Recovery Point Selection on a Reverse Binary Tree Task Model

SHYH-KWEI CHEN, STUDENT MEMBER, IEEE, W. T. TSAI, MEMBER, IEEE, AND M. BHAVANI THURAISINGHAM

Abstract—In this paper we analyze the complexity of placing recovery points where the computation is modeled as a reverse binary tree task model. The objective is to minimize the expected computation time of a program in the presence of faults. The method can be extended to an arbitrary reverse tree model. For uniprocessor systems, we propose an optimal placement algorithm. For multiprocessor systems, we describe a procedure to compute its performance. Since no closed form solution is available, we propose an alternative measurement which has a closed form formula. Based upon this formula, we devise algorithms to solve the recovery point placement problem. The estimated formula can be extended to include communication delays where the algorithm devised still applies.

Index Terms—Algorithms, placement algorithm, recovery block.

I. INTRODUCTION

The recovery proposed in [20] has been used for backward error recovery. It is a sequential program structure that consists of acceptance tests, recovery points, and alternative algorithms for a given process. A process saves its state at a recovery point and then enters a recovery region. At the end of a recovery block, the acceptance test is executed to check the correctness of the computation. When a fault is detected, i.e., the computation results fail the acceptance test, the process rolls back to an old state saved at the previous recovery point and executes one of the other alternatives. This procedure is repeated until the computation results pass the acceptance test or all alternative algorithms are exhausted. If all alternatives are exhausted, the control is transferred to a higher level handler.

In the absence of recovery points, the computation has to be restarted from the beginning whenever a fault is detected. It is clear that with respect to many objectives such as minimum completion time, minimum recovery overhead, and maximum throughput, the positioning of the recovery points involves certain tradeoffs. A survey on some of these tradeoffs is presented in [5].

Recovery block techniques can also be extended to concurrent programs. For example, [20], [12], and [14] have proposed synchronous schemes and [13], [25], [2], [26], [9], and [16] have proposed asynchronous schemes for concurrent programs.

The performance analysis on recovery schemes can be classified into two groups: one deals with sequential recovery schemes [3], [27], [4], [5], [7], [18], [23], [24], [6], [17], [22], and the other with concurrent recovery schemes [21], [11], [15]. For example, in [3] the authors used a directed graph (where the vertices denote the tasks and an edge \((i, j)\) exists if and only if task \(i\) may be followed by \(j\)) as their computation model, and the objective was to select optimum recovery points that minimize the maximum and expected time spent in saving states. In [24] the authors modeled the execution of the program as a linear sequence of tasks and considered the optimum recovery point selection problem with respect to minimizing the expected total execution time of the program subject to faults.

In [21] and [15], the authors proposed queueing models that analyze both asynchronous and synchronous concurrent recovery schemes assuming instantaneous message transmission. In [11], a synchronous recovery scheme has been simulated where message delays are taken into account. However, the placement of recovery points in concurrent recovery schemes have not been explored.

The optimal selection of recovery points in an arbitrary concurrent recovery model is rather complex. In this paper, we investigate a special case of this general problem, i.e., a system whose tasks are modeled as a reverse binary tree. Many parallel algorithms have reverse (binary) tree schedules. In fact, the parallel reverse binary tree scheduling algorithm is an active research topic in parallel processing [1]. The proposed placement algorithms are useful if we need to determine the placement of recovery points in these parallel schedules that minimize the expected execution time.

We study two placement strategies. The first is a category of uniprocessor systems, while the other is one of multiprocessor systems with sufficient number of processors. In the case of uniprocessor systems, we present a closed form formula to compute the expected performance of each assignment, and then select the optimal one from all of the possible assignments. A seemingly-
good bottom-up dynamic programming approach is proposed to solve the problem, but ends up with no significant improvement over the exhaustive evaluation. For multiprocessor systems with as many processors as needed, we describe a procedure to compute the expected performance for any given recovery point assignment instead, since no closed form formulas are available. We also propose an estimated formula which does have a closed form similar to that of uniprocessor systems. Based upon the closed form formula, we can evaluate the estimated performance and then design algorithms to place recovery points. The model can be extended to n-ary reverse tree tasks and to the environment where the communication delays are significant. The proposed algorithms can easily be modified to fit these extensions.

Section II describes the model of computation without communication delay and the associated assumptions. Section III first investigates optimal placement of recovery points in uniprocessor systems, and then analyzes the complexity of a given dynamic programming formulation. Section IV proposes a procedure to compute the expected performance of the reverse binary tree on multiprocessor systems, and presents an alternative measurement to estimate the performance. Section V applies the proposed measurement to solve some related problems. Section VI considers the communication delays in the model based on the estimated formulas. We conclude the paper in Section VII with a summary of the results.

II. MODEL OF COMPUTATION

The model of computation is a reverse binary tree in the form shown in Fig. 1. Each node represents a process or a task. The leaf nodes are the starting points of the computation and can be executed concurrently. A child node can begin execution only after all its parents have completed their executions. The computation halts when the root node has completed its execution. A recovery point is placed before the execution of each leaf node. It is further assumed that if a recovery point is to be associated with a node, then it is placed prior to the execution of the task. We always have a recovery point after the execution of the root node task. Inter-task communication delay is ignored since it is negligible in a multiprocessor environment. Section VI will consider communication delays in the system.

Associated with each node $i$, we let $t_i$ be the time required to complete task $i$ in the absence of faults, let $s_i$ be the time required to establish a recovery point before $i$ and let $r_i$ be the time required to roll back the computation to the recovery point before $i$. Assume that there are $N$ nodes in the tree out of which $M$ are leaf nodes, i.e., $M = N + 1$. The optimization problem can be now stated as follows: given $t_i, s_i,$ and $r_i$ for each node $i (1 \leq i \leq N)$ and a suitable fault model, select the subset of the $N - M$ recovery point locations (associated with the nonleaf nodes) such that the resulting expected total completion time for the computation is minimized, both in the uniprocessor and $M$-processor environments.

For an $M$-processor system, each leaf node begins execution at exactly the same time and the first job for each process is to place a recovery point before any execution. Failures occur independently and can be detected at the end of the task by an acceptance test (AT). We assume that the time for executing the acceptance test has been included in the execution time of the task. When we place a recovery point (RP) at node $i$, we save the states of the computation just before the execution of task $i$. We perform the acceptance test after task $i$ completes execution. If the acceptance test detects a fault, then it can roll back and restore the state that is previously saved. Without loss of generality, we discuss only the full reverse binary tree model, i.e., each nonleaf node has exactly two parents. We will give examples to extend the result to the general case.

For a uniprocessor system, we assume there is a job scheduler which can arrange the precedent relationships of these tasks. When node $i$ fails its acceptance test, certain tasks should be rolled back. These tasks are scheduled by the scheduler. We also assume the scheduling time is negligible.

