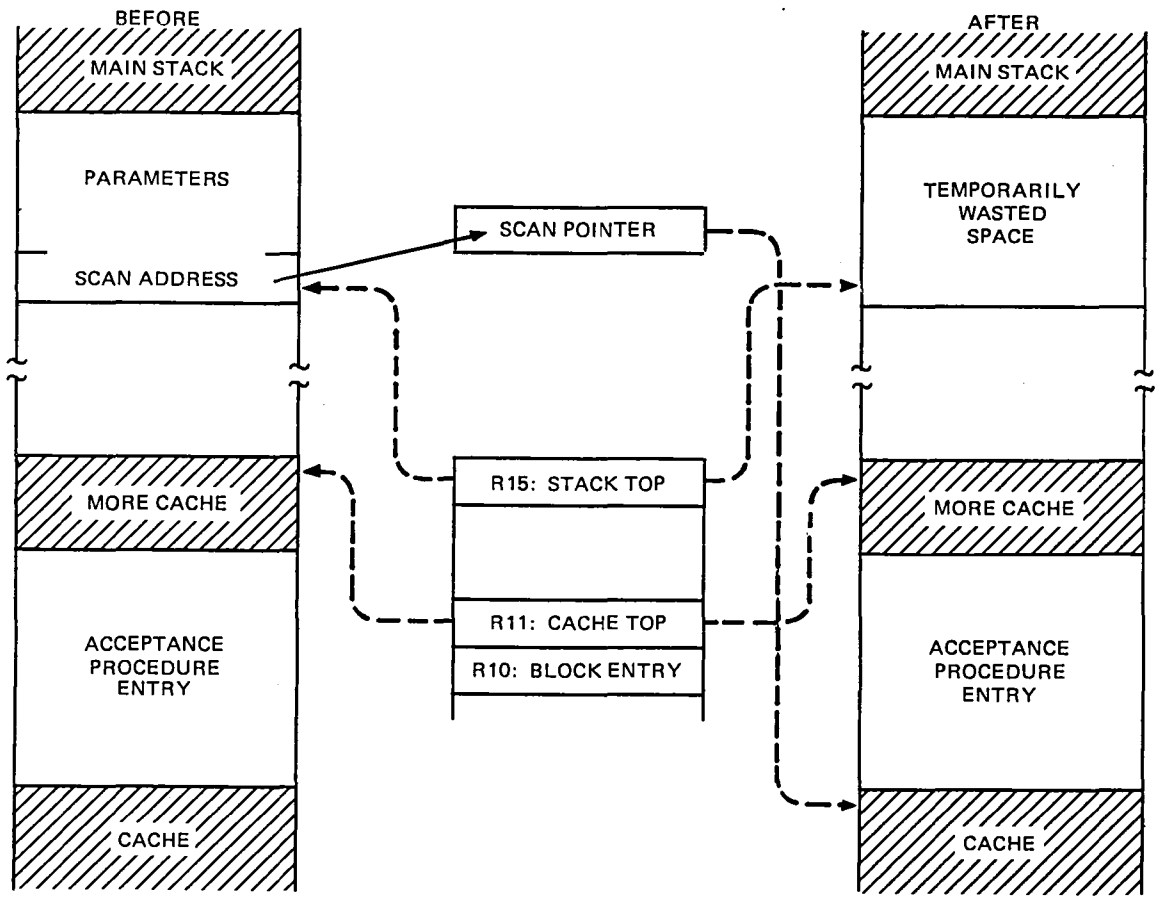

Figure 12: An Acceptance Procedure Exit

reset from the entry, and execution of the program resumes at the instruction addressed by the entry. If the scan encounters a bottom-of-stack marker (which would occur only if the outermost recovery block exhausts its set of alternates), the effect is equivalent to that for a block-entry marker except that the top of cache is set to that marker. The instruction address contained in the bottom-of-stack marker would presumably lead back to the executive with a failure indication.

If the scan encounters a recovery-procedure entry, the parameters in the entry are copied into the program's main stack (with one additional word as a dummy link), the top of cache is adjusted to below the entry, and a procedure call is made to the procedure address cited in the entry (with the most significant bit removed). The recovery procedure will return by using an acceptance_test failed instruction to resume the recovery scan. Again, the procedure parameters will be left on the main stack until a block-entry marker is encountered.

