Abstract:

Backward error recovery is an integral part of the recovery block scheme that has been advanced as a method for providing tolerance against faults in software; the recovery cache has been proposed as a mechanism for providing this error recovery capability. This paper describes a recovery cache that is being built for the PDP-11 family of machines. This recovery cache has been designed to be an "add-on" unit which requires no hardware alterations to the host CPU but which intersects the bus between the CPU and the memory modules. Specially designed hardware enables concurrent operation of the recovery cache and the host system, and aims to minimise the overheads imposed on the host. The paper describes the design of the recovery cache and the new instructions which have (conceptually) been added to the host CPU, and provides an overview of the software necessary to control the device.

A Recovery Cache for the PDP-11

By

P.A. Lee, N. Ghani and K. Heron

TECHNICAL REPORT SERIES

Editor: Dr. B. Shaw

© 1979 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NEI 7TRU, England.
Lee, Peter Alan

A recovery cache for the PDP-11. [By] P.A. Lee,
N. Chani, K. Heron.

Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1979.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 134.)

Added entries
GHANI, Noordin
HERON, Keith
UNIVERSITY OF NEWCASTLE UPON TYNE,

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th):
U. D. C.

001.642
681.322.06

621.381958
681.322.02

Suggested keywords
BACKWARD ERROR RECOVERY
RECOVERY BLOCKS
RECOVERY CACHE

Abstract

Backward error recovery is an integral part of the recovery block scheme that has been advanced as a method for providing tolerance against faults in software; the recovery cache has been proposed as a mechanism for providing this error recovery capability. This paper describes a recovery cache that is being built for the PDP-11 family of machines. This recovery cache has been designed to be an "add-on" unit which requires no hardware alterations to the host CPU but which intersects the bus between the CPU and the memory modules. Specially designed hardware enables concurrent operation of the recovery cache and the host system, and aims to minimise the overheads imposed on the host. The paper describes the design of the recovery cache and the new instructions which have (conceptually) been added to the host CPU, and provides an overview of the software necessary to control the device.

About the author
P.A. Lee is a Research Associate in the Computing Laboratory at the University of Newcastle upon Tyne.
N. Chani is a Research Associate in the Computing Laboratory at the University of Newcastle upon Tyne.
K. Heron is a Programming Adviser and Systems Engineer in the Computing Laboratory at the University of Newcastle upon Tyne.
Introduction

While fault tolerance at the hardware level is common in computing systems, fault tolerant software, that is, software that can produce acceptable results despite design faults in that software, is rare. The recovery block scheme has been proposed as a method of introducing redundancy at the software levels in a computing system in order to provide tolerance against such design faults. (It is beyond the scope of this paper to discuss the details of recovery blocks, although a knowledge of the scheme is assumed for this paper. The interested reader is referred to [Hor74, Ran75, And76, Shr78, Lee78].)

Design faults in a program will lead to the generation of erroneous states (errors) in the variables of that program. One of the features of recovery blocks is that if an error is detected within a recovery block then backward error recovery occurs in that the states of the variables of the program are reset to the states that existed just prior to entry of that block. By this means, the errors generated by the failing algorithm are recovered from and not allowed to propagate. In order to support this state restoration, a mechanism termed the recovery cache has been proposed for providing the necessary recovery capability in an efficient (and reliable) manner; in its simplest form the recovery cache can provide, by hardware, recovery for those variables that reside in the main store of the computer. This paper discusses the design of a recovery cache which can be incorporated into existing computer systems, and describes an experimental version which is being built at the Computing Laboratory of the University of Newcastle upon Tyne for the PDP-11 family of machines.

Overview of the Recovery Cache

The main purpose of the recovery cache is to record recovery data so that backward error recovery can be provided for the variables of a program containing recovery blocks.

