Abstract:

The programming language Sequential Pascal has been extended to include recovery blocks. This paper describes the modifications made to the kernel and interpreter of Brinch Hansen's Pascal system to support recovery blocks and the associated recovery caches needed for state restoration.

(To appear in Software-Practice and Experience)

Sequential Pascal with Recovery Blocks

By

S.K. Shrivastava

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 123
March, 1978

© 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.
**bibliographical details**

SHRVASTAVA, Santosh Kumar.

*Sequential Pascal with recovery blocks.* By S.K. Shrivastava.

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

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 123.)

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Suggested classmarks (primary classmark underlined)</td>
<td></td>
</tr>
<tr>
<td>Library of congress:</td>
<td>Dewey (17th)</td>
</tr>
<tr>
<td></td>
<td>U.D.C.</td>
</tr>
<tr>
<td></td>
<td>001.6424 001.6425 681.322.06</td>
</tr>
<tr>
<td>Suggested keywords</td>
<td>PAULT-TOLERANT SOFTWARE</td>
</tr>
<tr>
<td></td>
<td>RECOVERY BLOCKS</td>
</tr>
<tr>
<td></td>
<td>RECOVERY CACHE</td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL PASCAL</td>
</tr>
</tbody>
</table>

**Abstract**

The programming language Sequential Pascal has been extended to include recovery blocks. This paper describes the modifications made to the kernel and interpreter of Brinch Hansen's Pascal system to support recovery blocks and the associated recovery caches needed for state restoration.

**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.
INTRODUCTION

A program structure called recovery block has been proposed in the literature as a means of constructing fault-tolerant software\(^1\,2\) (defined to be software that produces acceptable results despite faults in the hardware and software). This paper describes an implementation of recovery blocks using Sequential Pascal\(^3\) as the host language. The objectives of this paper are twofold: firstly, the implementation details are believed to be sufficiently interesting in their own right and secondly, recovery blocks have attracted wide attention (for example, they are actively being evaluated for aerospace applications\(^4\)) thus, an account of a method of inclusion in Sequential Pascal, a language that is widely used for research in programming methodology, should prove interesting to workers in the field of fault-tolerant programming. The paper also demonstrates that the inclusion of recovery blocks into existing programming systems can be a practical proposition.

Recovery blocks were first implemented by my colleagues\(^5\); one of the aims of their work was to investigate a suitable computer architecture for directly supporting recovery blocks. The resulting system could however support only relatively simple sequential programs. A second experiment was therefore started with the aim of developing a system capable of supporting realistic sequential and concurrent programs incorporating recovery blocks. This paper describes the first phase of this experiment - the development of a system that is capable of supporting realistic sequential programs with recovery blocks. Work is underway to extend this system to support the features necessary for fault-tolerant concurrent programming\(^6\). While recovery blocks are described briefly in the next section, a familiarity with the concepts presented elsewhere\(^1\,2\) would be helpful to the reader.
RECOVERY BLOCKS

The syntax as incorporated in Sequential Pascal is shown below:

```
ENSURE <acceptance test> BY
  <statement> "primary"
ELSE-BY <statement> "first alternative"

ELSE-BY <statement> "n th alternative"
ELSE-ERROR;
```

The acceptance test (a boolean expression) is evaluated after the execution of the primary. If the result is true, the statement following the recovery block is executed. However, if the result if false, 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 all the alternatives fail to produce acceptable results, then this is regarded as a failure of the entire recovery block - any recovery actions must be undertaken by the enclosing recovery block, if any (recovery blocks may be nested). A 'recovery cache' is used for recording the state of the computation and restoring it when the primary or the current alternative fails. 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 the 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. When an acceptance test is passed, some of the recovery data of this recovery block may have to be merged with the recovery data of the enclosing recovery block (if any). Precise details of this merging and other related aspects of recovery cache are discussed elsewhere\textsuperscript{1,5}.