As to the failure model, we assume that each node $i$ has probability $p_i$ to complete its job without failures. To simplify our discussion, we further assume that processors are memoryless about previous attempts, so that task retry would face the same failure probability $p_i$.

III. RECOVERY POINT SETTING FOR UNIPROCESSOR SYSTEMS

For any given possible recovery point assignment, we compute its expected performance in time $O(N)$. Then we exhaustively try all possible recovery point assignments to get an optimal solution. A bottom-up dynamic programming formulation is given, and its complexity is analyzed.

A. Performance Evaluation

For the uniprocessor environment, no overlapping of the execution time is possible, so the computation time is
actually the summation of individual times spent by all tasks.

Define $C_i$ to be the expected computation time that is solely used to complete the subtree rooted at task $i$. Define $E_i$ to be the expected time for the completion of task $i$ after task $i$ is invoked, and $K_i$ to be the expected time from the time when task $i$ fails its acceptance test until it resumes computation. (Note that $C_i$, $E_i$, and $K_i$ only accumulate the computation times spent on executing those tasks in the subtree rooted at task $i$, thus exclude the computation times to perform those tasks not in the subtree.)

Computation times can be added because overlapping execution times is not possible in a uniprocessor environment. We have the following theorem.

**Theorem 3.1:** If there is a recovery point at task $i$, then

$$K_i = r_i$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i$$

$$C_i = C_{rl} + C_{rr} + E_i + s_i.$$  

Otherwise,

$$K_i = K_{rl} + E_{rl} + K_{rr} + E_{rr}$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i$$

$$C_i = C_{rl} + C_{rr} + E_i,$$

where $rl$ and $rr$ are the roots of the left and right subtree of node $i$, respectively.

The proof is in Appendix A.

From the above result it can been seen that for any given assignment of recovery points, its performance can be computed in $O(N)$ time where $N$ is the number of tasks. The exhaustive search, with $O(N2^{N/2})$ complexity, can find the optimal recovery point assignment.

The formula can also apply to $m$-ary reverse trees as follows.

If there is a recovery point at task $i$, then

$$K_i = r_i$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i$$

$$C_i = C_{rl} + C_{rr} + \cdots + C_m + E_i + s_i.$$  

Otherwise

$$K_i = K_{rl} + E_{rl} + K_{rr} + E_{rr} + \cdots + K_{m} + E_{m}$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i$$

$$C_i = C_{rl} + C_{rr} + \cdots + C_m + E_i,$$

where $ri$ is the root node of the $i$th subtree, counting from left to right.

**B. Dynamic Programming**

We propose an intuitive bottom-up dynamic programming approach to formalize this problem. However, we only describe the complexity of this approach.

Let $A$ be a full binary tree with number of levels $h$ and the number of nodes $N = 2^h - 1$.

Let $R$ be a connected subtree that have the same root node as the original tree $A$, and only leaf nodes have a recovery point before them ($R$ is not empty).

$q$ The number of subtrees remaining when we remove $R$ from $A$.

$A_i$ The $i$th subtree that is generated by removing $R$ from $A$, $1 \leq i \leq q$.

$k_i$ The number of nodes in the $i$th subtree, $1 \leq i \leq q$.

$l$ The number of leaf nodes of set $R$.

$S_i$ The recovery point setup time for the leaves of $R$.

Thus $\sum_{i=1}^{q} k_i$ is the number of nodes of $A$—the number of nodes in $R$.

**Example:** Consider the reverse tree as shown in Fig. 2, the dotted subtree is one possible $R$, and in this example, $q = 4$, $l = 4$, $k_1 = k_2 = 1$, $k_3 = k_4 = 3$.

Let $T^*_i$ be the optimal expected time of computation of system which is modeled by a reverse binary tree with a set $A$ of nodes and a maximum of $k$ recovery points.

The intuitive formula of dynamic programming is as follows:

$$T^*_i = \min_{\text{total set } R} \left\{ T_R^0 + \sum_{i=1}^{l} s_i + \sum_{i=1}^{q} T^*_{k_i}\right\}.$$  

The complexity of this technique depends on the number of possible $R$’s.

Let $G(h)$ be the total number of connected subtrees that can be obtained from $A$ of $h$ levels and which have the same root as that of $A$, in other words, $G(h)$ is the number of all possible $R$’s.

**Example:**

$$G(1) = 1$$

$$G(2) = 4$$  (as shown in Fig. 3)

Consider any possible $R$ which is rooted at node $x$. The left subtree of $x$ may either be null or one of the trees in $G(h - 1)$, and it is the same for the right subtree. Hence we have the following recurrence relation:

$$G(h) = \begin{cases} (G(h - 1) + 1)^2, & h > 1 \\ 1, & h = 1. \end{cases}$$
Fig. 3. The trees in $G(2)$. By using the mathematical induction, we have

$$O(N^{2h/2}) = O(2^{h2^{h-1}}) < O(6^{h-2}) < O(G(h))$$

$$< O(6^{h-2}), \quad h > 2.$$  

Thus, $O(G(h)) > O(N^{2N/2}).$

There is a one-to-one correspondence between the connected subtreess of the $(h-1)$ level full binary tree $B$, which contains the root of $B$, and all possible $R$'s of the $h$ level reverse tree $A$.

The complexity of using dynamic programming to solve this problem is at least $G(h-1) + 1 = O(N^{2h/4}).$ The reason is that we have to consider all possible $R$’s, which results in $G(h-1)$, and the case of all nonleaf nodes of $A$ without recovery points, which results in "+1".

Example: Fig. 4 demonstrates some of the subtreess of the 3 level full binary tree which corresponds to different sets of $A$ associated with a 4-level reverse tree. Empty tree corresponds to the tree that only the leaf nodes of $A$ can have recovery points, other nonleaf nodes has no recovery point. It is easy to see that there is a one-to-one mapping between the subtreess of the 3-level full binary tree and the sets of $R$ of 4-level reverse tree $A$.

IV. RECOVERY POINT SETTING FOR M-PROCESSOR SYSTEMS

In this section, we propose a procedure to compute the expected performance of the reverse binary tree tasks for an $M$-processor system, and present an alternative measure to estimate the performance.

A. Performance Evaluation

Given a reverse binary tree and a possible recovery point assignment, we can derive the generating function [19], whose coefficients correspond to the probabilities of all possible events. The expected performance can thus be obtained by enumerating the cases specified in the generating function, as long as it converges. We use three models to illustrate the procedure. The first two show the relationship between the probabilities of specific events and the coefficients of the generating functions, and briefly describe how to perform the procedure, while the other demonstrates how to define generating functions from the problem structures.

Model I: Consider $N = 3, M = 2$, task 3 has no recovery point as shown in Fig. 5.

Let $P_k$ denote the probability that task 3 fails its acceptance test $k-1$ times prior to its first success, while task 1 and task 2 have been invoked a total of $i$ and $j$ times, respectively. $P_k = 0$, if 1) $k < 1$, 2) $i < k$, or 3) $j < k$.

