Abstract:

The paper presents a methodology for programming using recovery blocks – a program structure for fault-tolerant programming. A few programs with recovery blocks were written in Sequential Pascal (extended with recovery blocks) and performance measurements were taken; the data from these measurements is presented.

(A short version of this paper is to be presented at the 8th International Symposium on Fault Tolerant Computing, France, June 1978)

Fault Tolerant Sequential Programming using Recovery Blocks

By

S.K. Shrivastava and A.A. Akipelu

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

© 1978 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
SHRIVASTAVA, Santosh Kumar.


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 122)

Abstract

The paper presents a methodology for programming using recovery blocks—a program structure for fault-tolerant programming. A few programs with recovery blocks were written in Sequential Pascal (extended with recovery blocks) and performance measurements were taken; the data from these measurements is presented.

About the author

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a lecturer. Prior to that he was with the Plessey Co. Ltd.

Mr. Akinpelu is at the University College of London, where he is studying for a Ph.D. in Computer Science. He obtained his M.Sc. degree in Computer Software and Systems Design from the University of Newcastle upon Tyne in 1977.
1. Introduction

Fault-tolerant software possesses the property of producing acceptable results despite faults in programs or in the system that supports them. The importance of such software arises from the realization that complex computer systems cannot be guaranteed to be free from faults - a system might contain undetected design faults in hardware and/or software. Many of the current efforts in the area of reliable system design have therefore been directed towards the investigation of techniques and methodologies for designing fault-tolerant software [1,2,3,4]. This paper discusses a methodology of programming using recovery blocks - a program structure developed for fault-tolerant programming - and presents some results from experimental work carried out by the authors to evaluate the performance of programs with recovery blocks.

In this paper we shall only be concerned with the fault-tolerant aspects of a single sequential process. Frequently, when writing systems software (e.g. an operating system), one is faced with the task of programming for concurrent processes. While the topic of fault-tolerant concurrent programming is briefly discussed towards the end of this paper, a detailed discussion of it is beyond the scope of the paper. The rest of the paper is structured as follows: section 2 contains a brief discussion on recovery blocks and the associated recovery cache; section 3 presents a methodology for designing programs employing recovery blocks; a few realistic programs were constructed according to this methodology and in section 4 we present some data from the measurements taken to evaluate their performance; finally, the conclusions drawn from this work are described in section 5 where we also discuss directions for future work.

2. Recovery Blocks

A computer system can contain residual faults both in hardware and software. Therefore, when an error is detected while executing a program, it is not always possible to determine the sources of the fault. A commonly used error recovery technique is to regularly save the state of the computation (such a state saving is usually termed 'taking a checkpoint'); when an error is detected, the computation is 'rolled back' to the most recent checkpoint and the computation is restarted in the hope of avoiding the fault. This form
of 'backward error recovery' can be hardware assisted - an example of a computer architecture providing such recovery facilities is the COPRA [5] system. A shortcoming of the above approach is that if the error detected were due to a software fault (e.g. the algorithm was faulty or the compiler had generated faulty code), then the re-execution of the same algorithm will not lead to fault avoidance. A solution to this problem is to use a different algorithm (after the rollback) thus providing a measure of fault-tolerance against software faults. A recovery block is a program structure embodying this form of backward error recovery [2,6].

The structure of a recovery block is shown below:

```
ensure <acceptance test> by
  <primary block>
  else by
  <alternative 1>
  ...
  else by
  <alternative n>
  else error;
```

Fig. 1: A recovery block.

