Rollback Recovery in Real-Time Systems
With Dynamic Constraints

Shambhu J. Upadhyaya

Department of Electrical and Computer Engineering
State University of New York at Buffalo
Buffalo NY 14260

Abstract: Rollback recovery is a backward error recovery technique to recover from transient faults in computing systems. Real-time systems employing fault tolerance and reconfiguration generally have time dependent (dynamic) constraints. In this paper, we present a new rollback point insertion strategy which evaluates the rollback conditions on-line. The proposed technique minimizes both time and space overhead associated with rollback thereby making it applicable to real-time systems with dynamic constraints. The algorithm presented here attains a near-optimum solution in terms of the time spent in saving the states of the system. Details of the simulation conducted to validate the technique are also given.

1. Introduction

It is well known that most hardware faults in computing systems are soft [2]. That is, the majority of faults in the hardware are of transitory nature. Rollback recovery [1,4,5,6] is a backward error recovery scheme to recover from transitory faults in computer systems. In real-time systems, it is crucial that recovery be possible within a specified time at any point in a program. Therefore, the rollback scheme used should be truly tailor-made so that the insertion of rollback points (called checkpointing) is automated. Chandy and Ramamoorthy [1] gave a model for inserting rollback points in process control type programs. However, the checkpointing technique proposed in [1] is not suitable for systems where the parameters of the program are time dependent during the mission. This is because, the parameters are treated as static (time invariant) in order to insert rollback points in an optimal way. Any changes in the parameters would therefore require a re-evaluation of conditions implying considerable precomputation. Thus, the technique of [1] is less plausible to react to the changes in the parameters of the program in real-time. In this paper, we propose a new checkpointing scheme to handle dynamic variations in the parameters of the program.

In Section 2, we discuss the basic algorithm of [1], and develop a new and simpler model for rollback in dynamic environment. A near-optimum, time-efficient rollback algorithm is presented in Section 3. This algorithm is first developed for simple linear graphs and later generalized to more complex acyclic program graphs. A computer simulation is conducted to study the performance of the proposed algorithm. The proposed algorithm is compared with that of [1] and the simulation results are presented in Section 4. Details of dynamic constraints and a discussion appear in Section 5.

2. Preliminaries and Problem Statement

We briefly present Chandy and Ramamoorthy's model [1], referred to in the sequel as CR model. The corresponding rollback point insertion algorithm is referred to as CR algorithm. In CR model, a program is represented by a directed graph where node i represents task i and a directed edge (i,j) exists if task i is followed by task j with nonzero probability. By suitable preanalysis, an arbitrary program is transformed into a graph with no cycles leading to an acyclic graph.

2.1 Worst Case Design Problem

For each node i of the program graph, a real number \( t_i \) is specified, \( t_i \) is the expected time of completion of task i. With each edge (i,j) of the program graph, a quantity \( S(i,j) \) - save time \([1]\) is associated, which is the time taken to save the state of the system (possibly in a secondary store) after task i is completed and when it is known that task j follows task i. Load time \( L(i,j) \) [1] associated with edge (i,j) is the time taken to load the state of the system at edge (i,j) to the main memory. The recovery time \( R_i \) at node i is the time required to load the saved state at the most recent rollback point and to rerun the program from this point to node i. If the first task executed after placing the most recent rollback point is denoted by task 1 and the next task by task 2 and so on, then \( R_i = L(1,2) + \sum_{k=1}^{i-1} r_k \), where \( r_k \) is the actual execution time of task \( s, r_k \leq t_k \).

Let task i be just completed and task j is to be processed next. At each edge (i,j), an interrogation is made to determine whether a rollback point is required in that edge. If a rollback point is required, then a copy of the state is made. If a transient fault is detected before the creation of the next rollback point, recovery is initiated.

Research funded in part by National Science Foundation Grant CCR-89-09243.
by restarting the program from the most recent rollback point. An algorithm based on the worst case design has the following inputs, constraints and objective.

**Inputs:** $t_i$ associated with each task $i$, $L(i,j)$, $S(i,j)$ associated with each edge $(i,j)$ and a specified maximum recovery time $M$.

**Constraints:** Rollback points are inserted such that at every point in the program, the maximum possible recovery time does not exceed the specified maximum recovery time $M$.

**Objective:** Minimize the maximum time that may be spent in saving the states of the system.

In CR algorithm, a constant $B(i,j)$ — a function of $t_i$, $L(i,j)$ and $S(i,j)$ is computed for each edge in the program graph. A rollback point is inserted in edge $(i,j)$ if the recovery time $R$ is greater than $B(i,j)$. Flow chart of Figure 1 describes the CR algorithm [1].