Theorem 4.1:

1. $\Sigma \Sigma \Sigma P_k = 1$.
2. $P_k = (1 - p)^i - p_1 (1 - p_2)^j - p_2 p_3.$

Proof: Only the $k$th term of $A_1(x_1) A_2(x_2)$, $A_3(x_3)$ contributes to $x_k$, thus the coefficient of $x_k$ in $P(x_1, x_2, x_3)$ depends on $A_1(x_1) A_2(x_2) A_3(x_3)$ which equals $A_1(x_1) A_2(x_2) (1 - p_3)^{-i} p_3 x_3^3$ which equals $A_1(x_1) A_2(x_2) (1 - p_3)^{-i} p_3 x_3^3$.

By using the mathematical induction, we have

$$O(N^{2h/2}) = O(2^{h2^{h-1}}) < O(6^{h-2}) < O(G(h))$$

$$< O(6^{h-2}), \quad h > 2.$$  

Thus, $O(G(h)) > O(N^{2N/2}).$
contribute the term \( x_i^k \) (or \( x_j^k \) whose coefficient equals \((1 - p_1)^{i-1} p_1 \) (or \((1 - p_2)^{j-1} p_2 \), where \( \sum_{a=1}^k i_a = i \) and \( j_a \geq 1 \) (or \( \sum_{a=1}^k j_a = j \) and \( j_a \geq 1 \)). There are \( \binom{k}{i_1} \binom{k}{j_1} \) possible sets of partitions of \( i \) and \( j \), hence the coefficient of \( x_i^k x_j^k \) in \( p(x_1, x_2, x_3) \) equals

\[
\left( i - 1 \right) \left( j - 1 \right) \prod_{a=1}^k \left( (1 - p_1)^{i_a-1} p_1 \right) \prod_{a=1}^k \left( (1 - p_2)^{j_a-1} p_2 \right) = \left( i - 1 \right) \left( j - 1 \right) p_1^{i-1} (1 - p_1)^{i-1} p_2 \left( 1 - p_2 \right)^{j-1} p_3 \right)
\]

Therefore,

\[
P(x_1, x_2, x_3) = A_1(x_1) A_2(x_2) A_3(x_1, x_2, x_3)
\]

\[
= \sum_{i,j,k \geq 1} \left( i - 1 \right) \left( j - 1 \right) \prod_{a=1}^k \left( (1 - p_1)^{i_a-1} p_1 \right) \prod_{a=1}^k \left( (1 - p_2)^{j_a-1} p_2 \right)
\]

\[
= \sum_{i,j,k \geq 1} \left( i - 1 \right) \left( j - 1 \right) p_1^{i-1} (1 - p_1)^{i-1} p_2 \left( 1 - p_2 \right)^{j-1} p_3 \right)
\]

\[
\cdot p_2^{j-1} (1 - p_2)^{j-1} (1 - p_3)^{k-1} p_3 \prod_{a=1}^k \left( x_i x_j x_k \right).
\]

Procedure: Now we briefly describe the procedure to compute the expected performance as follows.

Given a term \( x_i^k x_j^k \), we first partition \( i \) and \( j \) into positive \( k \) parts, respectively, say \( i_1, i_2, \ldots, i_k \) and \( j_1, j_2, \ldots, j_k \). Since \( i_a \) (or \( j_a \)) relates to the event that between the \((\alpha - 1)\)th and \( \alpha \)th try of task 3, task 1 (or task 2) fails its AT \( i_a - 1 \) (or \( j_a - 1 \)) times and passes it at the \( i_a \)th (or \( j_a \)th) try.

So we can get the time for executing each set of partitions, \( i_1, \ldots, i_k; j_1, \ldots, j_k \) by summing up the following times, for \( 1 \leq \alpha \leq k \):

\[
\max \left\{ s_1 + t_1 + (i_1 - 1) (t_1 + r_1), s_2 + t_2 + (j_1 - 1) (t_2 + r_2) \right\} \quad \text{for} \quad \alpha = 1;
\]

\[
t_3 + \max \left\{ i_a(r_1 + t_1), j_a(r_2 + t_2) \right\} \quad \text{for} \quad \alpha > 1.
\]

Calculate the sum of the products of the execution time for all possible sets of partitions of \( i \) and \( j \), and the corresponding probability \( p_1^{i-1} (1 - p_1)^{i-1} p_2^{j-1} (1 - p_2)^{j-1} p_3 \cdot p_2^{j-1} (1 - p_2)^{j-1} (1 - p_3)^{k-1} p_3 \).

Vary \( i, j, \) and \( k \), sum up the above results, then we can get the expected performance.

Model II: Consider the same model, but task 3 has a recovery point as shown in Fig. 6.

Definition: This time we define the generating functions as follows:

\[
A_1(x_1) = p_1 \sum_{i=1}^{\infty} (1 - p_1)^{i-1} x_1^i
\]

\[
A_2(x_2) = p_2 \sum_{i=1}^{\infty} (1 - p_2)^{i-1} x_2^i
\]

\[
A_3(x_3) = p_3 \sum_{i=1}^{\infty} (1 - p_3)^{i-1} x_3^i
\]

\[
A_4(x_4) = p_4 \sum_{i=1}^{\infty} (1 - p_4)^{i-1} x_4^i
\]

Fig. 6. Task 3 has a recovery point, where \( N = 3, M = 2 \).

Fig. 7. Task 5 has a recovery point, where \( N = 7, M = 4 \).
A3(\(x_1, x_2, x_3\)) = p_3 \sum_{i=1}^{\infty} (1 - p_3)^{-1} x_i^3 \\
A_6(\(x_3, x_4, x_6\)) = p_6 \sum_{i=1}^{\infty} A_3(\(x_3\)) A_4(\(x_4\)) (1 - p_6)^{-1} x_i^6 \\
A_7(\(x_1, x_2, x_3, x_4, x_5, x_6, x_7\)) \\
= p_7 \sum_{i=1}^{\infty} [A_3(\(x_1, x_2, x_3\)) A_3(\(x_3\)) A_4(\(x_4\)) A_6(\(x_3, x_4, x_6\)) \cdot (1 - p_7)]^{-1} x_i^7 \\
P(\(x_1, x_2, x_3, x_4, x_5, x_6, x_7\)) \\
= A_1(\(x_1\)) A_2(\(x_2\)) A_3(\(x_3\)) A_4(\(x_4\)) A_5(\(x_5, x_6\)) \times A_6(\(x_3, x_4, x_6\)) A_7(\(x_1, x_2, x_3, x_4, x_5, x_6, x_7\)).