THE PASCAL SYSTEM

The Pascal System, as developed by Børnich Hansen's group, is capable of supporting a number of concurrent processes programmed in Concurrent Pascal\textsuperscript{7}. A process is capable of executing sequential programs written in Sequential Pascal (this language is closely similar to Pascal\textsuperscript{8}, from which it has been derived). A process can make available
some of its procedures to the sequential program it is running - this forms the basis of the interface between user programs (written in Sequential Pascal) and the operating system (written in Concurrent Pascal). Such procedures have been called prefix procedures (as a consequence of prefix procedures, no input-output has been defined for Sequential Pascal; rather, a system designer can program appropriate input-output procedures in Concurrent Pascal as prefix procedures). Both the concurrent program and sequential programs are executed interpretively by a simple stack machine programmed to run on the host hardware (PDP 11/45). Certain details of this interpreter and related programs are of interest within the context of this paper.

The kernel and interpreter

The kernel is the initial piece of software written to run on the base machine and it implements processes, synchronising primitives, queues, basic input and output and other features as required by Concurrent Pascal. It also implements a virtual storage system. The virtual address space of a process is shown in Fig. 1. The space is divided into two regions. The common segment is common to all the processes in the system. The virtual code is the code produced by the Concurrent Pascal compiler and is executed by the interpreter. The interpreter table contains, for all the virtual instructions, pointers to the interpreter areas where the appropriate interpretive procedures are stored. The stack and heap in the private segment are maintained and used by the interpreter (the heap is needed for the implementation of dynamic store allocation features of Sequential Pascal). The private data in the private segment can contain any virtual code produced by the Sequential Pascal compiler, it is also executed by the same interpreter. The kernel maintains a process head for every process in the system. The process head contains such details as the processor time used by the process, priority, an area for saving the contents of the processor registers etc. When the kernel selects a particular process for running, it copies its process head into the current process head area of the common segment and hands over control to the interpreter. The current process head acts as an interface
Figure 1. Address space of a process
between the interpreter and the kernel. Finally, the common data in
the common segment contains the data (monitor variables) needed for
inter-process co-ordination.

Exception handling by the interpreter

The execution of a program is terminated either because it term-
inates properly or because some predefined error condition is detected
by the interpreter (such conditions include range error, stack limit,
heap limit etc.) In either case, the interpreter executes an exception
program; the essential features of this program are as shown. An entry
in the current process head, 'result' is used to record the cause of the
termination (result = 0 means 'proper termination', result = 1 means
'overflow error' etc).

```
exceptions if program = concurrent & result / proper then
  begin
    print('system error');
    stop "failure in the concurrent program (i.e.,
    operating system), so stop further processing"
  end else
  begin
    restore stack;
    return
  end
```

RECOVERY BLOCK IMPLEMENTATION

A number of changes were made to the kernel and interpreter to
support recovery blocks and associated recovery caches. Despite the
fact that the kernel and interpreter have been programmed in the
assembly language of PDP 11/45 (MACRO assembler on DOS operating system),
no particular difficulty was encountered in these modifications.

This is because the Pascal system was found to be an outstandingly
well engineered product.

The Sequential Pascal compiler was modified to accept the recovery
block construct shown previously and to generate the following code,
where only the recovery block virtual instructions are shown explicitly.
The particular control structure was selected after studying the code
generation characteristics of the compiler.
ENTER RECOVERY (NUMBER); "enter recovery block"

goto 12;
11: evaluate acceptance test;
    if true then goto 13;
    ATFAIL;'acceptance test failed, the control goes to the
    exception handler'
12: case next of
    0:{primary}; goto 11;
    n:{n th alternative}; goto 11; "n = NUMBER-1"
end
13: ATPASS; "acceptance test passed"

'NUMBER-1' equals the number of alternatives and 'next' indicates the
statement to be executed.