In order to evaluate $B(i,j)$ for edge $(i,j)$, two functions $f_i(R)$ and $g_j(R)$ are defined for each node $i$ and each edge $(i,j)$ respectively. An optimal decision variable $x_{ij}(R)$ is also defined for each edge $(i,j)$.

$f_i(R) = \text{minimum time spent in saving state of the system after task } i \text{ is completed and before the completion of program for all possible } R$;

$g_j(R) = \text{minimum time spent in saving state of the system after task } i \text{ is completed and before the completion of the program if task } i \text{ is followed by task } j$;

$x_{ij}(R) = 1 \text{ if a rollback point is to be inserted on edge } (i,j)\text{, otherwise.}$

### 2.2 Problem Statement

The computational complexity of $B(i,j)$'s in CR algorithm is $O(n + e)$ where $n$ and $e$ are the number of vertices and edges respectively in the program graph. The evaluation of $B(i,j)$'s requires complex data structures and considerable computation. Fixing $B(i,j)$'s for all runs of the program in advance restricts the application of the algorithm to only those systems in which $M$ and $t_i$ are constant. But the parameters of the program are not time invariant in all real-time systems. In order to apply CR algorithm to such situations, the constants $B(i,j)$'s have to be re-evaluated using the new values. The re-evaluation of $B(i,j)$'s for a program graph due to changes in the parameters will be time intensive and may not be practical in real-time systems. A new problem of rollback point insertion in a dynamic environment is defined as follows:

**Obtain an algorithm for rollback point insertion with the constraint that recovery should be possible within a specified time $M$, where $M$, $t_i$, $S(i,j)$ and $L(i,j)$ are all time dependent quantities during the mission.**

### 2.3 A Model for Rollback

In order to present a simple rollback point insertion strategy for real-time systems with time dependent parameters we set one the parameters $L(i,j)$ to a constant for all $(i,j)$. Though this simplification might introduce some errors, simulation of Section 4 shows that such errors are negligible.

**Definition 1:** For each pair of adjacent edges $(i,j)$ and $(j,k)$ joined at vertex $j$, a function $g(i,j,k)$ is defined as

$$g(i,j,k) = S(j,k) - S(i,j), \quad i < j < k \quad (1)$$

**Definition 2:** For an edge $(i,j)$, a quantity $S(i,j)_{max}$ is defined as the maximum of all the save times associated with the edges in the paths from node $j$ to $n$ where $n$ is the exit node. $S(i,j)_{max}$ is 0 if $j = n$.

**Definition 3:** For an edge $(i,j)$ of the program graph, a function $E(i,j)$ is defined as

$$E(i,j) = M - R_i - t_j \quad (2)$$

where $M$ is the maximum specified recovery time. The relevance of $E(i,j)$ in rollback point insertion can be intuitively explained as follows: Let task $i$ be just completed and task $j$ is to be processed next. Let $(i,j)$ and $(j,k)$ be adjacent. If $E(i,j)$ is found negative after the execution of task $i$, a rollback point must be inserted in edge $(i,j)$ before the execution of task $j$ so that the recovery time constraint is not violated. On the other hand, if $E(i,j)$ is
positive, then based on the expected time of completion of task \( k \), it will be possible to predict the requirement of a rollback point in edge \((j, k)\) even before the execution of task \( j \). This look-ahead feature constitutes the main part of the rollback point insertion algorithm (discussed in next section) leading to the minimization of system save times.

We now define another function \( D(i, k) \) as:

\[
D(i, k) = E(i, j) - t_k
\]

Clearly, if \( D(i, k) \geq 0 \) then, there is no need to insert rollback points at edge \((i, j)\) or \((j, k)\).

**Definition 4:** Let \((i, j)\) and \((j, k)\) be two adjacent edges in a given program graph. At edge \((i, j)\), the cumulative gain \( c.gain(j) \) is defined recursively as

\[
c.gain(j) = \begin{cases} 
  c.gain(i) - S(j, k)_{\max} + g(i, j, k), & \text{if } A \geq 0 \\
  c.gain(i) - S(j, i)_{\max} + g(i, j, k), & \text{if } A < 0 
\end{cases}
\]

where \( A = D(i, k) + R_i \).

When a program graph is entered (at the entry node), the cumulative gain is 0. In the above definition, \( A \geq 0 \) implies that no rollback point will be required in edge \((j, k)\) if a rollback point is inserted in edge \((i, j)\) (see the constraint in Section 2.1). On the other hand, if \( A < 0 \), insertion of rollback point in edge \((i, j)\) may not necessarily mean that a rollback point will not be required in edge \((j, k)\). Accordingly, \( S(j, i)_{\max} \) should be used in computing \( c.gain(j) \) instead of \( S(j, k)_{\max} \).