Overall, we can define generating functions to all the tasks according to the following criteria:

We define \(A_\alpha\) (in terms of itself and all its children) = \(p_\alpha \sum_{i=1}^{\infty} (1 - p_\alpha)^{-1} x_i^\alpha\), if task \(\alpha\) has a recovery point setting before its execution; = \(p_\alpha \sum_{i=1}^{\infty} [\Pi A_\beta (1 - p_\alpha)]^{-1} x_i^\alpha\), otherwise. Where \(\Pi A_\beta\) denotes the product of all all the generating functions of all the nodes belonging to the same subtree \(R\), and \(R\) is a subtree rooted at node \(\alpha\), with recovery points only at all leaf nodes. (Similar to that in Section III-B.)

The generating function of the entire model equals the product of all generating functions of all the nodes.

B. An Estimated Measurement of the Expected Performance

Since the procedure to compute expected performance takes too much effort, we propose an alternative measurement to speed up the estimation of the expected performance. Similar formulations appear in [10], [8]. As in the case of uniprocessor systems, we define \(E^*_i\) to be the estimated time of the completion of task \(i\), \(K_i^*\) to be the estimated computation time from the start until task \(i\) is completed, and \(K_i^*\) to be the estimated time from the time when a fault occurs until task \(i\) resumes computation, and if there is a recovery point at task \(i\), then

\[K_i^* = r_i\]
\[E_i^* = t_i/p_i + (1/p_i - 1) K_i^*\]
\[C_i = \max \{C_i^0, C_i^*\} + E_i^* + s_i\]

otherwise

\[K_i^* = \max \{K_i^0 + E_i^0, K_i^r + E_i^r\}\]
\[E_i^* = t_i/p_i + (1/p_i - 1) K_i^*\]
\[C_i = \max \{C_i^0, C_i^*\} + E_i^*\]

where \(r_i^L\) and \(r_i^R\) denote the root of the left and right subtree of node \(i\), respectively.

As in Section III, these formulas can be easily extended to an \(m\)-ary reverse tree.

V. RELATED SUBPROBLEMS

A. Problem 1

Given the reverse tree structured tasks as shown in Fig. 8. All tasks at level 1 have \(t = s = r = p = 1\). Let \(r_i, s_i, r_i, p_i, 0 \leq i \leq n\) be the parameters associated with tasks at level 2 and level 3, i.e., we have total of \(3n + 1\) tasks of which \(2n\) identical tasks are at level 1, \(n\) variable tasks at level 2, and task 0 at level 3. The problem is to find the estimated optimal recovery point assignment.

B. Problem 1 on a 2n-Processor System

Analysis:

Case 1: Task 0 has a recovery point.
Since each node at level 2 is independent of the other \(n - 1\) nodes according to the proposed estimated formulas, whether a node should have a recovery point or not can be decided individually.

Case 2: Task 0 has no recovery point.
We need to check the following subcases for task \(i, 1 \leq i \leq n\). The computation time for the execution of the assignment of recovery points for each of the subcases will be given.

Let \(C^0_j(s)\) denote the estimated performance(s) of the cases that exactly \(j\) tasks have recovery points. There is one \(C^0_0\), and one \(C^0_n\), but for \(1 \leq j \leq n - 1\), there are \(\binom{n}{j}\) different \(C^0_j\)s.

(0) No recovery points for task \(i, 1 \leq i \leq n\):

\[E_i^* = t_i/p_i + (1/p_i - 1) 2,\]
\[C^0_s = 2 + \max \{E_i^*\} + t_i/p_i + (1/p_i - 1)\]
\[\cdot (2 + \max \{E_i^*\})\]

(1) Only one task, say \(j\), has a recovery point:

\[E_j^* = t_j/p_j + (1/p_j - 1) r_j,\]
\[E_i^* = t_i/p_i + (1/p_i - 1) 2,\]
\[C^0 = 2 + \max \{E_i^*, s_j + E_j^*\} + t_i/p_i + (1/p_i - 1)\]
\[\cdot (\max \{E_i^*, s_j + E_j^*\}).\]

for \(1 \leq i \leq n, i \neq j\). There are at most \(\binom{n}{j}\) different \(C^0_j\)s.
(2) Only two tasks, say $k, j$ have recovery points:
\[
E_i^k = t_i/p_i + (1/p_i - 1) r_i,
\]
\[
E_i^j = t_i/p_i + (1/p_i - 1) r_i,
\]
\[
E_i^j = t_i/p_i + (1/p_i - 1) 2,
\]
\[
C_i^0 = 2 + \max\{E_i^k, s_i + E_i^k, s_i + E_i^j\} + t_0/p_0
\]
\[
+ (1/p_0 - 1) \left( \max\{2 + E_i^j, r_i + E_i^j\} \right).
\]
for $1 \leq i \leq n$, $i \neq k$, $i \neq l$. There are at most \( \binom{n}{2} \) different \( C_i^0 \)'s. Similarly for case \( \beta \), \( \beta \) tasks have recovery points, where \( 3 \leq \beta \leq n - 1 \).

(2) All tasks have recovery points:
\[
E_i = t_i/p_i + (1/p_i - 1) r_i
\]
\[
C_i^\beta = 2 + \max\{s_i + E_i^k, s_i + E_i^j\} + t_0/p_0
\]
\[
+ (1/p_0 - 1) \left( \max\{r_i + E_i^j\} \right).
\]

for $1 \leq i \leq n$.

Let
\[
W_i = t_i/p_i + (1/p_i - 1) 2
\]
\[
X_i = s_i + t_i/p_i + (1/p_i - 1) r_i
\]
\[
Y_i = (1/p_0 - 1) (2 + t_i/p_i + (1/p_i - 1) 2) + t_0/p_0
\]
\[
= (1/p_0 - 1) (t_i/p_i + 2) + t_0/p_0
\]
\[
Z_i = (1/p_0 - 1) (t_i/p_i + r_i) + t_0/p_0
\]
where
\[
W_i \quad \text{The estimated completion time for task } i \text{ without recovery point.}
\]
\[
X_i \quad \text{The estimated completion time for task } i \text{ with recovery point.}
\]
\[
Y_i \quad \text{The estimated recovery time for task } 0, \text{ where task } i \text{ has no recovery point.}
\]
\[
Z_i \quad \text{The estimated recovery time for task } 0, \text{ where task } i \text{ has a recovery point.}
\]

Then,
\[
C_i^0 = 2 + \max\{W_i, X_i\}
\]
\[
C_i^\beta = 2 + \max\{W_i, X_i\} + \max\{Y_i, Z_i\}, \quad i \neq j.
\]
\[
C_i^\beta = 2 + \max\{W_i, X_i, X_i\} + \max\{Y_i, Z_i, Z_i\},
\]
\[
i \neq k, i \neq l,
\]
Similarly for $C_i^\beta$, where $3 \leq \beta \leq n - 1$.
\[
C_i^\beta = 2 + \max\{X_i\} + \max\{Z_i\},
\]
for $1 \leq i, j, k, l \leq n$.

Algorithm 1 and Data Structures: We now state an algorithm that can extract the minimum $C_0$ from these $2^n$ $C_0$'s using only polynomial time $O(n^2)$. The data structures are:

1) \( \text{assign}[i] = \text{"yes"} \) if task \( i \) has a recovery point
\[ = \text{"no"} \) if task \( i \) has no recovery point
\[ = \text{""} \) if task \( i \) has not yet been considered.