As discussed in [2,6], the primary and the alternative blocks represent different algorithms for producing acceptable results, the primary block representing the preferred algorithm. After the execution of the primary block, the acceptance test (a boolean expression) is evaluated to check that the results produced are acceptable. If so, the statement following the recovery block is executed; otherwise the state of the computation is restored to that at entry to the recovery block and the first alternative is tried, and so on. If the primary and all the alternatives fail, then this is regarded as the failure of the current recovery block; further recovery may be performed by the enclosing recovery block, if any (recovery blocks can be nested). A 'recovery cache' is used for recording the original state of the computation. The recovery cache is organised as a stack and contains recovery data for the recovery blocks entered but not yet exited. The recovery data consists of the addresses and prior values of the global variables updated inside a given recovery block, so that the act of state restoration merely consists of copying the prior values into the variables. Complete details of the recovery block and the cache are discussed in [2,6].
The recovery block approach can be complemented by the inclusion of the more conventional techniques for coping with anticipated abnormal situations [7]. Such techniques are usually referred to as 'exception handling techniques. For example, if a programmer anticipates that a particular algorithm will occasionally generate an arithmetic overflow exception, then he can provide an exception handler for coping with this event. Note however that, if an exception is raised for which no handler is available, then this is taken to mean the failure of the primary or the current alternative and the same recovery actions are taken as in the case of an acceptance test failure.

3. Programming Using Recovery Blocks

A number of interesting issues arise when one considers the use of recovery blocks in a given system. These are, namely: (i) how should the overall program be structured?; (ii) what considerations should go into the design of alternatives?; (iii) how should acceptance tests be designed?; and (iv) how costly is it to provide software redundancy (in terms of space and time)? In this section we address these issues in turn.

3.1 Program Structure

While recovery blocks can be used to provide an arbitrary 'grain of recovery', their usage is likely to be most appropriate where a fairly complex function is to be implemented. The recovery facilities available in a language will strongly influence the overall structure of programs. For example, in the discussion on recovery blocks given in the last section, only assignments were considered as recoverable (since state restoration is equivalent to undoing of assignments). Under this condition, if actions other than assignments were performed from within a primary or alternative (e.g. input-output, opening a file), then the effects resulting from these actions will not be automatically undone when recovery is invoked — the programmer must provide recovery actions himself. If we assume that all non-assignment actions of a process are those concerned with file manipulation and that the complications of programmer provided recovery actions are to be avoided, then the following general method of organising programs suggests itself:
By copying parts of the opened files in the data area of the given program, we ensure that all data modifications reduce to assignment operations - hence can be made recoverable automatically. The merge sort example to be discussed later illustrates this approach.

3.2 Design of Alternatives

The 'distinct software' approach advanced in [3] appears to us the most appropriate way of designing the primary and the alternatives of a recovery block. The idea is to employ design teams to independently develop algorithms from the same specification - the independence should increase the chances of the algorithms not containing the same design faults. Thus the circumstances that lead to the failure of one algorithm might not cause the failure of the other.

There are three possibilities of choosing the appropriate algorithms:
(a) The first possibility is to have the design aim of the primary and the alternatives to produce exactly the same results - the primary representing the most efficient method of producing the results while the alternatives representing perhaps less efficient (but possibly logically simpler) methods of producing the same results (the sort examples of the next section typify this approach).
(b) The second possibility is to design alternatives to provide a degraded service (thus producing different but nevertheless acceptable results). As an example of this, assume that it is required to design a reliable file processing module. Fig. 3 shows a method of achieving this (the program structure of fig. 2 is employed). In this example, 'good processing' is taken to mean that either the file is processed appropriately or not at all. Thus
open file;
copy the relevant parts of the file into the program data area;
ensure `<good processing>` by begin .. process data .. end
else by
print ('file not processed')
else error;
copy program data to the file;

Fig. 3: Reliable file processing

the alternative has been designed to merely inform the user that the processing was not performed. (c) The third possibility is not to design an alternative but to re-execute the failed primary again:

ensure `<acceptance test>` by ..... else by retry else by ...

Here we have assumed a 'retry' facility which states 'execute again the primary'. It should be noted that the above facility resembles the more conventional 'checkpoint and restart' recovery techniques employed in some fault-tolerant systems and no tolerance to design faults is available. Clearly, in a given application, a combination of the three possible approaches can be employed.