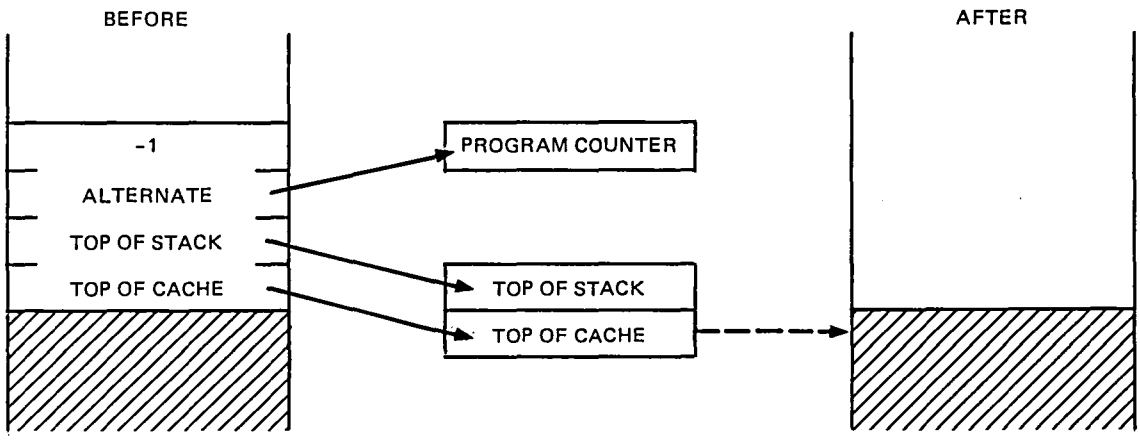

Figure 13: The Effect of a Block-entry Marker during a Recovery Scan

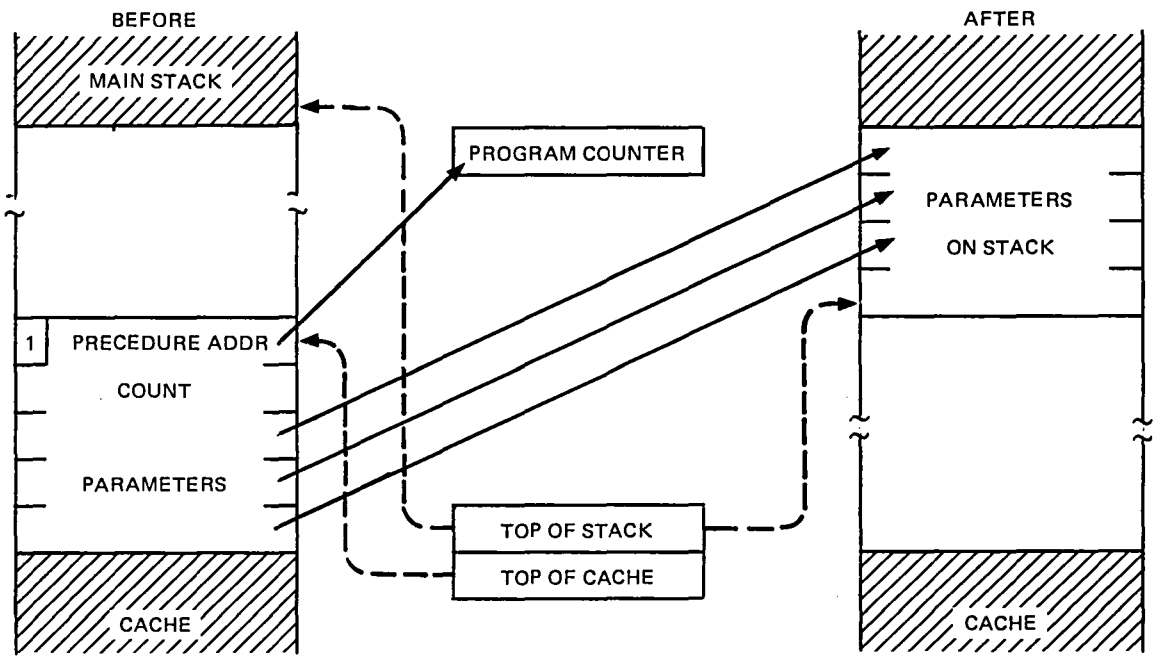

Figure 14: The Effect of a Recovery Procedure Entry during a Recovery Scan

9.3 Use of Recovery Block Instructions

Each alternate of a recovery block must start with a recovery block entry instruction, designating its next alternate. This is because the recovery scan following the acceptance_test_failed instruction eliminates the block-entry marker on the stack, thus allowing a further acceptance_test_failed instruction to initiate recovery of the enclosing recovery block. Block-entry markers are also specific to each alternate since they carry in them the address of their successor alternate. The last alternate of a recovery block cannot designate a successor alternate, but instead designates an acceptance_test_failed instruction.

```
ensure T by S1        generates          goto s1
else by S2                          t:  T;
else fail;                              if true then acc_test_passed;
                                                     goto e
                                                else s3: acc_test_failed;
                                    s1: rec_block_entry(addr_of s2);
                                        S1;
                                        goto t;
                                    s2: rec-block_entry(addr_of s3);
                                        S2;
                                        goto t;
                                     e:
```

Figure 15: The Code Generated for a Recovery Block

Thus far we have discussed procedures completely characterized by their assignments to variables. In the presence of interacting asynchronous processes, such a characterization does not suffice. In general it is impossible to restore the concrete state. Rather, recovery must be in terms of the restoration of the abstract state and special recovery procedures must be provided by the programmer to perform the abstract-state restoration. A typical example, requiring recovery procedures, involves the managing of some abstract resources. The procedures managing the resource must themselves generate and insert into the cache a recovery-procedure entry to invoke the recovery of the abstract state. It is then essential that the assignments they make to storage are not automatically undone during recovery. Thus these assignments cannot be left in the cache.

Two schemes are possible. One approach requires that the procedures of the resource manager temporarily substitute another cache stack for that of the user, restoring, on exit, the users cache complete with a recovery-procedure entry appended to it. Alternatively, and more probably, the procedures must, on entry, advance the top-of-cache pointer sufficiently to provide space for the recovery-procedure entry and also for an acceptance-procedure entry if required. On exit, it must adjust the top of cache to point to the recovery- or acceptance-procedure entry it has constructed in that space. Either approach allows the procedures of the resource manager to use recovery blocks internally, and have the cache entries for its assignments vanish on exit. Special code will be required to perform these manipulations of the cache pointer, to make assignments to the cache entries, and also to provide the recovery and acceptance

procedure addresses to be placed in those entries, but it would not appear that a special microprogram is justified. Special code will also be required to cause exit of the recovery and acceptance procedures by the appropriate special instructions, and also to allow the program to obtain the current value of the recovery-block depth or the prior value of a variable. In each case, it is envisaged that the required operations can be provided by an assembler procedure invoked by the Pascal source code, without imposing too much of a burden on the programmer.

