Optimal Scheduling of Imprecise Computation Tasks in the Presence of Multiple Faults *

Hakan Aydın, Rami Melhem, Daniel Mossé
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
(aydin, melhem, mosse)@cs.pitt.edu

Abstract

With the advance of applications such as multimedia, image/speech processing and real-time AI, real-time computing models allowing to express the "timeliness versus precision" trade-off are becoming increasingly popular. In the Imprecise Computation model, a task is divided into a mandatory part and an optional part. The mandatory part should be completed by the deadline even under worst-case scenario; however, the optional part refines the output of a mandatory part within the limits of the available computing capacity. A non-decreasing reward function is associated with the execution of each optional part. Since the mandatory parts have hard deadlines, provisions should be taken against faults which may occur during execution. An FT-Optimal framework allows the computation of a schedule that simultaneously maximizes the total reward and tolerates transient faults of mandatory parts. In this paper, we extend the framework to a set of tasks with multiple deadlines, multiple recovery blocks and precedence constraints among them. To this aim, we first obtain the exact characterization of Imprecise Computation schedules which can tolerate up to k faults, without missing any deadlines of mandatory parts. Then, we show how to generate FT-Optimal schedules in an efficient way. Our solution works for both linear and general concave reward functions.

1 Introduction

Real-time computing models which are able to express the "timeliness versus precision" trade-off are attracting more attention with the advance of applications such as multimedia, image/speech processing, information gathering and real-time AI. Imprecise Computation (IC) [9, 12], Increased-Reward-with-Increased-Service (IRIS) [6] and Q-RAM [10] models address the problem by logically dividing each task into a mandatory part and an optional part. Despite differences, some fundamental traits are shared by these models: The optional part need not to be completed by the deadline, instead its (partial or complete) execution *refines* the approximate/initial result produced by the mandatory part. A non-decreasing reward/utility function (alternatively, a non-increasing error function) is associated with the execution of the optional part to quantify the precision or refinement of the final output. While IC studies have mostly dealt with linear reward functions [9, 12], IRIS [6] and Q-RAM [10] models extended the framework to general concave reward functions. Linear and general concave reward functions can successfully represent most of the applications, since most realistic applications (such as multimedia, image processing, real-time decision making) exhibit non-increasing marginal return behavior during the execution, as in [13, 8, 7]. In addition, we have shown that relaxation of the concavity assumption results in an NP-Hard problem [1].

Although these reward-based models allow for greater scheduling flexibility when compared to traditional hard real-time models, it should be emphasized that the mandatory parts have still stringent timing constraints. As such, not only the mandatory parts should be guaranteed timely completions under worst-case scenarios, but also provisions must be taken against unexpected events, such as faults. First studies considering fault tolerance issues in IC schedules appeared in [4, 5]. Recently, we introduced an FT-Optimal scheduling framework for IC tasks, which involved the computation of the schedule with maximum reward among *all* possible fault tolerant (FT) schedules [2]. The framework was based on *exploiting the time redundancy provided by optional parts to recover mandatory tasks* and had several desirable properties in that it: (a) assumed only knowledge about the maximum number of faults of the entire task set, (b) considered general concave reward functions, and (c) required no on-line adjustment as long as no faults were encountered.

The work in [2], although it considered both independent tasks and tasks with linear precedence constraints, was limited to frame-based systems with a single end-to-end deadline.

*This work has been supported by the Defense Advanced Research Projects Agency (Contract DABT63-96-C-0044).

1530-1427/00 $10.00 © 2000 IEEE

The case of independent tasks with different deadlines was further explored in [3]. Two recovery schemes, namely *immediate* and *delayed recovery*, were proposed and it was shown that the FT-Optimality problem of independent tasks with delayed recovery and multiple deadlines was an intractable problem even under modest assumptions. Immediate recovery with non-identical ready times was also shown to result in an NP-Hard problem, and a pseudo-polynomial time algorithm was developed for linear reward functions and identical ready times.