3.3 Design of Acceptance Tests

These run time tests ensure that computations produce acceptable results. The criterion for acceptability will depend upon the way a recovery block has been designed. If a primary and its alternatives have been designed using the first possible approach discussed previously (i.e. they produce identical results) such that a relationship 'P' should hold among the program variables (after the execution of the primary or the current alternative) then the acceptance test should be designed to check 'P'. If this is likely to prove expensive, then a weaker condition, 'Q' such that 'Q' is necessary but not sufficient to guarantee 'P' may be chosen. The condition 'Q' may be taken as the 'adequacy criterion' for the recovery block [8].

If, on the other hand, a primary and its alternatives have been designed according to the second possible approach discussed before (i.e. the alternatives provide a degraded service) then the acceptance test can only be as rigorous
as the check involved in testing the adequacy of the alternative providing the most degraded service (usually the last one). Quite often this test reduces to the checking of certain invariant relationships that exist between program variables. For example, 'good processing' in fig. 3 may only be an expression to check that certain relationships among file records have been preserved, whether processing was performed or not. Some times this may prove unacceptable where the invariant relationship is not strong enough to detect any incorrect processing (i.e. processing that maintains the invariant but nevertheless is incorrect). What is needed is an explicit check to test the results of file processing. This may be done neatly as shown in fig. 4, where part of fig. 3 has been programmed.

```
ensure I by
begin .. process data ..; assert Q end
else by
    print ('file not processed')
else error;
```

Fig. 4: File processing with rigorous checking

Let 'I' represent the invariant relationship and 'Q' be the condition that should hold after successful processing so that 'I&Q' should be true after the execution of the primary. It is assumed that a failure of the asserted condition causes a failure of the primary.

3.4 Cost of Providing Software Redundancy

The recovery block approach requires the provisions of result checking and a recovery data collection and maintenance facility (recovery cache) hence it is important to know the following overheads: (i) time overhead for collecting and maintaining recovery data, (ii) time overhead for evaluating acceptance tests, and (iii) space overhead for storing recovery data. While some data regarding these overheads is presented in the next section, some general remarks can be made here.

As discussed in [6], recovery caches can be implemented by hardware, thus recovery data collection and maintenance procedures can be made very fast. A hardware architecture has also been proposed for the evaluation of acceptance tests concurrently with the execution of recovery blocks [9]—thus speeding
up computations. Regarding space overheads, it should be noted that the recovery cache collects recovery data for only those global variables that are modified inside a given block. This is at variance with the more conventional complete checkpointing schemes where the recovery data for all global variables is recorded.

4. Experimental Work

The usefulness of recovery blocks in recovering from residual faults in a system was demonstrated by the earlier experimental work described in [10]. The purpose of the work to be described in this section was to evaluate the performance of a recovery cache. The experimental set up consisted of the programming language Sequential Pascal [11] extended to include recovery blocks [12]. The code produced by the Sequential Pascal compiler is for a simple stack machine and is executed by an interpreter programmed to run on the host hardware (PDP11/45). This interpreter was modified to support recovery blocks and caches. The relative timing measurements presented here may be taken as indicative of the performance of hardware implemented recovery caches. Note that recovery cache implementation employed a very simple approach to recovery data collection and maintenance; in particular, none of the possible optimizations mentioned in [6,10] were implemented. Thus the timing performance figures given here should be regarded as conservative. A few programs were written and their performance was evaluated (most of the algorithms were taken from a text book [13]):

1) To find the median of n items - the primary algorithm was the efficient 'partition' method, the alternative was a simple but slow scanning method. The acceptance test checked that the number of items smaller than or equal to (greater than or equal to) the median were ≥(n-1)/2 (see entry 1 in tables 1 and 2).