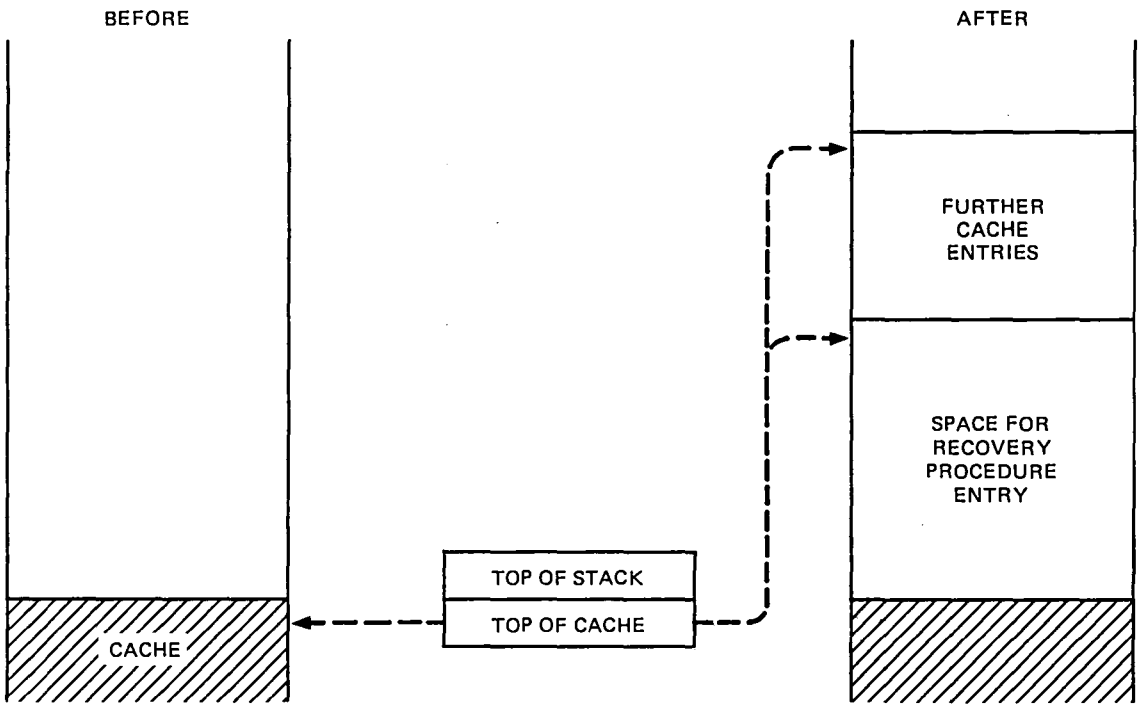

Figure 16: Cache Structures for a Resource-Management Procedure

Conversations provide recovery for interacting asynchronous processes by synchronizing entry to the conversation and also coordinating the results of the acceptance tests. Providing such synchronization and coordination should not be difficult, but the details depend on how asynchronous processes can be generated at all by the Pascal compiler. At present, the compiler makes no such provision.

More interesting is the problem of ensuring that the recovery operations of the several processes within the conversation avoid damaging interactions. Conceptually, the various processes within a conversation should share a single cache, but in practice this is not feasible. Consequently, if the various processes in a conversation have each made assignments to the same shared word, there will be assignment entries in each of their caches. Thus the recovery actions will restore prior values in some arbitrary order, and the value left in that word may be a value other than that on entry into the conversation.

The simplest solution to this problem appears to be to design the cooperating processes

so as to avoid assignments to the same word by different processes within the conversation. Failure of one process of a conversation should cause recovery of all processes. This can be arranged by inserting a specific recovery-procedure entry into the cache of each process of the conversation. The function of this recovery procedure is to force immediate recovery for all recovery blocks of the conversation.