This paper extends the FT-Optimality research of [2] for tasks with linear precedence constraints in two ways. First, we consider non-identical deadlines, which needs little justification. Second, for multiple faults, we no longer assume that all the recovery blocks associated with a given task have the same worst-case execution time. Non-identical recovery blocks are more realistic: for example, one or two recovery blocks may try simply to re-execute the original code and execute an alternate code for the task, then a final recovery block may try to load a 'safe state' onto memory. The use of exact worst-case execution times per recovery block, instead of relying on a maximum amount for recovery block for each task, may considerably improve the total reward, as illustrated in Section 3.

After introducing the system model and basic terminology in Section 2, we provide a motivating example in Section 3. We present our main results in two parts: In Section 4, we address the problem of **efficiently checking the feasibility of a given IC schedule under any pattern of k faults with (potentially) multiple recovery blocks per task.** Besides its importance per se, this section also lays ground for Section 5, where we show **how to generate k-FT Optimal schedules in an efficient manner for general concave reward functions.** We conclude with a discussion of this research effort and future work considerations.

2 System Model

2.1 Task Model

We consider a set $\mathbf{T}$ of n imprecise computation tasks $T_1, T_2, \ldots, T_n$, on a uniprocessor system. The deadline of task T_i is denoted by d_i. Without the loss of generality, we assume that T_1 is the task with the earliest deadline, T_2 is the second and so on (ties are broken arbitrarily). Each task T_i consists of a mandatory part M_i and an optional part O_i. The length of the mandatory part is denoted by m_i; each task must receive at least m_i units of service time in order to provide output of acceptable quality. The optional part O_i becomes ready for execution only when the mandatory part M_i completes.

We assume precedence constraints between tasks; that is, first M_1 and O_1 execute, then M_2 and O_2, and so on; the execution completes with M_n and O_n. Without loss of generality, M_1 is assumed to be ready at $t = 0$. Note that besides tasks with linear precedence constraints, this model can also capture the IC scheduling of *independent* tasks with identical ready times in a non-preemptive environment; it is not difficult to show by a swapping argument that executing each task (which consists of a (M_i, O_i) pair) in increasing order of deadlines is the optimal policy if preemption is not allowed.

Associated with each optional part of a task is a reward function $R_i(t)$ which indicates the reward accrued by task T_i when it receives t units of service *beyond its mandatory portion*. $R_i(t)$ is of the form:

$$R_i(t) = \begin{cases} F_i(t) & if \quad 0 \leq t \leq o_i \\ F_i(o_i) & if \quad t > o_i \end{cases} \tag{1}$$

where F_i is a nondecreasing, concave and continuously differentiable function over nonnegative real numbers and o_i is the length of *entire* optional part O_i. In this formulation, when the task's optional execution time t reaches the threshold value o_i, the reward accrued ceases to increase.

Given a task set $\mathbf{T}$, a *schedule* for $\mathbf{T}$ determines the amount of service each task receives. S is a *feasible* schedule for $\mathbf{T}$, if and only if in S,

i) Each mandatory task M_i completes before its deadline d_i,

ii) No optional task O_i is executed beyond d_i, although partial optional executions are acceptable,

iii) Precedence constraints are adhered to.

Further, given a schedule S, define the following functions:

- $Start_S(M_i)$: The time at which M_i is scheduled to start execution.
- $End_S(M_i)$: The time at which M_i is scheduled to complete execution.
- $[\omega_1, \omega_2]_S$: The portion of schedule S between time points ω_1 and ω_2.

As an example, Figure 1 illustrates a schedule where $Start_S(M_2) = 15$ and $End_S(M_2) = 20$. Note that $End_S(M_i) - Start_S(M_i) = m_i \ \forall\, i$, since preemption never occurs.

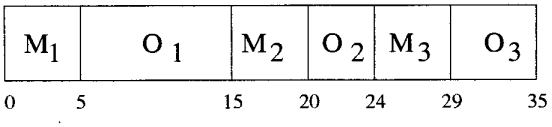

Figure 1. A schedule of IC tasks

The *Total Reward* of an imprecise computation schedule S is $REW_S = \sum_{i=1}^{n} R_i(t_i)$ where t_i is the amount of service that optional part O_i receives in S. A feasible schedule is *optimal* if it *maximizes* the total reward accrued.