The heaps in the private segments of processes (see fig. 1) were
replaced by caches. Two reasons are given for this decision: (1) there
does not appear to be any straightforward method of arranging the
recovery of heap variables (since they do not follow the normal block
structured rules), hence it was decided not to incorporate them in this
modified version of Sequential Pascal; (ii) it is necessary to associate
a cache with each process in the system; the most suitable place for its
incorporation is the private segment of each process. Fig. 2 shows the
structure and organisation of the recovery cache of a process; the
starred entries in the current process head show the additions made to
the process head of every process to support the corresponding recovery
cache (in the actual implemented version, a few more entries have been
included in the process heads and the caches, with a view to future use
for recovery for concurrent programs; for simplicity these have been
ignored in this discussion). The details of the various virtual instruc-
tions and related programs will now be described.

Enter recovery block

The algorithm of the virtual instruction ENTER RECOVERY (NUMBER) is
as shown. The data for recovery blocks that have been entered but not
yet exited are separated by barriers as shown in fig. 2 (which shows the
data for two recovery blocks). The cachbr entry points to the barrier
of the current recovery block; the barriers are linked as shown. The
Fig. 2. Recovery Cache Organisation
The interpreter uses four processor registers for pointing to the stack top (S), local variables (B), global variables (G) and the next virtual instruction to be executed (Q). These registers are saved as shown.

```pascal
procedure enter recovery (number:integer);
begin
  if heaptop > (S "stack top" + 14 "bytes") then
    begin
      result := 'heap limit'; "ie, cache limit"
      goto exception
    end
  else
    begin
      Using heaptop, store current
      S,B,G and Q in the cache (=heap);
      store number; next := 0;
      push this value of next on the stack;
      "to be used by the 'case' instruction of the
      recovery block virtual code"
      create a barrier and link it to the
      previous one; cachbr := location of the new
      barrier;
      nest := nest + 1; "counts the nesting of recovery
      blocks"
    end
end
```

**Assignments**

Whenever a sequential program performs an assignment, it is necessary to check the following: (i) is it being performed from within a recovery block? (ii) if so, is the variable global? (iii) is so, does the current recovery data in the cache include the value - address pair(s) for this variable? If the answers to the first two questions are yes and no to the third, the value and the address of the variable are recorded in the cache (before the assignment is performed). This recording is done on a word basis, so if the length of the variable is more than a word, the appropriate number of entries are made. In step (iii), the search of the current cache region each time an assignment is performed is likely to be the major contributing factor to the execution time overheads. However as it is generally accepted that well structured

*When recovery caches are hardware implemented, extra bits can be added to the store words and utilised such that no search overheads are involved.*
programs should contain a minimal number of assignments to globals, the search time should not prove prohibitive for such programs. All the interpreter instructions that performed assignments (e.g. COPYWORD, COPYSET etc.) were modified to perform the above 'cacheing'.

**Acceptance test fail and pass**

The algorithms for the virtual instructions ATFAIL and ATPASS are shown below.

```plaintext
procedure acceptance test fail;
    begin
        result := 'acceptance test fail';
        goto exception
    end

procedure acceptance test pass;
    begin
        nest := nest-1;
        if nest > 0 then merge else discard
    end
```

The procedure 'merge' merges the appropriate recovery data of the recovery block just completed with the recovery data of the enclosing recovery block. When the recovery block just completed is the outermost one (nest=0), 'discard' is called to throw away all the recovery data generated for the program in execution.

**Exception handling**

From the algorithm given previously for exception handling, we see that the execution of a sequential program is terminated as soon as an abnormal condition is detected. This is no longer the case with recovery blocks: if an abnormal condition is detected while executing a recovery block, it is merely regarded as a failure of the primary or the alternative, as the case may be, and the same recovery actions are invoked as in the case of acceptance test failure. When the current program terminates properly, nest=0 will also hold; from the new exception handling algorithm we see that the stack will be restored and a return made to the appropriate point in the executing process. If result /\ proper, then next>0 implies that recovery capability exists. From the algorithm,
we see that $S, B, G$ and $Q$ are restored and procedure 'restore' is called. This procedure restores the prior values of the cached variables. If an alternative exists (next < number) then this alternative will be executed when an exit is made from the exception handler. Otherwise, state restoration is carried out for the enclosing recovery block (if any), and so on. A number of error reporting messages have also been included in the handler. For example, when result / proper and nest = 0 then this implies that no recovery is available, so a message 'recovery exhausted' is printed. Fig. 3 shows a toy program with recovery blocks and the error messages produced.