2) \( \text{cn} \): a counter that should be no less than \( n \) before the algorithm can enter the inner loop to guarantee that each index has appeared at least once.

3) \( \text{supno} \): a super exponentially growth number, in this case it is 2, at the end of this section a formula that can compute this number for different levels will be provided.

4) \( A \): a one-dimensional array of $2n$ elements. \( B \) has the same structure as \( A \).

5) \( T \): a 2 by \( n \) array. \( T[W, i] = 1 \) if the case that task \( i \) has no recovery point has been considered, \( T[W, i] = 0 \), otherwise, \( T[X, i] = 1 \) if the case that task \( i \) has a recovery point has been considered, \( T[X, i] = 0 \), otherwise.

6) \( \text{minassign}[1..n] \): current minimum recovery point assignment so far.

7) \( \text{mintime} \): current minimum time so far.

After the algorithm is executed, \( \text{minassign}[1..n] \) contains the solution.

Algorithm 1

/* Initialization */
0. \( \text{cn} \leftarrow 0; \text{T} \leftarrow 0; \text{supno} \leftarrow 2; \)
\( \text{minassign} \leftarrow \text{""}; \)
\( \text{mintime} \leftarrow \infty; \)
\( \text{compute } W_i, X_i, Y_i, Z_i; \)
1. sort \( W_i, X_i, Y_i, Z_i; \)
2. sort \( Y_i, Z_i \) nondecreasingly into array \( A; \)
3. keep index \( u \) for array \( A, v \) for array \( B; \)
4. do \( u \leftarrow 1 \) to \( n \text{supno} \)
5. Case: \( A[u] \) comes from \( W_i; \)
6. if \( T[W, i] = 1 \) then \( \text{cn} \leftarrow \text{cn} + 1 \)
7. \( A[u] \) comes from \( X_i; \)
8. if \( T[X, i] = 1 \) then \( \text{cn} \leftarrow \text{cn} + 1 \)
9. \( \text{endcase} \)
10. \( \text{assign} \leftarrow \text{""} /\* \text{clear assign before entering the loop */} \)
11. if \( \text{cn} \geq n \) then
12. do \( v \leftarrow 1 \) to \( n \text{supno} \)
13. if \( B[v] \) comes from \( Y_i \) and \( T[W, i] = 1 \) and \( \text{assign}[i] = \text{""} \)
\( \text{then assign}[i] \leftarrow \text{"no"} \)
14. else if \( B[v] \) comes from \( Z_i \) and \( T[X, i] = 1 \) and \( \text{assign}[i] = \text{""} \)
\( \text{then assign}[i] \leftarrow \text{"yes"} \)
17. \( \text{end} \)
18. compute the executing time using array assign
19. if \( \text{mintime} > \text{ctime} \)
20. then \( \text{minassign} \leftarrow \text{assign} \)
Appendix C.

are obtained by enumerating all possible recovery point assignments. For instance, the term, max \{W_l, Y_l, Z_l\} twice. When it is entered for the first time, table contains

\[ W, = \{W_1, W_2, W_3, W_4\} \]

\[ X, = \{X_1, X_2, X_3, X_4\} \]

\[ Y, = \{Y_1, Y_2, Y_3, Y_4\} \]

\[ Z, = \{Z_1, Z_2, Z_3, Z_4\} \]

Similarly case for \( C_n \), where \( 3 \leq \beta \leq n - 1 \).

An \( O(n) \) algorithm can be used to solve this problem:

Algorithm 2

1. do \( i \leftarrow 1 \) to \( n \)
2. if \( W_i + Y_i > X_i + Z_i \)
3. then minassign[\( i \)] = 'yes'
4. else minassign[\( i \)] = 'no'
5. end
6. end of Algorithm 2

After this algorithm is executed, minassign[1..n] contains the optimal solution.

Because either \( W_i + Y_i \) or \( X_i + Z_i \) will be included in the performance, and each task is independent of any other tasks at the same level, by considering each task separately, we can get the optimal solution.

In fact, if \( X_i < W_i \), then \( 0 < s_i < (1/p_i - 1)(2 - r_i) \), then \( r_i < 2 \), so \( Z_i < Y_i \). This is useful when solving a uniprocessor system, since either \( W_i + Y_i \) or \( X_i + Z_i \) will be added into the solution. If \( W_i + Y_i > X_i + Z_i \), then task \( i \) must have a recovery point to ensure that the performance is optimal.

For example we use the same set of data as in the case of 8-processor system. Since

\[ 1 + 3 < 4 + 1 \]
\[ 5 + 6 > 2 + 5 \]
\[ 3 + 7 = 6 + 4 \]
\[ 7 + 8 > 8 + 2 \]

the estimated optimal recovery point assignment can be (no, yes, no, yes) or (no, yes, yes, yes) and the performance

\[ = (1 + 2 + 3 + 8) + (3 + 5 + 7 + 2) + 2*4 = 39. \]

D. Problem 2

Given the 4-level reverse tree structured tasks with \( n \) groups at level 2 and 3 as shown in Fig. 9. All tasks at level 1 are identical and have \( t = s = r = p = 1 \). Let \( t_i, s_i, r_i, p_i \), \( 0 \leq i \leq n \) be the parameters associated with tasks at level 2, 3, and 4. The question is to find the estimated optimal recovery point assignment.
A) Consider the 4n-processor system. Similar arguments as in the case of subproblem 1 hold for this subproblem, also. Therefore, we can modify Algorithm 1 to obtain the estimated minimum solution. A and B are one-dimensional arrays, supno = 8 for this case, T is an 8 by n array, and the computation time is $O(8n^2)$.

B) For single-processor systems, similar arguments as in the case of subproblem 1 also hold for this case. An $O(8n)$ algorithm can be used to get the minimum performance.

E. Extend to k-Level Reverse Trees

We have already inspected 3-level and 4-level trees. Now we can extend the model to k-level tree, for any fixed $k \geq 3$ (i.e., a total of $n(2^k - 1) + 1$ tasks). Since tasks at level 1 are identical and fixed, we have $2^{k-2} - 1$ tasks of variable parameters in each group. Thus there are $2^{k-2} - 1$ subcases, hence supno = $2^{k-2} - 1$. If we follow the approach of solving 3-level and 4-level reverse trees, i.e., we use algorithm 1 to solve $n2^{k-2}$-processor systems and algorithm 2 to solve uniprocessor systems, it will take $O(n2^{k-2}) = O(2^{k-2}(n^2))$ and $O(n(supno)\log(supno)) = O((2^{k-2} - 1)2^{k-2}n\log n)$ time to execute those algorithms, respectively, where $\log(supno)$ is used to compute $C_i$, $R_i$ for any specific i, j. For the k-level reverse binary tree model, the algorithm will lead to $O(2^{k/2})$ and $O(n2^{k/2})$ time for $2^{k-2}$-processor and uniprocessor systems, respectively, since $n = 2$.