10 A STORAGE MANAGER FOR A SIMPLE OPERATING SYSTEM

In this section we give an example of the use of recovery blocks in a multilevel asynchronous system with conversations. We choose as our example a fragment of an operating system, not because the techniques are especially suited to operating systems but rather because operating systems are widely understood and relatively easy to structure. The example is written in a dialect of Modula, augmented by a subrange type constructor and recovery block primitives. Ellipses and English text enclosed in { } are used where detail can be suppressed. This code has not been checked by a Modula compiler.

```
type FileId = ...;
type File = record name: FileId;
                   contents: UserBlockId end;


type ProcessId = (P1, P2, P3, PPS);
   (* each process that makes requests to the storage manager (the three
      user tasks and the print spooler) has a unique process identifier.
      process identifiers allow the storage manager and its attendant
      processes to ensure proper storage access.  *)


const maxValue = {the largest representable value};
type UnsignedInt = 0 : maxValue;


procedure sendNastyMessage ... end sendNastyMessage;
   (* used by recovery procedures to send the operator a message when
      it is too late to do more corrective recovery (for example,
      when an attempt to undo a print operation occurs after the file
      has been printed) *)
```

```
process UserTask1;
   use P1, spool, allocate, ..., UserTask2, UserTask3;


   (* DECLARATIONS FOR EACH PROCESS TO SUPPORT RECOVERY BLOCKS

      there is an instance of this set of declarations in each process,
      since each process conceptually has its own cache.

      all but "RBTypeOfProcessing" require support from the language
      interpreter.  the procedures are probably assembly language
      routines with access to implementation data structures.  *)

   procedure RBDepth
      : UnsignedInt;
      begin
      {return the current recovery block depth for this process}
      end RBDepth;

   type RBTypeOfProcessing = (Forward, Reverse, Accept);

   procedure RBDirection
      : RBTypeOfProcessing;
      begin
      {return the current direction of processing for this process}
      end RBDirection;

   type RBCacheEntry = {a variant record with a variant for each
         procedure that has effect other than by making assignments
         (and that can be undone)};
      (* note that this requires that all such procedures be known at
         the process level; otherwise, a more severe type breach is
         needed.  *)

   procedure RBMakeCacheEntry(ce: RBCacheEntry);
      begin
      {put an entry on the cache for this process}
      end RBMakeCacheEntry;


   ...
   begin
   ...
   spool(...);
   ...
   end UserTask1;
```

```
process UserTask2;
   use P2, spool, allocate, ..., UserTask1, UserTask3;

   procedure RBDepth ... end RBDepth;
   type RBTypeOfProcessing = (Forward, Reverse, Accept);
   procedure RBDirection ... end RBDirection;
   type RBCacheEntry = ... ;
   procedure RBMakeCacheEntry ... end RBMakeCacheEntry;

   ...
   begin ... end UserTask2;


process UserTask3
   use P3, spool, allocate, ..., UserTask1, UserTask2;

   procedure RBDepth ... end RBDepth;
   type RBTypeOfProcessing = (Forward, Reverse, Accept);
   procedure RBDirection ... end RBDirection;
   type RBCacheEntry = ... ;
   procedure RBMakeCacheEntry ... end RBMakeCacheEntry;

   ...
   begin ... end UserTask3;
```

```
interface module PrinterSpooler;
   define spool, despool;
   use File, ProcessId, UserBlockId, allocate, copyBlock,
      sendNastyMessage;

   (* "spool" and "despool" maintain a FIFO queue implemented as a
      circular buffer.

      successful undoing both "spool" and "despool" operation depends
      critically on "PrinterSpooler" being a monitor so that the
      operations properly maintain the monitor's data structures.

      the same array is also used to store despooled files where the
      "despool" has not yet been accepted.  there is no order on the
      files in this list.  *)


   const sBSize = 123;  (* to pick a size *)
   type BufferIndex = 0 : sBSize - 1;
   type BufferCount = 0 : sBSize;
   type FileEntry = record sf: File;
                           allocatedRBD: UnsignedInt end;

   var spoolBuffer: array BufferIndex of FileEntry;
      in, out: BufferIndex;
      spooledCount, despoolNotYetAcceptedCount: BufferCount;
      nonempty, nonfull: signal;
   (* implements a circular buffer of files with invariant:
            (spooledCount = (in - out) mod sBSize
               | spooledCount = sBSize)
         & (ALL i: 1 <= i <= spooledCount)
               INITIALIZED(spoolBuffer[(out+i-1) mod sBSize])
         & spooledCount + despoolNotYetAcceptedCount <= sBSize
      where INITIALIZED(x) is the condition that variable x has been
      assigned a legal value.

      the queue of spooled files occupies the "spooledCount" entries
      of "spoolBuffer" in the interval:
         [out, (in - 1) mod sBSize]

      the list of despooled files have that not yet been accepted
      occupies the "despoolNotYetAcceptedCount" entries of "spoolBuffer"
      in the interval:
         [in, (in + despoolNotYetAcceptedCount - 1) mod sBSize]

      whenever "spoolCount" is incremented, "nonempty" is sent out;
      whenever spoolCount + despoolNotYetAcceptedCount is decremented,
      "nonfull" is sent out.

      it is purely for convenience that the queue of spooled files and
```

```
the list of despooled files are next to each other.  *)
```

```
procedure swapBufferEntries(entry1, entry2: BufferIndex);
   begin
   {spoolBuffer[entry1], spoolBuffer[entry2]
      := spoolBuffer[entry2], spoolBuffer[entry1]}
   end swapBufferEntries;


procedure shiftBufferEntriesOver1(from, to: BufferIndex);
   begin
   {shift all entries in "spoolBuffer" from "from" to "to" one
    position to the right}
   end shiftBufferEntriesOver1;


procedure searchSpoolBuffer(f: File; start: BufferIndex;
      count: BufferCount)
   : BufferCount;
   (* search the spool buffer for file "f" in the "count" entries
      starting at "start".  return the index of the entry if
      found and return "sBSize" otherwise.  *)

   var delta: BufferCount;
   begin  (* do linear search since the spool buffer is unordered *)
   delta := 0;
   loop

      when delta = count do  (* file not there -- search fails *)
         searchSpoolBuffer := sBSize
         exit;

      when f = spoolBuffer[(start + delta) mod sBSize] do
            (* file found *)
         searchSpoolBuffer := (start + delta) mod sBSize
         exit;

      inc(delta)
      end
   end searchSpoolBuffer;
```