The basic functioning of the recovery cache is as follows: when a recovery block has been entered and an object is about to be written to (for the first time), the original value of that object is retrieved and stored away (together with its address) by the recovery cache, before the object is actually updated. By this means the recovery cache maintains a minimum of recovery data, and backward error recovery simply involves restoring those changed objects using the values and addresses retained in the recovery cache. (Several methods have been devised for the implementation of the recovery cache algorithms - two examples are presented by Horning et al [Hor74] and by Anderson and Kerr [And76].)

Ideally, the recovery cache would be designed as an integral part of the computer's architecture and incorporated in the CPU and memory hardware. However, it is also desirable to have a recovery cache that could be added to existing computer systems to enable recovery blocks to be used realistically on those systems.

This paper presents the design of an "add-on" recovery cache that can be incorporated into an existing system with no changes to the CPU, memory or other peripheral device, and will not affect the existing software running on that system. Many of the aspects of the design will be independent from the host system to which it is to be attached. However, for our initial experiments the host system will be a PDP-11,
and this will be assumed in the rest of this document. The host-dependent features will be discussed further in later sections.

The way in which the recovery cache is to be added to a PDP-11 system is depicted in figure 1. The recovery cache will intersect the Unibus between the CPU and the memory. In this position the recovery cache can monitor all of the CPU activity on the Unibus, although it would not be able to monitor activity between the memory and any other peripheral device (i.e. those to the right of the memories in figure 1). A second point to note is that the recovery cache would not be able to directly access the registers which are internal to the CPU, namely the program counter, the stack pointer, the program status word and the other six general purpose registers. Direct access to these quantities would require alterations to the CPU itself, which was not considered desirable. Nevertheless, this is the simplest method of adding the recovery cache to a system and it has been designed to be flexible enough that these problems can be circumvented.

It may also be noted that in this position the recovery cache will have to work in terms of real addresses. Thus, application programs which made use of overlays would be difficult to handle. Similarly, concurrent use of the recovery cache by multiple parallel processes would produce extra difficulties. Therefore, for the experimental version of the recovery cache, it is proposed to provide recovery only for a single core-resident program, and the recovery cache will contain base-limit address registers to define the address range for which recovery is provided. Extensions to the recovery cache mechanism to deal with some of the problems mentioned above have been discussed in other papers [Ram75, And78, Shr78]. The incorporation of these extensions with the hardware recovery cache described here will be investigated in a later phase.

**Host Software Organisation**

The software running on the host will be divided into two parts: (i) the application program which contains the recovery blocks; and (ii) a small kernel providing some minimal set of operating system-like functions for the application program. The kernel will contain the routines to interface with the recovery cache and will provide recovery actions for the objects for which the recovery cache cannot provide recovery (e.g. saving and restoring the internal registers). Thus the application programs will not need to be concerned with the provision of recovery nor with some of the finer details of controlling the recovery cache. The kernel will also generate the error log which indicates the progress of the program, using one of the peripheral devices on the host system.

Figure 2(a) depicts a high-level language form of a recovery block as it might appear in an application program. The execution of this recovery block would be as follows: when the recovery block is entered, the primary alternate (alt1) is executed. At the end of the alternate the acceptance test (a boolean expression) is evaluated. If the acceptance test is 'true' then the recovery block is exited. However, if the acceptance test is 'false' then recovery occurs to restore the state of the program and the second alternate (alt2) is entered. This sequence is repeated until either the acceptance test is 'true' or the set of alternates has been exhausted. The way in which this recovery block will be represented on the host system is depicted in figure 2(b), where the uppercase names refer to recovery cache instructions. The semantics
of these instructions are discussed below.

Recovery Cache Instructions

The recovery cache has been designed to be as automatic as possible and to require a minimum of control from the program that is utilising its features. For example, all of the CPU-memory activity on the Unibus will be automatically intercepted by the recovery cache and will not require nor be dependent upon explicit actions by the program.