VI. COMMUNICATION DELAYS

The estimated formulas can apply to the case where communication delay is significant such as in distributed system.

We can still use the procedure to compute the expected performance of a given recovery point assignment, or exploit the problem structure to get the estimated performance. This time we should consider the communication delay by including the delay time in the task execution time ($t_i$) and the rollback time ($r_i$).

A. Analysis

Let $\alpha_i$ be the processor to which task $i$ is assigned.

Recall that in the reverse tree model, each task can be invoked by many tasks, but it can invoke only one task. Since we have as many processors as the number of tasks at the first level, we can ensure that if task $i$ invokes task $j$ which is executed on a different processor, then the processor that executes task $i$ will no longer be assigned any more tasks. Furthermore, once task $i$ has been assigned to processor $j$, then task $j$ must be executed on processor $j$ whenever task $i$ is invoked. For instance Fig. 10 gives a set of tasks with $h = 3$, $N = 7$, and an example of a feasible schedule.

The estimated formulas can be slightly modified so as to include the communication delay among processors. Let $d_{i,j}$ represent the delay time caused by the task in processor $i$ invoking another task in processor $j$, where $i \neq j$ and $d_{i,j} = 0$.

Theorem 6.1: If there is a recovery point at task $i$, then

$$K_i' = r_i$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i'$$

$$C_i = \max \{ C_{i_l} + d_{a_{i_l},a_j}, C_{i_r} + d_{a_{i_r},a_j} \} + E_i + s_i,$$

otherwise,

$$K_i' = \max \{ K_{i_l}' + E_{i_l} + d_{a_{i_l},a_j}, K_{i_r}' + E_{i_r} + d_{a_{i_r},a_j} \}$$

$$E_i = t_i/p_i + (1/p_i - 1)K_i'$$

$$C_i = \max \{ C_{i_l} + d_{a_{i_l},a_j}, C_{i_r} + d_{a_{i_r},a_j} \} + E_i$$

where $d_{a_{i_l},a_j}$ (or $d_{a_{i_r},a_j}$) is the delay time to signal processor $\alpha_j$ that processor $\alpha_{i_l}$ (or $\alpha_{i_r}$) has finished task $i_l$ (or task $i_r$), $d_{a_{i_l},a_j}$ (or $d_{a_{i_r},a_j}$) is the delay time to signal processor $\alpha_j$ (or $\alpha_j$) to decide which tasks should roll back because task $i$ does not pass the acceptance test.

Proof: The proof is in Appendix D.

Consider the example given in Fig. 10. The possible timing diagram is as shown in Fig. 11.

Tasks 1, 2, 3, and 4 are first initiated at processors 1, 2, 3, and 4, respectively. After tasks 1 to 4 have passed their acceptance tests, task 5 and task 6 are initiated by processors 1 and 3. Task 5 again passes its test. However, task 6 does not pass its test. Since there is no recovery point at task 6, tasks 3 and 4 will have to roll back and recompute. After the recomputation of tasks 3 and 4, task 6 can then recompute. This time, task 6 passes its acceptance test. In the meantime, task 7 would have to wait
Recovery Point Setting: Acceptance Test

\[ \text{Delay Time} \]

Fig. 11. One possible timing diagram of Fig. 10.

```
Fig. 12. The possible schedule of the 3-level reverse tree.
```

until task 6 finishes before it can start the computation. Unfortunately, task 7 also fails its acceptance test, and tasks 3, 4, 5, and 6 have to roll back and recompute.

We can use the approach proposed in Section IV-B to find the estimated solution where that communication delays are significant. For example, Fig. 12 is a possible task schedule for problem 1.

Based on Fig. 12, we can use algorithm 1 to solve the estimated optimal recovery point assignment problem, provided the definition for \( W, X, Y, Z \) should be modified.

Let \( \mu = 2i - 1 \) and \( \nu = 2i \), then

\[
W_i = \max \{d_{\mu \nu}, d_{\nu \mu}\} + t_i/p_i + (1/p_i - 1)
\]

\[
\cdot (2 + \max \{d_{\mu \nu} + d_{\nu \mu}, d_{\nu \nu}, d_{\mu \mu}\})
\]

\[
X_i = \max \{d_{\mu \nu}, d_{\nu \mu}\} + t_i/p_i + (1/p_i - 1)r_i + s_i
\]

\[
Y_i = t_0/p_0 + (1/p_0 - 1)^* (d_{\mu \mu} + d_{\nu \nu} + t_i/p_i + 1/p_i(2 + \max \{d_{\mu \mu} + d_{\nu \nu}, d_{\nu \mu} + d_{\mu \nu}\}))
\]

\[
Z_i = t_0/p_0 + (1/p_0 - 1)^* \cdot (d_{\mu \mu} + d_{\nu \nu} + t_i/p_i + r_i/p_i).
\]

And if we consider that \( d_{\mu \mu} = 0 \), then

\[
W_i = d_{\mu \nu} + t_i/p_i + (1/p_i - 1)(2 + d_{\nu \nu} + d_{\mu \nu})
\]

\[
X_i = d_{\mu \nu} + t_i/p_i + (1/p_i - 1)r_i + s_i
\]

\[
Y_i = t_0/p_0 + (1/p_0 - 1)^* \cdot (d_{\mu \mu} + d_{\nu \nu} + t_i/p_i + 1/p_i(2 + d_{\nu \nu} + d_{\mu \nu}))
\]

\[
Z_i = t_0/p_0 + (1/p_0 - 1)^* \cdot (d_{\mu \mu} + d_{\nu \nu} + t_i/p_i + r_i/p_i).
\]
If \( d_{x} = d_{y} = d_{a} = d_{a+1} = 0 \), then the above equations are exactly the same as in the previous section.

VII. CONCLUSION

In this paper, we have studied a special case of concurrent recovery schemes where the computation is modeled by a reverse tree task model. We have obtained a closed form solution for tasks in uniprocessor systems. In multiprocessor systems, we have obtained the procedure to compute the optimal placements of recovery points. However, since no closed form solution is available, the computation will be extensive. Thus, we have suggested an estimated measurement which does have a closed form solution. However, since no closed form solution is available, the computation will be extensive. Thus, we have suggested an estimated measurement which does have a closed form solution and can be derived directly from the problem structure. This closed form is similar to the one for uniprocessor systems. We have also proposed algorithms to compute the estimated solution. Specifically, we have proposed algorithms for 3-level, 4-level, and \( k \)-level reverse tree models. The estimated formula can be extended to the case where communication delays are significant.

Theorem 3.1

Inverse tree models. The estimated formula can be extended to the case where communication delays are significant such as in distributed systems where the previously proposed algorithm can still apply. Since the complexity of the special case, i.e., the reverse tree task model is so great for general computation models such as arbitrary concurrent task models, a simulation technique such as [11] may be a better approach.