```
procedure spool(f: File);
   var ub: UserBlockId;
   var bc: BufferCount;

   begin
   case RBDirection of

      Forward:
         (* allocate storage for the file, copy the contents so the
            user process can continue, and put the file onto the
            circular buffer when there is room *)
         begin
         ub.PId := PPS;
         ub.blockSize := f.contents.blockSize;
         allocate(ub);

         copyBlock(ub, f.contents);  (* make a copy for spooling *)
            (* the recovery for this "copyblock" illustrates the
               difference between recovery actions at different
               levels.  the user block just allocated has no
               meaningful prior value.  thus, the reverse spool
               operation below need not worry about restoring its
               abstract value.  its current concrete value will,
               however, be dutifully restored should this block be
               undone because the recovery scan can't distinguish
               which writes are needed.  note, in contrast, that it
               is important that the lower level recovery do a
               reverse allocate.  *)

            (* wait until there is room in the buffer *)
         if spooledCount + despoolNotYetAcceptedCount = sBSize
               (* since spooledCount + despoolNotYetAcceptedCount is
                  only increased in this routine, and then only
                  incremented by 1, it is sufficient to test for
                  equality here *)
            then
            wait(nonfull)
            end;

            (* make room for the new file to be spooled by moving
               the first despooled file to the end of the list *)
         spoolBuffer[(in + despoolNotYetAcceptedCount) mod sBSize]
            := spoolBuffer[in];

         spoolBuffer[in].sf.name := f.name;
         spoolBuffer[in].sf.contents := ub;
         spoolBuffer[in].allocatedRBD := RBDepth;

         RBMakeCacheEntry('spool', f);
```

```
        in := (in + 1) mod sBSize;
        inc(spooledCount);
        send(nonempty)
        end; (* Forward *)

    Reverse:
        (* undo a "spool" operation: if the file is still there,
           remove it from the spool buffer; if not, notify the
           operator.  *)
        begin
        bc := searchSpoolBuffer(f, out, spooledCount);
        if bc = sBSize then  (* file wasn't there *)
           sendNastyMessage("too late to unspool file", f.name)
        else  (* file found, so remove from buffer *)
           shiftBufferEntries(out, (bc - 1) mod sBSize);
           out := (out + 1) mod sBSize;
           dec(spooledCount);
           send(nonfull)
           end
        (* the reverse allocation will be done underneath us *)
        end; (* Reverse *)

    Accept:
        begin
        bc := searchSpoolBuffer(f, out,
           spooledCount + despoolNotYetAcceptedCount);
        if bc < sBSize then (* not too late *)
           if RBDepth < spoolBuffer[bc].allocatedRBD then
              spoolBuffer[bc].allocatedRBD := RBDepth
           (* if RBDepth = allocatedRBD, no action required *)
              end
           end
        end (* Accept *)
    end (* case *)
end Spool;
```

```
procedure despool(var f: File);
      (* "despool" is a procedure with a write-only parameter
         instead of a function so that the file will be part of the
         cache entry *)
   var bc: BufferCount;
   begin
   case RBDirection of

      Forward:
         (* "despool" gives the next file to its calling process and
            moves it to the despooled list.

            "despool" assumes that the consumer process deallocates
            the storage associated with the file despooled.
            "despool" does not itself deallocate the storage since
            the consumer process presumably needs the use of the
            storage.  *)
         begin
            (* wait until there's a file to give out *)
         if spooledCount = 0 then wait(nonempty) end;

         f := spoolBuffer[out];
         RBmakeCacheEntry('despool', f);

         spoolBuffer[(in + despoolNotYetAcceptedCount) mod sBSize]
            := spoolBuffer[out];
         out := out + 1) mod sBSize;
         dec(spooledCount);
         inc(despoolNotYetAcceptedCount)
            (* since no room is made in the spool buffer, "despool"
               does not send out "nonfull" *)
         end; (* Forward *)

      Reverse:
         (* undo a "despool": put the file back on the spool
            buffer *)
         begin
         bc := searchSpoolBuffer(f, in, despoolNotYetAcceptedCount);
         swapBufferEntries(in, bc);
           (* this is where having the spool queue and the despooled
              list next to each other pays off *)
         in := (in + 1) mod sBSize;
         dec(despoolNotYetAcceptedCount);
         inc(spooledCount);
            (* like "spool", reverse despool sends out "nonempty" *)
         send(nonempty)
         end; (* Reverse *)

      Accept:
         begin
```

```
            bc := searchSpoolBuffer(f, in, despoolNotYetAcceptedCount);
            if RBDepth = spoolBuffer[bc].allocatedRBD then
               swapBufferEntries(bi,
                 (in + despoolNotYetAcceptedCount -1) mod sBSize);
               dec(despoolNotYetAcceptedCount);
               send(nonfull)
               end
            end (* Accept *)
         end (* case *)
      end despool;


begin  (* initialization *)
spooledCount := 0;
despoolNotYetAcceptedCount := 0;
in := 0;
out := 0

end PrinterSpooler;
```

```
(* "Printer1" and "Printer2" are print demons: they are continually
   running processes that keep despooling and printing files,
   deallocating the associated storage when they're done *)

process Printer1;
   use File, UserBlockId, PPS, read, UserAddress, despool, deallocate;
   var f: File;

   procedure print(f: File) ... end print;

   begin
   loop (* forever *)
      despool(f);
      print(f);
      deallocate(f.contents)
      end
   end Printer1;


process Printer2;
   use File, UserBlockId, PPS, read, UserAddress, despool, deallocate;
   var f: File;

   procedure print(f: File) ... end print;

   begin
   loop (* forever *)
      despool(f);
      print(f);
      deallocate(f.contents)
      end
   end Printer2;
```

```
module StorageManager;

   define allocate, free, read, write, copyBlock,
      Address, UserAddress, StorageUnit, UserBlockId;

   use ProcessId, store1Read, store2Read, store3Read,
      store1Write, store2Write, store3Write,
      lastStore1Address, lastStore2Address, lastStore3Address;

   (* the storage manager maintains a user address space, which
      conceptually is a linearly addressable array of type:
         array 0 : maxAddress of StorageUnit
      operations provided to manipulate this array are "allocate",
      "free", "read", "write", and "copyBlock".

      the user can allocate variable-sized blocks.  a user block
      identifier consists of
        (1) the process identifier for the owning process
        (2) a user address where the block starts
        (3) the block size
      "blockSize" is the amount of storage requested by the user.
      if "allocate" cannot allocate that amount of storage, it will
      reset "blockSize" to 0; if it can allocate the storage, it
      assigns a "start" address.
      (the block size could instead have been made part of the
      identifier for each store block.  making it part of the user
      block identifier collects all the information the user cares about
      in one place.)

      this user address space is implemented using many stores ("many"
      in this case is 3).  a store is also a linearly addressable array:
         array 0 : lastStoreiAddress of StorageUnit
      where "lastStoreiAddress" is a constant exported by store i.

      each user block is implemented as contiguous storage on one store.
      corresponding to each user address is a store address
      consisting of
        (1) the store on which the block is allocated
        (2) the store address where the block starts

      the mapping between user and store addresses is maintained in
      the storage map.  a storage map entry consists of
        (1) a user block identifier
        (2) a store address
        (3) the recovery block depth at which the user block was
            allocated
        (4) a pointer to the next storage map entry in the storage list
            (see below)

      all storage is either:
```

```
      (1) available,
      (2) allocated, or
      (3) freed and not yet accepted (referred to below simply as
          "freed")

   the allocation status (available, allocated, or freed) of a
   storage entry is defined by which storage list the storage entry
   is on.

   all storage lists are contained in the same storage map, which is
   an array of linkable storage map entries.  storage lists are
   maintained as ordered linked lists, largely to make them
   straightforward to implement.  (greater realism only improves
   efficiency; our real concern is how data is shared and used.)

   since user blocks can be of variable length, "allocate" and "free"
   maintain these lists with something like a buddy system
   reservation scheme.

   during allocation, an available block may be split into two
   blocks: one of the requested size and the other for the remaining
   storage still available.  similarly, blocks may be merged when
   a block is freed (to be precise, when it is accepted and becomes
   available).  since the number of blocks in use changes, a list is
   kept of unused storage map entries.
   (we will ignore the possibility that "allocate" may exhaust the
   supply of unused storage map entries.)

   the organization of the available storage list depends on the
   storage allocation scheme used.

   care must be used in assigning user addresses so that a user
   address is not reused until it is not only free, but has also been
   accepted (i.e., returned to the available storage list).  *)


const maxAddress = maxValue;

   (* types exported by the storage manager *)
type Address = 0 : maxAddress;
type UserAddress = Address;
type StorageUnit = {blank storage};
type UserBlockId = record pId: ProcessId;
                          start: UserAddress;
                          blockSize: UnsignedInt end;


const #ofStores = 3;
type Store = (S1, S2, S3);
type StoreAddress = record sId: Store;
```

```
                              addr: Address end;
   (* this assumes
         (ALL i: 1 <= i <= #ofStores)
            (lastStore:iAddress <= maxAddress)                    *)


   (* the storage map *)
const #sMaps = 321;  (* to pick a size *)
type SMIndex = 1 : #sMaps;
type SMPtr =   0 : #sMaps;
const nullSMPtr = 0;
type SMap = record ubId: UserBlockId;
                   sa: StoreAddress;
                   allocatedRBD: UnsignedInt;
                   next: SMPtr end;
var storageMap: array SMIndex of SMap;
   availableBlocks, allocatedBlocks, freedBlocks: SMPtr;
   nextUnusedSMap: SMPtr;
```

```
procedure associatedStorageEntry(ub: UserBlockId)
   : SMPtr;
   begin
   {return the pointer to the storage entry for the block with the
    given user block identifier; if no such user block identifier,
    return "nullSMPtr"}
   end associatedStorageEntry;


procedure translateAddress(ua: UserAddress)
   : StoreAddress;
   begin
   {return the store address for this user address}
   end translateAddress;


procedure legalAccess?(p: ProcessId; ua: UserAddress)
   : Boolean;
   begin
   legalAccess? := {does this user address belong to this process?}
   end legalAccess?;


procedure onList?(ub: UserBlockId; storageList: SMPtr)
   : Boolean;
   begin
   onList? := {is the user block on the specified storage list?}
      (* in checking the allocated and freed lists, an exact match
         must be found.  in checking the available list, the user
         block need only be contained in an available block.  *)
   end onList?;

procedure moveStorageMap(sMapToBeMoved, FromList, ToList: SMPtr);
   begin
   {move the specified store entry from one storage list to another}
      (* moving a storage entry to the available list may involve
         merging entries.  *)
   end moveStorageMap;


procedure assignUserAddress(ub: UserBlockId);
   begin
   {based on ub.blockSize, set ub.start to a user address for the
    block just acquired}
   end assignUserAddress;
```

```
procedure allocate(var ub: UserBlockId);
   begin
   var smi: SMIndex;
   case RBDirection of

      Forward:
         begin
         {search the available storage list for a big enough block};
            (* how this search is done defines the storage
               allocation scheme *)
         if {no block big enough} then
            ub.blockSize := 0    (* reset the user block to size 0 *)
         else
            smi := {storage entry of block found};
            if ub.blockSize < storageMap[smi].ubId.blockSize then
               {create new storage entry for unneeded storage};
               storageMap[smi].ubId.blockSize := ub.blockSize
               end;

               (* tell the user where his storage starts *)
            assignUserAddress(ub);

               (* set up storage map entry *)
            with storageMap[smi] do
               ubId := ub;
                  (* save recovery block depth at which block is
                     allocated *)