To make use of the recovery cache some new "instructions" will have to be added to the instruction set of the CPU. As alterations to the CPU are to be avoided, these new instructions can be most easily obtained by making the recovery cache look like a normal Unibus peripheral device which is controlled by writing to its status registers. Similarly, parameters to instructions can be passed through device registers.

In general, operation of the CPU should be held up once an instruction has been sent to the recovery cache, in order that it has time to complete its actions before the CPU continues. This is not easy to achieve in the present design, and a simple solution involving a "done" bit in the status register has been adopted.

The instructions for the recovery cache reflect the control structure required for recovery blocks. The recovery cache cannot easily directly access the program counter of the CPU; hence most of the desired control structure will have to be present in the program containing recovery blocks. The proposed instructions aim to provide as much assistance as possible from the recovery cache.

INIT
This instruction will cause the recovery cache to assume an initialised inactive state. This effect will also be achieved when the "START" key on the host system is pressed.

INITENV(lowaddr, highaddr)
For this paper, "recovery environment" refers to that set of memory words for which backward error recovery is to be provided when a recovery block is entered. This instruction will mark the start of a new recovery environment, and will cause the recovery cache to store the state of the environment it is currently monitoring (e.g. addressing range), and sets the (real) address range which is to be monitored when activated by the ENTERRB instruction.

ENDEV
This instruction indicates the end of a recovery environment and will cause the recovery cache to restore its state and restart monitoring the previous environment.

ENTERRB(no. of alts, addr. of alt1, addr. of alt2, ...)
This instruction indicates the start of a new recovery block. The parameters indicate the number of alternates together with their starting address.

AFAIL
This instruction, indicating the successful termination of a recovery block by the passing of the acceptance test, will cause the recovery cache to process and discard as necessary the recovery
data that it had recorded for the current recovery block.

RECOVER
This instruction will cause the recovery actions provided by the recovery cache to be invoked. It will also cause the address of the next alternate to be made available (see below), or will indicate an error if all of the alternates had been attempted.

RAISEERROR
This instruction will be used to raise an error indication for the currently executing program, and has the effect of initiating the ERRINT interrupt discussed below.

PRIOR(address)
This instruction will cause the prior value of the variable whose address is specified to be made available by the recovery cache.

Recovery Cache Provided Information

The following (read-only) information will be provided by the recovery cache via its registers:

REC.LEVEL
The current depth of nesting of recovery blocks. Zero indicates that no recovery is available.

ALT.NUMBER
The number of the next alternate to be obeyed.

NEXTADDR
The address of the next alternate to be entered.

CACHEDONE
When set this bit will indicate that the recovery cache has completed its actions and is ready to receive further instructions.

Note that these quantities will be maintained automatically by the recovery cache. Other status and monitoring information will also be provided to enable the performance of the recovery cache to be monitored.

Recovery Cache Generated Interrupts

Each recovery cache instruction will cause an interrupt to be immediately generated in the host system. Thus routines in the kernel of the host can be provided and automatically invoked to ensure that the program using recovery blocks is properly synchronised with the actions of the recovery cache and, for instance, is not resumed until the recovery cache has completed its actions. These interrupts also provide the means through which the recovery actions provided in the kernel can be invoked to supplement the recovery provided by the recovery cache. Some examples of such actions are given below.

ENTERRB INTERRUPT
This interrupt will be generated from the ENTERRB instruction, and will allow for kernel-provided recovery data recording to be initiated. For example, the kernel can record recovery data to preserve the states of the internal registers.
ATPASS INTERRUPT
This interrupt, generated from the ATPASS instruction, can be used to invoke any processing of kernel-provided recovery actions. For example, the kernel could discard the recovery data it had recorded for restoring the internal registers.

ERRORINT INTERRUPT
This interrupt will be generated by the recovery cache to indicate that it has detected (or had been informed) that an error condition existed. The interrupt would be used to force the CPU to initiate recovery actions and enter the next alternate, as well as allowing for the initiation of any kernel-provided recovery actions (e.g. the restoration of the internal registers).