2.2 Fault Model

We assume that at most k faults may occur during the execution of the task set. The results are produced or committed

at the end of M_i and then again at the end of O_i. Consistency or acceptance checks are performed before the results are committed. If an error in a task T_i is detected at the end of its mandatory part M_i, then the system initiates the *recovery mode*, where a recovery block $B_{i,1}$ is immediately executed. A recovery block [11] may simply re-execute M_i or execute an alternative code. Should the error persist or another error be detected at the end of $B_{i,1}$, the second recovery block of T_i, namely $B_{i,2}$, is executed and so forth. A fault of M_i or of any of its recovery block is referred to as "a fault of task T_i" for the sake of simplicity.

The worst-case execution time of the recovery block $B_{i,j}$ is denoted by $b_{i,j}$. Note that there might be at most k recovery blocks for a given task, since we allow no more than k faults in the system. If an error is detected at the end of the optional part O_i, the result is not committed; but the recovery mode is not started either: the execution of M_{i+1} uses the approximate result produced by M_i. Since the recovery mode is initiated only for mandatory parts, throughout the paper we will use the expression "j^{th} fault during the execution" to refer to j^{th} error detected at the end of any mandatory part (or recovery block).

A *multiple fault pattern* is a set $P = \{p_1, p_2, \ldots, p_n\}$, where p_i denotes the *actual* number of faults that task T_i incurs and such that, $\sum_{i=1}^{n} p_i \leq k$.

In general, a schedule is said to be *k-fault tolerant (k-FT)* if every mandatory part M_i and any of its executed recovery blocks, can complete by d_i under *any* k-fault pattern P. Note that k-fault tolerance of a schedule implies its feasibility, but the converse is not true.

Finally, a schedule S is *k-FT Optimal* if and only if:

i) it is *k-FT*, and

ii) its total reward is maximum among all *k-FT* schedules.

As in [2, 3], in this study, the recovery operations will have to rely on the existence of optional parts which provide a sort of slack, due to the their non-essential nature. Hence, once recovery is initiated, the remaining optional parts may not have a chance to execute. Yet, checking the tolerance of a schedule to any pattern of k faults in the existence of multiple deadlines and recovery blocks, and further, determining CPU allotments of optional parts to simultaneously guarantee k fault tolerance and maximize system reward are non-trivial problems.

3 k-Fault Tolerant Optimal Scheduling: A Motivating Example

In this section, we present an example with three tasks. We assume that only two faults need to be tolerated (i.e., $k = 2$). Further, although non-linear reward functions are more realistic, the tasks in the example have linear reward functions, for the sake of simplicity. Despite its simplicity, the example illustrates many facets of scheduling imprecise computation tasks in the presence of multiple faults.

id	m_i	o_i	d_i	$b_{i,1}$	$b_{i,2}$	$F_i(t)$
T_1	5	25	25	5	3	$5t$
T_2	5	10	30	5	1	$4t$
T_3	5	20	35	5	1	t

As it can be seen, each task has two recovery blocks. The first recovery blocks are merely re-executions ($b_{i,1} = m_i$, $\forall i$). The worst-case execution times of the second recovery blocks are much less than the first ones. Observe that the marginal return of O_1 is the largest among all tasks, hence, the schedule with the highest reward consists in favoring O_1 as much as possible (Fig. 2); call this schedule S_1.

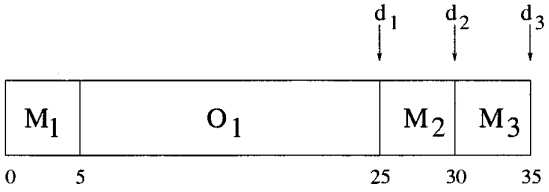

Figure 2. An optimal but non-FT Schedule

S_1 is optimal (total reward of 100), but it is not FT: M_2 and M_3 complete just before their deadlines and it is not possible to tolerate even a single fault which occurs in either of them.

Provided that a task set is k-FT, one can always obtain a trivial FT-schedule by completing every mandatory part as soon as possible, and the remaining idle time will be assigned to the *last* optional part. If we adopt this strategy, we end up by getting S_2, shown in Fig. 3.