In dynamic checkpointing, optimal insertion of rollback points is accomplished by judiciously placing the rollback points on the edges of the program graph based on certain criteria. The quantity \( c.gain(j) \) is an indicator of the gain in save time by shifting a rollback point from an edge to its adjacent edge. The criterion for the insertion of rollback points and the associated algorithm is presented next.

3. A New Algorithm for Rollback

We first present a necessary condition for the insertion of rollback points. For the sake of simplicity, assume that the entry node of the program graph is denoted by integer 1. Also, whenever a rollback point is inserted at an edge, say \((i, j)\), node \( j \) is renamed as \( 1 \). We call this process a resetting activity. By reset, we mean that insertion of rollback point is complete and execution proceeds to the next adjacent edge by renaming task \( j \) as task 1.

**Step 1:**

for edge \((i, j)\) do

compute \( E(i, j) \);

if \( E(i, j) < 0 \) then insert rollback point and reset;

end

Step 1 inserts rollback points following the constraint of the problem. This step contains the necessary condition for rollback point insertion, irrespective of any optimization strategy. However, in a special case where \( S(i, j) \)'s are equal for all edges, then Step 1 is also a sufficient condition to minimize the total time spent in saving the states of the system by minimizing the total number of rollback points.

Step 1 alone is not sufficient to optimize the rollback point insertion in a scenario where \( S(i, j) \)'s are variable. The algorithm which incorporates the effect of the variable save times is presented below first for a class of simple graphs.

3.1 Case of Linear Graphs

A linear graph corresponds to a simple sequence of tasks. Programs written in a structured language often consist of simple sequence of modules (tasks). Although, it is common to have branches in structured programs, by properly subsuming such constructs within a module, most such programs can be transformed into linear graphs.

The Algorithm (ONLINE1)

**Inputs, Constraint and Objective:** Same as before (in Section 2).

**Initialization:** Determine \( S(i, j)_{\max} \) for all edges of the program graph. Set cumulative gain \( c.gain(1) = 0 \).

**Rollback point insertion strategy:** Let task \( i \) be just completed and edges \((i, j)\) and \((j, k)\) be adjacent. The following strategy is used for rollback point insertion in edge \((i, j)\).

execute Step 1;

Step 2:

if \( E(i, j) \geq 0 \) then compute \( D(i, k) \);

if \( D(i, k) \geq 0 \) then do not insert rollback point;

\( i \leftarrow j, j \leftarrow k \); \{ go to adjacent edge without reset \}

end;

Step 3:

if \( D(i, k) < 0 \) then compute \( g(i, j, k) \) and \( c.gain(j) \);

if \( g(i, j, k) > 0 \) and \( c.gain(j) \geq 0 \) then

insert rollback point;

reset;

else do not insert rollback point;

reset \( c.gain(j) \) to its previous value;

\( i \leftarrow j, j \leftarrow k \); \{ go to adjacent edge \}

end;

end;

The above steps are executed at each edge in the execution path between the entry node and the exit node of the program graph during the normal run of the program. The quantity \( c.gain(j) \) is evaluated in edge \((i, j)\) only if the condition given in Step 3 is satisfied.
The following theorems are used to justify algorithm ONLINE1.

**Theorem 1:** Let $S$ be a linear graph. Let the total number of rollback points inserted in $S$ using Step 1 be $N$. Let it be known that a rollback point is inserted in edge $(j,k)$. Let $(i,j)$ and $(j,k)$ be adjacent. If Step 1 is forced to insert a rollback point at edge $(i,j)$, then $N$ will increase by no more than one.

A pessimistic interpretation of Theorem 1 is that each misplaced rollback point can cause an extra rollback point to be inserted. However, with $S(i,j)$ a variable, ONLINE1 algorithm performs the misplacement of rollback points only if there is a non-negative gain in save time in doing so. This aspect of the algorithm ONLINE1 is summarized in the following theorem.

**Theorem 2:** Let the algorithm ONLINE1 be used for rollback point insertion in a program. Then the gain in save times $c$-gain$(n)$, where $n$ is the exit node is non-negative at the end of execution of the program.