The code sequence in the CPU to handle the ERRORINT would be of the form indicated below (where uppercase names refer to the recovery cache instructions and information):-

ERRORINT INTERRUPT:
if REC.LEVEL \leq 0
  then terminate program
else begin
  perform error logging;
  perform any kernel-provided recovery;
  RECOVER;
  wait for CACHEDONE;
  if REC.LEVEL = 0
    then terminate program
  else begin
    record recovery data for any
    kernel-provided recovery actions;
    return from interrupt but use
    address specified in NEXTADDR;
  end;
end;

Recovery Cache Hardware

The experimental version of the recovery cache now being built is based on an LSI-11 microcomputer with a number of special purpose peripherals (see figure 3). This results in simple, low-cost hardware with many of the functions carried out by microprocessor software, and will provide a flexible research vehicle which will allow further studies to be made in optimisation of the recovery cache algorithms and in extensions of the techniques to more sophisticated environments.

While there will of necessity be some overheads associated with the recovery cache (discussed below), it is anticipated that the overheads imposed on the host system will be minimal, particularly as use is made of specially designed hardware to enable concurrent operation of the recovery cache and the host system. Moreover, the design of the recovery cache is such that the transfer of functions from microcomputer software into hardware could be easily achieved in order to reduce further any overheads that the experimental system demonstrates. For example, it is envisaged that the LSI-11 could be simply replaced by a special purpose processor capable of executing the recovery cache algorithms at high speed.
The peripherals shown in figure 3 have the following functions:

1. The CHIU (Cache Host Intercommunication Unit) appears to the host PDP-11 as a peripheral device and provides the interface to the host software as described above. It allows the recovery kernel in the host and the recovery cache to interact explicitly.

2. The CHMAU (Cache Host Memory Access Unit) allows the recovery cache to access the host memory directly through the normal Unibus mechanism. It is used mainly for restoring old values to the memory when recovery is invoked.

3. The BMU (Bus Monitor Unit) performs hardware monitoring of all data transfers between the host processor and its memory. It is able to prevent the host from writing into a memory location until the old value from that location has been saved, as necessary, by the recovery cache. The BMU is transparent to the host, and appears to the recovery cache processor as a device which provides a stream of address-value pairs.

4. The CDU (Cache Decision Unit) is a hardware unit that executes the decision process necessary to establish whether an address-value pair should be saved in the recovery cache, and it transfers the data into the appropriate place in the recovery cache memory, as necessary. Currently, the CDU will implement an algorithm based on that described by Anderson and Kerr [And76].

The only host-dependent features of the recovery cache are the parts of these four peripherals which actually interface with the host bus. Of these, only the BMU interface requires special consideration in its design; the others make use of standard I/O interfacing techniques (interrupts in the case of the CHIU and direct memory access in the case of the CHMAU). The BMU interface must be able to delay certain signals propagating between the host processor and memory, and to hold up the processor during a memory transfer. This is relatively easy to achieve in most small computer systems. It should therefore be relatively simple to modify the recovery cache hardware to enable it to be used with other host systems.

**Recovery Cache Operation**

The BMU contains base-limit address registers which are initialised in response to an INITENV instruction. When enabled by an ENTERBB instruction, the BMU begins monitoring the address and control lines of the host bus. Transfers on this bus involving addresses outside the limits are allowed to proceed without interference. Otherwise only straightforward read transfers are allowed to proceed and write transfers are intercepted. A write transfer may or may not require the original value of an object to be saved by the recovery cache (for instance, an address-value pair may only need recording the first time an object is updated; subsequent updates of that object can then be ignored by the recovery cache.) While this decision process is relatively simple, the time taken for its execution may be comparable with the cycle time of the host memory. Therefore, in the proposed design this decision process is executed concurrently with the read of the original value from the host memory. If an address-value pair is to be saved in the recovery cache then the CDU initiates the necessary direct memory transfers to store this recovery data in the memory of the recovery cache; during this time the write transfer from the host system
can be allowed to proceed.