               allocatedRBD := RBDepth;
               moveStorageMap(smi, availableBlocks, allocatedBlocks)
               end;

            RBMakeCacheEntry('allocate', ub)
            end
         end; (* Forward *)

      Reverse:
         (* at the time of doing the reverse allocate, the block
            can either be allocated or available: if it had been
            freed, a previously executed reverse free will have
            re-allocated the block *)
         begin
         if onList?(ub, allocatedBlocks) then
            moveStorageMap(associatedStorageEntry(ub),
               allocatedBlocks, availableBlocks)
         else (* it's too late to do anything because the block has
                 been returned to available storage *)
            error
            end
         end; (* Reverse *)
```

```
   Accept:
      begin
      if onList?(ub, allocatedBlocks) then
         smi := associatedStorageEntry(ub);
         if RBDepth < storageMap[smi].allocatedRBD then
            storageMap[smi].allocatedRBD :=  RBDepth
               (* if RBDepth = allocatedRBD, no action is
                  required *)
            end
      else
         error  (* this shouldn't occur *)
         end
      end (* Accept *)
   end (* case *)
end allocate;
```

```
procedure free(ub: UserBlockId);
   var smi: SMIndex;
   begin
   case RBDirection of

      Forward:
         begin
         moveStorageEntry(associatedStorageEntry(ub),
            allocatedBlocks, freedBlocks);
         RBMakeCacheEntry('free', ub)
         end; (* Forward *)

      Reverse:
         begin
         if onList?(ub, freedBlocks) then
            moveStorageEntry(associatedStorageEntry(ub),
               freedBlocks, allocatedBlocks)
            end
         end; (* Reverse *)

      Accept:
         begin
         smi := associatedStorageEntry(ub);
         if RBDepth = storageMap[smi].allocatedRBD then
            moveStorageEntry(smi, freedBlocks, availableBlocks)
            end
         end (* Accept *)
      end (* case *)
   end free;
```

```
(* "read" and "write" deal with storage units, ignoring (i.e.,
   coercing) the types of values read and written.  *)


procedure read(ub: UserBlockId; ua: UserAddress)
   : StorageUnit;
   (* since "read" has no side-effects, no recovery operations are
      needed.  *)
   var sa: StoreAddress;
   begin
   {check that this user can legally access this address};
   sa := translateAddress(ua);
      (* read the storage unit on the appropriate store *)
   case sa.sId of
      S1:
         begin
         read := store1Read(sa.addr)
         end; (* S1 *)
      S2:
         begin
         read := store2Read(sa.addr);
         end; (* S2 *)
      S3:
         begin
         read := store3Read(sa.addr)
         end (* S3 *)
      end (* case *)
   end read;
```

```
procedure write(ub: UserBlockId; ua: UserAddress;
      su: StorageUnit);

   procedure sWrite(ua: UserAddress; su: StorageUnit);
      (* "sWrite" does the real work, writing the storage unit on
         the appropriate store *)
      var sa: StoreAddress;
      begin
      sa := translateAddress(ua);
      case sa.sId of
         S1:
            begin
            store1Write(sa.addr, su)
            end; (* S1 *)
         S2:
            begin
            store2Write(sa.addr, su)
            end; (* S2 *)
         S3:
            begin
            store3Write(sa.addr, su)
            end (* S3 *)
         end (* case *)
      end sWrite;

   begin
   case RBDirection of

      Forward:
         begin
         {check that this user can legally access this address};

         if onList?(ub, freedBlocks) then
            error  (* scream if no longer allocated *)

         else  (* encache the previous value *)
            RBMakeCacheEntry('write', ub, ua, read(ub, ua));

            sWrite(ua, su)  (* do the write *)
            end
         end; (* Forward *)

      Reverse:
         begin
         if onList?(ub, freedBlocks) then
            error
         else  (* restore prior value *)
            sWrite(ua, su)
            end (* if *)
         end; (* Reverse *)
```

```
    Accept:
        begin
            (* no action required *)
        end (* Accept *)
    end (* case *)
end write;
```

```
procedure copyBlock(from, to: UserBlockId);
   (* copy the contents of the "from" block to the "to", using
      "read" and "write".

      since the only effects of "copyBlock" on the outside world are
      achieved using a procedure within this module with its own
      recovery actions ("write"), no recovery actions for "copyBlock"
      itself are required.  *)

   var uaFrom, uaTo: UserAddress;
   begin
   uaFrom := from.start;
   uaTo := to.start;
   while uaFrom < from.start + from.blockSize do
      write(to, uaTo, read(from, uaFrom));
      inc(uaFrom);
      inc(uaTo)
      end
   end copyBlock;


begin  (* initialization *)
   (* each store is initially all available *)
availableBlocks := 1;
storageMap[1].sa.sId := S1;
storageMap[1].sa.addr := 0;
storageMap[1].ubId.blockSize := lastStore1Address + 1;
storageMap[1].next := 2;
storageMap[2].sa.sId := S2;
storageMap[2].sa.addr := 0;
storageMap[2].ubId.blockSize := lastStore2Address + 1;
storageMap[2].next := 3;
storageMap[3].sa.sId := S3;
storageMap[3].sa.addr := 0;
storageMap[3].ubId.blockSize := lastStore3Address + 1;
storageMap[3].next := nullSMPtr;

   (* all the remaining storage entries are unused *)
nextUnusedSMap := 4;  (* use this as a loop variable for now *)
while nextUnusedSMap < #sMaps do  (* initialize linked list *)
   storageMap[nextUnusedSMap].next := nextUnusedSMap + 1;
   inc(nextUnusedSMap)
   end;
storageMap[#sMaps].next := nullSMap;
nextUnusedSMap := 4;  (* now initialize it *)

   (* initially no storage allocated (or freed) *)
allocatedBlocks := nullSMap;
freedBlocks := nullSMap
end StorageManager;
```

```
interface module Store1;
   define lastStore1Address, store1Read, store1Write, store1Status;
   use StorageUnit, Address;

   const lastStore1Address = 65432;  (* to pick a number *)

   procedure store1read(addr: Address)
      : StorageUnit;
      begin ... end store1read;

   procedure store1Write(addr: Address; contents: StorageUnit);
      begin ... end store1Write;

   procedure store1status ... end store1status;

   ...
   begin  (* initialization *)
   ...
   end Store1;


   (* this example could be made more interesting by putting two stores
      on the same multiplexor *)


interface module Store2;
   define store2Read, store2Write, lastStore2Address;
   ...
   begin ... end Store2;


interface module Store3;
   define store3Read, store3Write, lastStore3Address;
   ...
   begin ... end Store3;
```