APPENDIX A

The Proof of Theorem 3.1

We list the possible events of task \( i \) in the following table.

<table>
<thead>
<tr>
<th>Event</th>
<th>Probability</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passes its AT without failure</td>
<td>( p_i )</td>
<td>( t_i )</td>
</tr>
<tr>
<td>Fails its AT once, then passes it</td>
<td>( p_i (1 - p_i) )</td>
<td>( t_i + K_i + t_i )</td>
</tr>
<tr>
<td>Fails its AT twice, then passes it</td>
<td>( p_i (1 - p_i)^2 )</td>
<td>( 2(t_i + K_i) + t_i )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Fails its AT ( j ) times, then passes it</td>
<td>( p_i (1 - p_i)^j )</td>
<td>( j(t_i + K_i) + t_i )</td>
</tr>
</tbody>
</table>

So,

\[
E_i = \sum_{j=1}^{\infty} p_i (1 - p_i)^j \left( j(t_i + K_i) + t_i \right) = \frac{t_i}{p_i} + \frac{(1/p_i - 1) K_i}{p_i}
\]

Case 1: There is a recovery point at task \( i \); then,

\[
E_i = \frac{t_i}{p_i} + \frac{(1/p_i - 1) K_i}{p_i}
\]

Case 2: There is no recovery point at task \( i \).

If a fault occurs at task \( i \), then certain tasks will have to roll back. These tasks are described below. Consider each ancestor \( j \) of \( i \) that has a recovery point and it is not the case where there are any other tasks on the path from \( j \) to \( i \) which also has a recovery point. When task \( i \) detects a fault, it signals its parents to roll back, and its parents in turn signal their parents to roll back until all the ancestors up to task \( j \) is informed. Then task \( j \) will first recompute. After task \( j \) finishes, it will then inform its child so that the latter can resume computation. This process continues until task \( i \) recomputes. This means that the root of the left and right subtrees of \( i \) will roll back before task \( i \) rolls back. Therefore,

\[
K_i = K_0 + E_0 + K_{rr} + E_{rr}
\]

\[
E_i = \frac{t_i}{p_i} + \frac{(1/p_i - 1) K_i}{p_i}
\]

\[
C_i = C_{rr} + C_{rr} + E_i + s_i
\]

APPENDIX B

The Proof of Theorem 4.1

1) is true since \( P_Y^Y_{ij} \) for all \( i, j, k \), includes the probabilities of all possible cases.

2) is true since task 1 fails its AT \((i - 1)\) times, \((= (1 - p_1)^{i-1})\) and passes it at the \( i \)th try, \((= p_1)\). Tasks 2 fails its AT \((j - 1)\) times, \((= (1 - p_2)^{j-1})\) and passes it at the \( j \)th try \((= p_2)\). Then task 3 passes its AT at the first try \((= p_3)\).

3) Task 3 has failed its AT \((k - 1)\) times before it passes the AT at the \( k \)th try. Every time when task 3 fails its AT, task 1 and task 2 have to execute again. Hence between the \((\alpha - 1)\)th and \(\alpha\)th try of task 3, both task 1 and task 2 execute at least once, for \(1 \leq \alpha \leq k\).

If we let \( i_0 \) (or \( j_0 \)) denote the number of times that task 1 (or task 2) has executed between the \((\alpha - 1)\)th and \(\alpha\)th try of task 3, where \(1 \leq \alpha \leq k\), then we have \((1) i_0 \geq 1, j_0 \geq 1; (2) \sum_{a=1}^{\alpha} i_0 = i, \sum_{a=1}^{\alpha} j_0 = j\).

Hence each possible set of partition-and-permutations of \( i \) and \( j \) into \( k \) parts one-to-one corresponds one possible case which constitute \( P_Y^Y_{ij} \).

Since task 3 fails its AT’s before the \( k \)th try, so

- if \( \alpha \neq k \), the probability that task 1 fails its AT \( i_0 \) - 1 times and succeeds once, task 2 fails its AT \( j_0 \) - 1 times and succeeds once, and task 3 fails its AT once, equals \( P_{i_0}^+ P_{j_0}^+ \). \( (1 - p_3/p_3) \).

- if \( \alpha = k \), the probability that task 1 fails its AT \( i_0 \) - 1 times and succeeds once, task 2 fails its AT \( j_0 \) - 1 times and succeeds once, and task 3 succeeds its AT once, equals \( P_{i_0}^+ \).

Hence for the specific set of partition-and-permutations of \( i \) and \( j \), we have the probability \( P_{i_0}^+ P_{j_0}^+ \cdots P_{i_0}^+ \). \( (1 - p_3/p_3)^{k-1} \).

Summing over all possible sets of partition-and-permutations of \( i \) and \( j \), we have

\[
P_Y^Y_{ij} = \sum_{\text{all possible sets of partitions}} P_{i_0}^+ P_{j_0}^+ \cdots P_{i_0}^+ \frac{(1 - p_3)}{p_3}^{k-1}
\]

\[
= \sum_{\text{all possible sets of partitions}} \prod_{a=1}^{k} P_{i_0}^+ \frac{(1 - p_3)}{p_3}^{k-1}
\]

4) The solution is equivalent to those of the following combinatorial problems: i) the number of ways to put \( i \) identical balls into \( k \) distinct boxes, and ii) the number of
positive integer solutions of \( y_1 + y_2 + \cdots + y_k = i \). Therefore, the number of ways equals
\[
\begin{align*}
(i - k + k - 1) & = (i - 1) \\
(k - 1) & = (i - 1)
\end{align*}
\]

5) Task 3 has been executed \( k \) times. Therefore task 1 passes its AT \( k \) times, task 2 \( k \) times, and task 3 once. That means task 1, 2, and 3 fail their corresponding AT’s \( i - k, j - k, \) and \( k - 1 \) times, respectively. This results in
\[
p_1^i(1 - p_1)^{i-k} p_2^j(1 - p_2)^{j-k} (1 - p_3)^{k-1} = p_3^k.
\]

There is a one-to-one correspondence between the partition-and-permutations of \( i \) (or \( j \)) into \( k \) positive parts and the possible cases for \( \pi'_j \). For a given partition-and-permutation of \( i \) into \( k \) positive parts, say \( a_1, a_2, \ldots, a_k \), \( \pi'_j \) relates to the number of times that task 1 executes between the \((\alpha - 1)\)th and \( \alpha \)th attempt of task 3. Hence the probability for each such set of partition-and-permutation equals
\[
\left[ \prod_{\alpha + 1}^{k} \frac{1 - p_3}{p_3} \left( \frac{1 - p_3}{p_3} \right)^{k - 1} \right] = \left[ \prod_{\alpha + 1}^{k} \frac{1 - p_3}{p_3} \left( \frac{1 - p_3}{p_3} \right)^{k - 1} \right] \cdot \left( \frac{1 - p_3}{p_3} \right)^{k - 1} = (1 - p_1)^{i-k} p_2^k(1 - p_3)^{k-1} p_3.
\]