2) Internal sort - to sort lines of texts into alphabetical order. The primary algorithm was the efficient 'Quicksort' method while the alternative was the 'Shellsort' method. The acceptance test ensured that the lines were in the ascending order (see entry 2 in tables 1 and 2).
(3) **Merge sort** - to sort lines of texts residing in secondary storage. This program was developed using the organisation of fig. 2. Parts of the file to be sorted were brought into the main store and sorted using the internal sort program developed for (2) above. The sorted parts of the file were stored on temporary files. These files were then merged to form a single sorted output file. Merging consists of bringing in the main store a line each from the heads of the temporary files, sorting them and outputting the 'smallest' line to the output file; a further line from the appropriate file is read in and the process is repeated. In this experimental program, 'heapsort' was used as the primary for merge sorting and Quicksort as the alternative. Acceptance tests employed were similar to those used in (2). Two sets of readings were taken: one with an assumed error in Quicksort (see entry 3 in tables 1 and 2) and the other with an assumed error in heapsort (see entry 3* in tables 1 and 2).

(4) **The stable marriage problem** - given two sets $M$ and $W$ of equal cardinality $n$, the problem is to find $n$ 'couples' $(m,w)$ such that $m$ in $M$ and $w$ in $W$ satisfy a certain constraint. A number of solutions exist for a given pair of sets, hence an 'optimality criterion' was chosen to select one solution out of all possible ones. The program was structured with an algorithm that generates the optimal solution as the primary and an algorithm that generates the first possible solution as the alternative. Because the primary generates the optimal solution, its execution time is substantially larger than that of the alternative (see entry 4 in tables 1 and 2). The acceptance test checked that there was no 'polygamy'.

For all these programs, the number of words actually recorded in the recovery cache for the primaries was calculated and compared with the case of complete checkpointing (where the state of all the global variables would be recorded)*. The time to evaluate acceptance tests were also measured - they turned out to be negligibly small compared with the execution time of the primaries or the alternatives. From the data presented, the following conclusions can be drawn:

* For every word occupied by a variable to be 'cached', there will be two entries in the recovery cache - address and value; for a complete checkpoint, only values need be recorded.
### Table 1: Performance Figures

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$% T_{CM}$</th>
<th>$% T_{RC}$</th>
<th>$N_G$</th>
<th>$N_R$</th>
<th>$% C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.6</td>
<td>.62</td>
<td>.64</td>
<td>1.31</td>
<td>3.33</td>
<td>9.33</td>
<td>30</td>
<td>8</td>
<td>26.66</td>
</tr>
<tr>
<td>2</td>
<td>.36</td>
<td>.4</td>
<td>1.2</td>
<td>1.68</td>
<td>11.11</td>
<td>21.9</td>
<td>58</td>
<td>8</td>
<td>13.78</td>
</tr>
<tr>
<td>3</td>
<td>4.69</td>
<td>4.74</td>
<td>14.33</td>
<td>20.4</td>
<td>1.28</td>
<td>28.25</td>
<td>227</td>
<td>8</td>
<td>3.52</td>
</tr>
<tr>
<td>3*</td>
<td>-</td>
<td>-</td>
<td>4.6</td>
<td>10.47</td>
<td>1.18</td>
<td>25.21</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
<td>2.06</td>
<td>.18</td>
<td>2.26</td>
<td>3.0</td>
<td>0.7</td>
<td>257</td>
<td>40</td>
<td>15.56</td>
</tr>
</tbody>
</table>

### Table 2: Performance Figures (for larger data)

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$% T_{CM}$</th>
<th>$% T_{RC}$</th>
<th>$N_G$</th>
<th>$N_R$</th>
<th>$% C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.8</td>
<td>4.0</td>
<td>4.1</td>
<td>9.26</td>
<td>5.26</td>
<td>30.5</td>
<td>138</td>
<td>54</td>
<td>39.12</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.94</td>
<td>5.8</td>
<td>8.3</td>
<td>7.77</td>
<td>31.08</td>
<td>234</td>
<td>32</td>
<td>13.66</td>
</tr>
<tr>
<td>3</td>
<td>39.03</td>
<td>39.6</td>
<td>118.73</td>
<td>170.0</td>
<td>1.46</td>
<td>30.07</td>
<td>482</td>
<td>20</td>
<td>4.14</td>
</tr>
<tr>
<td>3*</td>
<td>-</td>
<td>-</td>
<td>37.66</td>
<td>87.66</td>
<td>1.05</td>
<td>31.37</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>16.06</td>
<td>16.23</td>
<td>2.56</td>
<td>19.13</td>
<td>1.03</td>
<td>1.05</td>
<td>307</td>
<td>72</td>
<td>22.14</td>
</tr>
</tbody>
</table>