11 CONCLUSIONS

First, we have concluded that the effects of correlated design errors on reliability, and the limited effectiveness of reliability blocks against correlated errors, preclude the recovery block technique from achieving the degree of reliability required by NASA for flight control systems. It appears to us that the techniques of formal mathematical program verification will be needed to meet that reliability requirement. However, it may be possible to employ recovery blocks in addition to formal verification, for instance, by use of verified acceptance tests and second alternates to protect against errors in unverified first alternates.

We have concluded that it is possible to formally specify software systems and employ

those specifications to construct vigorous acceptance tests of reasonable cost, and that it is possible to provide recovery block mechanisms that match modern program structuring techniques such as types and processes. We have not, however, been able to find any completely satisfactory approach to the recovery of systems of interacting asynchronous processes. The most suitable technique, that of "conversations" imposes severe structuring constraints on the design of the application.

Lastly, we have been able to demonstrate that the special characteristics of flight control programs permit certain simplifications in the recovery block mechanisms, simplifications that would permit the implementation of a reasonably efficient mechanism on an aircraft flight control computer, such as the Bendix BDX 930, using only microprogram. This would permit the provision of recovery blocks for flight control programs at a relatively modest hardware cost.

References

[1] Avizienis, A.
Fault-Tolerance: The Survival Attribute of Digital Systems.
In *Proceedings of the IEEE, 66(10)*, pages 1109-1125. IEEE, Oct, 1978.

[2] Davies, C.T.
Recovery Semantics for a DB/DC System.
In *Proceedings of the 1973 ACM National Conference.* ACM, 1973.

[3] Hopkins, A.L., Jr., Smith, T.B., III, and Lala, J.H.
FTMP--A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft.
In *Proceedings of the IEEE*, pages 1221:1239. IEEE, Oct, 1978.

[4] Horning, J.J., et al.
Lecture Notes in Computer Science #16 Operating Systems: A Program Structure for Error Detection and Recovery.
Springer-Verlag, 1974.

[5] Wensley, J.H., et al.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
In *Proceedings of the IEEE 66(10)*, pages 1240-1255. IEEE, Oct, 1978.
Also in: Siewiorek, D.P. and Swarz, R.S., The Theory and Practice of Reliable System Design, Digital Press, 1982.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
172122		

4. Title and Subtitle	5. Report Date
Development of Software Fault-Tolerance Techniques	June 1983
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Peter Michael Melliar-Smith	SRI Project 7614
	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	NAS1-15480
	13. Type of Report and Period Covered Final Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665	

15. Supplementary Notes

16. Abstract

As computers become more widely used, and in particular as they become used in more safety critical applications, the reliability of the computer system and its software becomes more important. There is also an increasing need for high levels of reliability in applications involving very large numbers of inexpensive units where recall of the units would be disproportionately expensive. This report considers the nature of faults and the assumptions made by different approaches to correct operation. It then describes the recovery block approach and provides a probabilistic analysis of its effectiveness, with and without correlated design errors. Mechanisms for generating acceptance tests from specifications, and for providing recovery in the presence of asynchrony, are described. An analysis of, and design for, the provision of recovery blocks in the microprogram of the Bendix BDX930 processor is provided. The report concludes with an example of the use of recovery blocks in a simple operating system.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Reliability, software reliability, fault-tolerance, recovery blocks, error recovery	Unclassified--Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	65	

For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document