### 3.2 An example

We discuss an example to illustrate the application of algorithm ONLINE1. A linear graph consisting of 10 nodes is given in Figure 2. The parameters $t_i$, $S(i,j)$ and $S(i,j)_{\text{max}}$ are listed on the graph. Without loss of generality, we set $L(i,j)'s$ to 0. Assume that $M$ is specified as 25 units and each task takes 5 units of time for a given run.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$S(i,j)$</th>
<th>$S(i,j)_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

![Figure 2: An example](image)

Start with edge $(1,2)$. Step 1 of algorithm ONLINE1 determines $E(1,2) = M - R_i - t_2 = 25 - 5 - 15 = 5$. Since $E(1,2) \geq 0$ we compute $D(1,3) = E(1,2) - t_3 = 5 - 10 = -5$. Next, we go to Step 3 and compute $g(1,2,3)$ and $c$-gain$(2)$. Since $g(1,2,3) = S(2,3) - S(1,2) = 5 - 1 = 4 > 0$, but $c$-gain$(2) = c$-gain$(1) - S(2,3)_{\text{max}} + g(1,2,3) = 0 - 15 + 4 = -11 < 0$, go to next edge without inserting a rollback point at $(1,2)$. We reset $c$-gain$(2)$ to its previous value which is 0. At edge $(2,3)$, the following computations are done.

$E(2,3) = 25 - 10 - 10 = 5$, $D(2,4) = 5 - 10 = -5$

$g(2,3,4) = 15 - 5 = 10$, $c$-gain$(3) = 0 - 8 + 10 = 2$

Since both $g(2,3,4)$ and $c$-gain$(3)$ are positive, we insert a rollback point at edge $(2,3)$. Continuing like this, it can be shown that rollback points will be inserted at edges $(2,3),(4,5)$ and $(7,8)$. The cumulative gain at the end of the execution is 3. The time spent in saving the states of the system is 13 units.

Application of CR algorithm [1] to the same program graph would force rollback points at edges $(4,5)$ and $(8,9)$ resulting in 12 units of time spent in saving the states of the system.

### 3.3 Generalization

Algorithm ONLINE1 can be extended to more general acyclic graphs such as a tree structure. The extended algorithm is called ONLINE2 and is described below.

Let task $i$ be just completed and task $j$ is to be processed next. Edge $(i,j)$ can now have several adjacent edges. Let there be $l$ adjacent edges denoted by $(j,k_1)$, $(j,k_2)$, ..., $(j,k_l)$. If $l = 1$, $(j,k_1)$ is represented by the unsubscripted notation $(j,k)$ as in linear graphs.

The general structure of ONLINE2 is exactly the same as that of ONLINE1. However, the following changes are required. In Step 2 and Step 3, $D(i,k)$ is replaced by the maximum of $D(i,k_1)$, $D(i,k_2)$, ..., $D(i,k_l)$. Further, in Step 3, $g(i,j,k)$ and $c$-gain$(j)$ are computed for all $k = k_1$, $k_2$, ..., $k_l$. Finally, in line 3 of Step 3, the condition $g(i,j,k) > 0$ and $c$-gain$(j) \geq 0$ is checked for all $k = k_1,k_2,...,k_l$. These modifications in the algorithm are necessitated by the uncertainty at edge $(i,j)$ with respect to the execution path. In view of the above changes, $c$-gain$(j)$ needs to be redefined as follows. The quantity $c$-gain$(j)$ gives the predicted cumulative gain at edge $(i,j)$. The actual cumulative gain at $(i,j)$ can be obtained after the execution of task $j$, that is, after learning which of the $l$ adjacent edges of $(i,j)$ will be executed. The actual value of $c$-gain$(j)$ can then be used to update the current value of $c$-gain$(j)$ for the subsequent edge. Note that $c$-gain$(j)$ is evaluated cumulatively only for the execution path (path taken by the execution of a program) of the program graph. The algorithm ONLINE2 requires more sophisticated data
structures compared to ONLINE1, and consequently, the amount of on-line computation will be higher than that of ONLINE1.

4. Simulation Studies

We have done a simulation to obtain the performance of the proposed ONLINE algorithm (here onwards, we refer to ONLINE1 and ONLINE2 collectively as ONLINE). Along with it we implemented the CR algorithm and made a comparison of ONLINE and CR algorithms from both speed and optimality considerations. Since the size of tasks can be arbitrary in a given program, parameters such as \( t_i \) and \( S(i,j) \) can essentially be treated as random numbers. For our analysis, we generated a number of program graphs using random numbers on a VAX-8600 system using Pascal. The number of nodes in the program graphs are varied between 5 and 650.

