EFFICIENT LOCAL CHECKPOINTING FOR SOFTWARE FAULT TOLERANCE

Krishna Kant
EECS Department, Northwestern University
Evanston, IL 60201

INTRODUCTION

The Recovery Block (RB) concept proposed by Randell [1] for implementing software fault tolerance involves the establishment of local checkpoints (LCP) whenever a process enters a RB. In order for the scheme to be practical, it is important that the overhead of this checkpointing be small both in time and storage. Therefore, we must look into alternatives to simply caching the complete process state at the point of entry to a RB. Since the checkpointing mechanism need be concerned with only those variables that are modified within a RB, an obvious approach is to keep only the previous (i.e. unmodified) values of these variables. In this note, we examine some implementation difficulties with this approach and then propose a somewhat unusual scheme which saves only the modified values at the end of a RB execution.

RECORDING PRIOR VALUES

The idea here is to record the value of a variable just before it is to be modified for the first time within the RB. A rollback then simply consists in restoring previous values of these variables. Unfortunately, the scheme suffers from two problems.

First, a reasonable implementation would require additional hardware which constantly monitors the system bus for store type of instructions and delays the instruction execution in case the original contents of this address need to be cached [2]. The hardware required for monitoring, synchronization and caching may be substantial. Secondly, the mapping between the variable names and their bus addresses is rarely straightforward. The mapping depends both on the memory management techniques used by the system and the storage allocation mechanism used by the language in which the program was written. Take for example the case of a pascal program running on a system using paged memory management. In this case, two levels of "backward" translation will be required in order to identify the variable from the bus address. First, a search should be made in order to determine the page number corresponding to the block number part of the physical address. Secondly, the resulting virtual address must be translated into the appropriate offset into the run time stack used by the pascal implementation. The hardware could, of course, be designed to monitor virtual addresses rather than physical addresses; however, this may not be very desirable.

A second problem with this scheme arises when we consider consistent state restoration in concurrent programs. Since the rollback of one process in the program may require the rollback of another, the final LCP to which a process needs to be rolled back may be far removed from the last established LCP. In such a case, if every LCP had a complete record of the process state, we could first determine the final LCP for rollback and then restore the process to this LCP. However, with each LCP holding prior values of only those variables that are actually modified in the associated RB, this cannot be done and a step-by-step rollback, starting with the last established LCP is necessary. Figure 1 shows the difficulty pictorially. The average number of rollback steps required will grow as the process interaction level and hence the average length of rollback increases. The problem can be solved by establishing a full LCP after every N partial LCP´s. This would place an upper bound of N/2 on the average number of rollback steps required for error recovery. The parameter N can be chosen so as to minimize the combined overhead of checkpointing and recovery.

RECORDING MODIFIED VALUES

The basic idea in this scheme is to record only those variables that have been modified. We assume that we have a tagged architecture, such as the one proposed for capability implementation [3]. Every word of the memory contains a "modified" bit which is automatically set to 1 by hardware whenever its contents are modified. We also assume an instruction for initializing the modify bit for a sequence of memory locations to 0. The initialization is done every time the execution of a new RB alternate is initiated. Upon exit from a RB, a scan is made over all memory locations corresponding to the nonlocal variables of the RB, and new values recorded for all those which have been modified. It may be noted that this mechanism records exactly those variables which are stored by the first scheme; the only difference is that we store the modified values rather than the prior values. We also assume that a full LCP is established after every N partial LCP´s. The parameter N can again be chosen so as to minimize the total cost of checkpointing and recovery.

The recovery scheme can now be stated very simply. We first determine the final LCP to which a given process must be rolled back and then locate the nearest full LCP established prior to it. We restore the process to this full LCP and then successively use the later partial LCP´s for state modification until we reach the correct rollback point. (See Figure 2).

The main advantage of this scheme over the first one is that the required hardware support is very simple; it is essentially the "tagged architecture" advocated for capability based addressing [3]. The step-by-step rollback will, of course, be required in any scheme which establishes partial LCP´s.

It is clear that the checkpointing cost is same for both mechanisms. The number of rollback steps required to complete error recovery is always N/2 for scheme 2; however, for scheme 1, it depends upon the average rollback span (ARS), i.e. the number of previous RB executions that need to be discarded. Note that ARS1 for sequential programs. As ARS increases, so does the average number of rollback steps in scheme 1 but is is always bounded by N/2. Thus the first scheme is cheaper for sequential programs and concurrent programs involving very infrequent process interaction. In other situations, both schemes incur approximately the same amount of recovery overhead.

REFERENCES

1. B. Randell, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering, Vol SE-1, pp220-232, June 1975.

2. P. A. Lee, N. Ghani and K. Heron, "A Recovery Cache for the PDP-11", Technical Report No. 134, Univ. of Newcastle upon Tyne, March 1979.

3. R. S. Fabry, "Capability based Addressing", Communications of the ACM, Vol 17, No 7, pp403-412, July 1974.

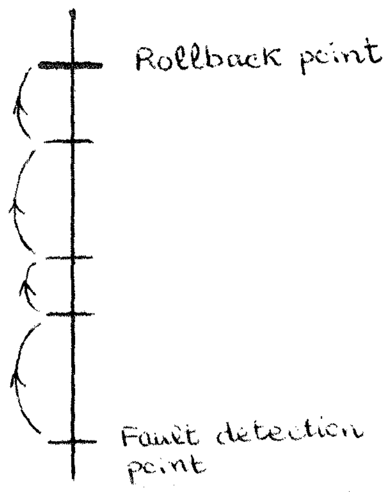

Figure 1

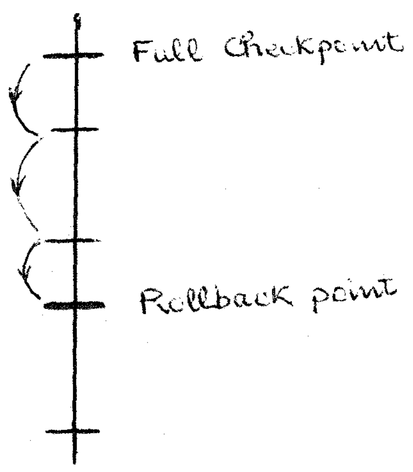

Figure 2