On the Unibus there are in fact two types of write transfer: read-modify-write transfers and simple write transfers. The read-modify-write transfer automatically generates the information required by the recovery cache - when the memory location is read, the value can be captured by the BMU; the following write part of the transfer can then be ignored by the BMU. Conceptually, simple write transfers have to be delayed by the BMU until the original value has been read from memory. In practice, the write transfer is converted to a read-modify-write transfer by the BMU to achieve the desired effect.

**Performance Considerations**

The recovery cache will slow down the host system in two main ways. Firstly, there is the time needed to interpret the recovery cache instructions such as ENTERRE, as the application program cannot be allowed to proceed until such instructions have been completed. Such instructions are expected to form only a small percentage of the executed instructions of the application program. Hence the delay caused by their execution should not be significant, although it should be noted that the time to interpret the ATPASS and RECOVER instructions will in general depend on the behaviour of the application program.

The second source of delays are those introduced through the interference of the BMU with memory transfer cycles of the host system. These delays are more significant as every memory cycle is delayed to some extent.

The minimum level of degradation is defined by the delays introduced by the address and control line checking. In the experimental (non-optimised) system this delay is expected to be of the order of 200 nanoseconds per transfer, not much more than that introduced by, say, a bus extender unit.

The BMU performs its saving operations on both read-modify-write and pure write cycles that are within the address range defined by the INITENW instruction; normally only write cycles need to be extended, as explained above. Monitoring the activity on a PDP-11 Unibus has indicated that in general approximately 8% of transfers are write transfers and 2% are read-modify-write transfers, figures which agree with those published elsewhere [Str76]. These figures suggest that the extra overhead on the host system would be of the order of 8%, caused by the need to precede each write by a read. If memory with destructive readout is used, the fact that a write cycle is converted into a read-modify-write cycle means that the additional delay is less than that incurred by forcing a separate read cycle. In this case the extra overhead would be expected to be of the order of 4%. Of course, these overheads will depend critically on the behaviour of the application program and these figures only provide a general indication of the overheads which it is hoped will occur in practice.

The above figures assume that the time required to execute the caching decision and to record the address-value pair is comparable with the time between write transfers on the host system. The monitoring figures discussed above indicate that, on average, the time available is equivalent to the time for 9 memory reads by the host CPU. The CDU has been designed to satisfy this criterion. Clearly, if successive write
transfers occur on the Unibus then some extra delay is inevitable. In practice, most write transfers are followed by at least one read transfer (the instruction fetch), and it is therefore anticipated that this extra delay will only occur on rare occasions. Further 'smoothing out' of successive write transfers will be achieved by incorporating a first-in first-out buffer in the EMU.

Summary

This paper has presented the design of an add-on recovery cache and has discussed an experimental version that is currently being built for a PDP-11 host system. While the overheads imposed on the host by the recovery cache will depend on the behaviour of the application program, the paper has shown how the delays can be minimised. Measurements of the experimental system will indicate where optimisations should be applied, and this will be reported in a later paper.

Acknowledgements

We would like to express our gratitude to the fellow members of the Reliability Project, which is sponsored by the U.K. Science Research Council, at the University of Newcastle upon Tyne.

References


Figure 1. PDP-11 System with Recovery Cache

```
.
.
.
.
ensure acceptance test
by primary alternate (alt1)
  alt1: primary alternate ;goto at;
elseby second alternate (alt2)
  alt2: second alternate ;goto at;
  .
  .
elseby nth alternate (altn)
  altn: nth alternate ;goto at;
else error:
  at: acceptance test;
  if true then ATPASS
  else RAISEERROR;
  .
  .
ENDENV
```

(a) High-level language format  (b) Low-level language format

Figure 2. Recovery Blocks in a Program
Figure 3. Hardware Organisation of the Add-on Recovery Cache