The CR and ONLINE algorithms were run on identical program graphs that were generated using the above approach. Two metrics \textit{state save overhead} and \textit{time overhead} were defined for the purpose of comparison. The \textit{state save overhead} is the total time (minimized) spent in saving the states of the system. This is simply the sum of \( S(i,j) \)'s associated with the edges where rollback points are inserted. The \textit{time overhead} is the time of execution of the rollback point insertion algorithm. The \textit{time overhead} is divided into two parts: (1) the \textit{precompute time overhead}: time required to compute \( B(i,j)'s \) in CR algorithm \( (S(i,j)_{\text{max}} \) in ONLINE algorithms), and (2) the \textit{runtime overhead}: time spent during the normal run of the program, to determine the conditions for the rollback point insertion.

4.1 Linear Graphs

Figures 3 and 4 are the plots of the \textit{state save overhead} and the \textit{precompute time overheads} respectively versus the number of nodes in a program graph. From Figure 3 it is clear that the ONLINE1 algorithm gives very close results compared to the CR algorithm. The gains obtained by the CR algorithm are therefore marginal (the maximum of percentage difference in save time from Figure 3 is less than 3%).

![Figure 3: State save overhead comparison](image1)

![Figure 4: Precompute time comparison](image2)

4.2 General Graphs

Figures 5 and 6 give the comparison of CR and ONLINE2 algorithms. It is evident from these figures that ONLINE2 gives better performance than CR algorithm although, it is not as effective as ONLINE1. This is mainly because, the computation of \( S(i,j)_{\text{max}} 's \) in ONLINE2 is more involved than the computation of \( S(i,j)_{\text{max}} 's \) for a linear graph.

4.3 Remarks

(1) ONLINE1 deviates from CR algorithm in that CR algorithm optimizes the save time in a global sense by con-
sidering all the edges before execution. ONLINE1 attempts to optimize the save time considering only a few adjacent edges in a one step look-ahead manner. ONLINE1 will not do any optimization if maximum of $S(i,j)$'s occurs on the edge prior to the exit node of a program graph.

(2) Rollback point insertion is rather insensitive to $L(i,j)$'s. This is justified by the negligible degradation in the optimality of the results of ONLINE algorithms.

(3) Evaluation of $B(i,j)$'s in CR algorithm involves the computation of $g_i(R)$ and $f_i(R)$ for all possible values of $R (O(n + e))$ whereas the evaluation of $S(i,j)_{max}$ requires the examination of all the edges of the program graph once ($O(e)$). In fact, for linear graphs, this precomputation time is negligibly small.

(4) The generation of a program graph is not inexpensive. The programs have to be preanalyzed and reasonable estimates of several parameters of the program have to be worked out. It is essential that the benefits of employing rollback for practical programs should outweigh the expensive task of preanalysis. Thus, the programs that are considered for dynamic checkpointing should have such features as large processing time, repeatability for different data sets and crucial error recovery. This also means that tasks should be of reasonable size.

(5) Program graphs can often represent the high level description of a given program. If the branches in the program are subsumed inside the nodes, the resulting program graph is a linear graph. The proposed technique is most suitable in such applications.

5. Discussion

The results of simulation indicate that the performance degradation (increase in state-save overhead) due to the use of the ONLINE algorithms instead of CR algorithm is negligible. This conclusion holds for any set of values of $t_i$, $S(i,j)$'s and $M$. Therefore, the ONLINE algorithms can be used for programs where the parameters are time dependent.

Application of the ONLINE algorithms to the scenario where $M$ may change entails no extra time overhead. All that we need to do is to use alternative value(s) of $M$ while determining rollback points during that stage of operation in which value(s) of $M$ is variable. CR algorithm cannot handle real-time changes because, it requires large precomputation time to compute $B(i,j)$'s.

Like $M$, parameters $t_i$ and $S(i,j)$ can also vary in reconfigurable systems. In such situations, if ONLINE algorithms are used, only $S(i,j)_{max}$'s will have to be re-evaluated. This computation can be done with negligible overhead, especially if the program graph is linear and the algorithm ONLINE1 is used. Alternatively, if CR algorithm is used, values of $B(i,j)$'s need to be stored for different parameters of the system. But, this will require large amount of memory space.

The simulation study has established that the degradation in performance by using the proposed algorithms is insignificant and the precomputation time is very small for programs that can be represented by general acyclic graphs. On the other hand, for structured programs that can be represented by a simple sequence of tasks, the computational overhead is almost zero. This aspect will make the algorithm applicable to practical systems.

We have compared our technique with an existing rollback point insertion technique which is due to Chandy and Ramamoorthy [1]. The simulation revealed that although their algorithm inserts rollback points in an optimal manner the amount of precomputation makes it less likely applicable to dynamic environment unlike the ONLINE algorithms.

An extension of our work to concurrent systems [3] is currently under study.

Acknowledgments

The author likes to thank Dr. Kewal K. Saluja for the useful discussions during this research.

References