There are
\[
\begin{align*}
\left( i - 1 \right) & = (i - k) \\
(j - 1) & = (j - k)
\end{align*}
\]
possible sets of partition-and-permutations, so we have
\[
p_2^i = \left( i - 1 \right) (j - 1) \cdot p_1(1 - p_1)^{i-k} p_2^k(1 - p_3)^{k-1} p_3.
\]

**Example:**
\[
P_{41} = (1 - p_1)^3 p_1 p_2 p_3
\]
\[
P_{33} = (P_{11} p_{21} + P_{11} p_{21} + P_{12} p_{21} + P_{12} p_{21} + P_{12} p_{11}) \left( \frac{1 - p_3}{p_3} \right) = 4(1 - p_1)^3 p_2^2(1 - p_3) p_3.
\]

**APPENDIX C**

**CORRECTNESS OF ALGORITHM 1**

Because of the counter variable \( cn \), only after each task index has appeared at least once in table \( T \), Algorithm 1 can get into the inner loop to map array \( B \) onto table \( T \), and gets a candidate solution each time it enters. After the first entrance to the inner loop, Algorithm 1 will always enter it again when a new entry from \( A \) is entered into \( T \). Algorithm 1 terminates when all the elements in array \( A \) have been put into table \( T \).

Note that the \( C \)’s can be viewed as
\[
C = \max \{ \text{first group} \} + \max \{ \text{second group} \} + 2.
\]

The first group consists of \( W \)’s and \( X \)’s, and the second group consists of \( Y \)’s and \( Z \)’s. There may be several solutions \( I_1, I_2, \ldots, I_{\text{max_of_solution}} \) we can pick an \( I_{\text{special}} \) among \( I \)’s, such that \( j \) is the smallest index of array \( A \) and \( A[j] \) constitutes the maximum term in the first group of \( I_{\text{special}} \), and \( n \leq j \leq 2n \). Under such \( j \), let \( l \) be the smallest index of array \( B \), such that \( B[l] \) constitutes the maximum term in the second group of \( I_{\text{special}} \), and \( n \leq l \leq 2n \). However, if there are more than one largest element in the second group, then let \( l \) be the largest element among them.

**Fact:** In \( A[1], A[2], \ldots, A[j] \), each task index has appeared at least once. The same holds for \( B[1], B[2], \ldots, B[l] \). If there is a \( W \)’s in the first group, then there is a corresponding \( Y \)’s in the second group. So is an \( X \)’s corresponding to a \( Z \)’s.

Now we show that Algorithm 1 can find a solution which is at least as good as \( I_{\text{special}} \). Because of the outer loop, the corresponding entry of \( A \) is entered into \( T \) will eventually be set to 1, and then Algorithm 1 enters the inner loop. Let \( B[k] \) be the last element that makes Algorithm 1 execute statement 14, or 16, i.e., after considering \( B[k] \), all elements of \( assign \) are filled. Note that \( n \leq k \leq 2n \), since each task index must be represented. We have three cases to consider.

**Case 1:** \( k < l \).

Then the largest element of array \( B \) found by Algorithm 1 is \( B[k] \), and \( B[k] \leq B[l] \). If \( B[k] < B[l] \), then the maximum element in the first group is less than or equal to \( A[j] \) and the maximum in the second group is \( B[k] \) which is less than \( B[l] \). Thus, \( I_{\text{special}} \) is not estimated optimal, a contradiction. If \( B[k] = B[l] \), then the maximum element in the second group is equal to \( B[l] \), and the maximum in the first group is either less than or equal to \( A[j] \). If the maximum element is less than \( A[j] \), then we find a solution which is better than \( I_{\text{special}} \), a contradiction. If the maximum term is equal to \( A[j] \), then we find a solution that violates the assumption that either \( j \) is the smallest index of array \( A \) or \( l \) is the smallest index of array \( B \).

**Case 2:** \( k = l \).

Then \( B[l] \) must be in the second group by Algorithm 1, but \( A[j] \) may not be selected. If \( A[j] \) is selected, then \( A[j] \) and \( B[l] \) are the maximum elements of the first and second groups, respectively. Then Algorithm 1 finds the estimated optimal solution. If \( A[j] \) is not selected, then the maximum element is either less than or equal to \( A[j] \). If it is less than \( A[j] \), then we find a solution which is better than \( I_{\text{special}} \), a contradiction. If it is equal to \( A[j] \), then it violates the assumption that \( j \) is the smallest index,
Case 3: $k > l$. 
Since $B[1]$ is the largest element in the second group, there exists an estimated optimal solution, i.e., $I_{special}$, that contains all the index up to and including $A[j]$ and $B[1]$ in array $A$ and $B$, respectively. According to the fact stated earlier, we have to consider two cases. In the first case, all the exact elements in $I_{special}$ are chosen by Algorithm 1, then Algorithm 1 does not go beyond $l$ in array $B$, a contradiction. The second case, at least one pair of elements in $I_{special}$ is not selected by Algorithm 1. Without loss of generality, we can assume the elements not selected are $W_i$ and $Y_j$, therefore $Z_k$ is in front of $Y_j$ in array $B$, and $X_k$ must be in front of $A[j]$ including $A[j]$ in array $A$. Thus, $Z_k$ is before $B[1]$ in array $B$. By the same reasoning all the elements that are not selected by Algorithm 1 can be found before $B[1]$. Recall all the elements in $I_{special}$ can be selected by $B[1]$, thus, Algorithm 1 should stop either before or at $B[1]$, a contradiction. This proves the correctness of Algorithm 1.

APPENDIX D

THE PROOF OF THEOREM 6.1

Case 1: Task $i$ has a recovery point; then,

$$K'_i = r_i$$

and $\max \{ C_t + d_{aux}, C_r + d_{aux} \}$ can be obtained from the fact that either left or right subtree will signal task $i$ to start execution.

Case 2: Task $i$ has no recovery point.

It takes $d_{aux}$ (or $d_{aux}$) time to invoke processor $\alpha_t$ (or $\alpha_r$) to transmit a signal or to roll back. And it is necessary for processor $\alpha_t$ (or $\alpha_r$) to inform processor $\alpha$, to resume execution. So the estimated resume time $K'_i$ should be

$$\max \{ K^*_{t_l} + E_t, d_{aux} + d_{aux}, K^*_{r_r} + E_r + d_{aux} + d_{aux}, \}.$$

ACKNOWLEDGMENT

We thank T. Jiang for providing the complexity to solve the recurrence relation in Section III-B, K. W. Hwang for excellent discussions, and Jack Tan for his help in the preparation of this paper. The support of Honeywell Inc. is gratefully acknowledged. We also thank the anonymous referees whose comments greatly improved the presentation of this paper.

REFERENCES