CONCLUDING REMARKS

In the modifications described here, care has been taken to see that programs that do not use recovery blocks are not affected. Thus, the SOLO operating system$^9$ and all the application programs available on it run on the modified kernel and interpreter. The SOLO system can be used to develop Sequential Pascal programs with recovery blocks.
exception: var recovery:boolean;
    recovery:=false;
    if result / proper then print (result type);
    if nest>0 then
    begin
        while nest>0 & ~recovery do
        begin
            using cachbr, copy back S,B,C and Q;
            restore; next:= next+1; if next<number then
            begin
                push the value of next
                on the stack;
                recovery:=true
            end else
            begin print ('recovery block fail');
            nest:=nest-1;
            cachbr:= location of the previous barrier
            end
        end;
        if ~ recovery then
        begin
            if result / proper & nest = 0 then
            begin print ('recovery exhausted!');
                if program = concurrent then
                begin print ('system stop');
                    stop
                end
            end; restore stack;
            return
        end
    end

The modified exception handler
PROGRAM TOY;
TYPE M = 1..10;
VAR N: ARRAY(J.M.) OF INTEGER;
I, J: INTEGER;
BEGIN
  ENSURE (I = 2) & (J = 2) BY USER LINE 11 RANGE ERROR
  BEGIN
    ENSURE J=10 BY USER LINE 9 A.T. FAIL
    BEGIN
      J:=0;
      N(J):=0
      END ELSE_BY
    END ELSE_ERROR;
    I:=2
  END ELSE_ERROR
  BEGIN
    I:=3; J:=45
  END ELSE_BY
BEGIN
  I:=2; J:=2
END ELSE_ERROR

Fig. 3(a) A toy program
Fig. 3(b) Error messages
Only a few modifications to the kernel were needed — those concerned with the changes in process heads and error reporting facilities. The majority of the modifications were confined to the interpreter. The size of the original interpreter was about 1K words, the new size is 1.8K words.

Since it is possible now to develop realistic programs with recovery blocks, several interesting questions arise regarding the design and performance of such programs (e.g. how should a unit of recovery be chosen, what is the time needed for state restoration, etc). An attempt has been made to answer some of these questions elsewhere. However, two results should be of interest to the readers of this paper. Firstly, timing measurements taken for a few programs without recovery blocks and with recovery blocks (containing from about 5 to 50 global assignments) indicated that the time overhead for collecting and maintaining recovery data for a recovery block ranged from about 2 to 10 percent. Secondly, state restoration time for these programs ranged from 20 to 30 percent of the execution time of the primaries. It is thus seen that, while a hardware implementation of caches is the best method, even the simple method of implementing caches by software as described here can be quite practical for many applications.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the efforts of two of my colleagues: P.C. Treleaven who modified the Sequential Pascal compiler and P.A. Lee who developed the support software needed for transferring data from the DOS system to the Pascal system. His constructive criticisms on all aspects of this work are also gratefully acknowledged. Acknowledgement is also due to R. Kerr whose critical comments led to the improvement of this paper. This work was supported by the Science Research Council as a part of the 'Highly Reliable Computing Systems' project at the Newcastle University.
References


(9) P.B. Hansen, 'The SOLO operating system: a Concurrent Pascal program', Software-Practice and Experience, 6, 141-149 (1976).

(10) S.K. Shrivastava and A.A. Akinkelu, 'Fault-tolerant sequential programming using recovery blocks', Tech. Report, Computing Laboratory, University of Newcastle upon Tyne.