**Note:** Times are given in seconds.

- $T_1$ - execution time without any recovery facilities;
- $T_2$ - execution time for a primary block;
- $T_3$ - execution time for an alternative;
- $T_4$ - execution time with failure in the primary (a failure of the acceptance test is assumed);
- $\% T_{CM}$ - time to collect and maintain recovery data expressed as a $\%$ of $T_1 = \frac{T_2 - T_1}{T_1} \times 100$;
- $\% T_{RC}$ - time to restore system state expressed as a $\%$ of $T_1 = \frac{T_4 - (T_2 + T_3)}{T_1} \times 100$;
- $N_G$ - number of global variable 'words';
- $N_R$ - number of words recorded in the cache;
- $\% C = \frac{N_R}{N_G} \times 100$;
(a) If we assume that in a well designed system, failure of a primary will be a rare event, then it is important to know the time overheads for recovery data collection and maintenance when no errors are detected. From the sample programs we see that $\%T_{CM}$ ranged between 1 to 12% of the execution time of the primaries.

(b) When a primary does fail, it is of interest to know the time taken to restore system state. For the sample programs, $\%T_{RC}$ was up to about 30% of the execution time of the primaries.

(c) A comparison with complete checkpointing shows that, for the sample programs, a substantial saving in space was made by the recovery cache.

While performance data collected will vary from program to program, from the data collected, it does seem that even a simple implementation of recovery cache could prove satisfactory for many applications.

5. Concluding Remarks

A methodology for programming using recovery blocks was presented in section 3. A few medium sized programs employing recovery blocks were written in Sequential Pascal (extended with recovery blocks). The recovery cache needed for programs was implemented by modifying the interpreter. Time and space overheads to support recovery blocks were measured for the above mentioned programs. These figures seem quite encouraging and lead us to believe that recovery caches can provide adequate performance for many applications.

In this paper only the assignments of sequential programs were considered as recoverable. To make other actions of a process recoverable (e.g. file manipulation, control of input-output processes etc) requires the facility of programmed recovery actions. The facility consists of programmer provided recovery procedures to be executed by the recovery support system when recovery is invoked [6,14]. Such recovery procedures will play an important part when programming fault-tolerant features for concurrent processes (say of an operating system) - such programs are termed fault-tolerant concurrent programs. Programming language features for specifying recovery procedures for fault-tolerant concurrent programming are described in [14]. Fig. 5 shows the relationship between fault-tolerant sequential and concurrent programs in a hierarchically structured system. Much interesting work needs to be done in
the following areas: (i) design of appropriate interfaces between sequential and concurrent programs, (ii) investigation of methodologies for structuring fault-tolerant concurrent programs, and (iii) performance measurements (similar to those performed here) for concurrent programs.

<table>
<thead>
<tr>
<th>user programs</th>
<th>&quot;fault-tolerant sequential programs&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>operating system providing virtual resources with</td>
<td>&quot;fault-tolerant concurrent programs&quot;</td>
</tr>
<tr>
<td>recoverable operations</td>
<td></td>
</tr>
<tr>
<td>hardware resources (CPU, I/O equipment, memory units ...)</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 5:** Fault-tolerant concurrent and sequential programs

**Acknowledgements**

This work was supported by the Science Research Council (UK) as a part of the 'highly reliable computing systems' project at the Computing Laboratory of the Newcastle upon Tyne University.

**References**