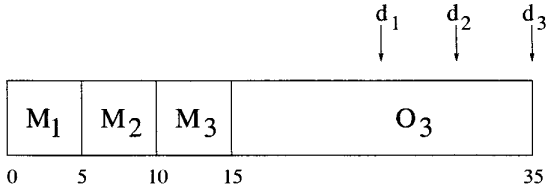

Figure 3. A trivial FT Schedule

However, the total reward of S_2 is only 20, due to low marginal return of O_3. The study in [2] enforced that the sum of optional assignments after a given task (and before its deadline) should be at least equal to the recovery time that it may needed. Further, each recovery block was assumed to have the same worst execution time. However, this approach is inadequate for multiple deadlines, in that it excludes some legal k-FT schedules. Indeed, using that strategy, the schedule with the highest reward is S_3 (Fig. 4, total reward of 55).

Yet, there is another schedule S_3 with considerably higher reward (72), shown in Figure 5, which tolerates any pattern of k faults in addition to taking into account non-identical deadlines and non-identical recovery blocks. Yet, it can be easily verified that it is $k-$FT and, in fact, S_3 is a k-FT Optimal schedule for the example task set.

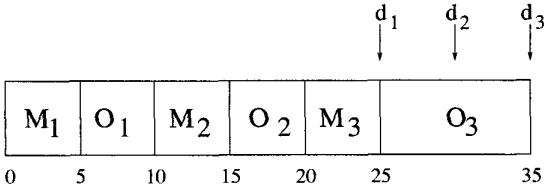

Figure 4. A k-FT Schedule with higher reward

In summary, from this example it is clear that new and effective fault tolerance criteria are needed for the case of task sets with multiple deadlines and different recovery blocks.

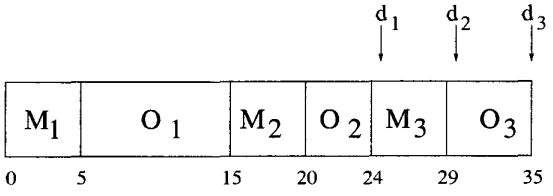

Figure 5. The k-FT Optimal Schedule

4 Exact characterization of k-FT Schedules

In this section, we consider and solve the following problem: **Given a schedule S of imprecise computation tasks, how can we efficiently check its tolerance to any pattern of k faults?**

Naturally, one can solve the problem by generating all schedules corresponding to the $O(n^k)$ fault patterns and checking the timeliness of each of them in $O(n)$ time, which yields a total time complexity of $O(n^{k+1})$. This is clearly an unacceptable computational cost. The solution that we present stems from a rigorous characterization of FT schedules and it is simultaneously efficient and exact. Besides, such a rigorous characterization of *all* the FT schedules for a given task set gives us valuable insights to achieve our ultimate aim of generating FT-Optimal schedules.

Definition: *An IC schedule S is called* k-FT at level i *if it remains feasible after recoveries of any fault pattern P where $p_j = 0, \quad \forall\, j < i$.*

Informally, level i k-fault tolerance enforces that the schedule remain feasible if *all* the faults occur during or after the execution of M_i (for convenience, we define $p_0 = 0$). The following propositions can be easily verified in view of the above definition.

Proposition 1 *A schedule S is k-FT at level* n *if and only if the schedule remains feasible in case that all the k faults occur in the last task M_n (or its recovery blocks).*

Proposition 2 *A schedule S is k-FT at level $i < n$ if and only if:*

1. *it is k-FT at level $i+1$, and,*
2. *under any k fault pattern which begins with j faults of M_i, the tasks $M_i, M_{i+1}, \ldots, M_n$ and their recovery blocks complete before their corresponding deadlines.*

Since any k-fault pattern should begin with an error detected at the end of a specific M_i $(1 \leq i \leq n)$, Proposition 2 lets us to deduce the following:

Proposition 3 *A task set* T *is k-FT if and only if there exists a schedule which is k-FT at level 1.*

A strict characterization of k-fault tolerance will be obtained iteratively by studying tolerance of consecutive levels. Specifically, we will first obtain necessary and sufficient conditions for a level n k-FT system, then for level $n-1$ and so on, until we reach level 1. Clearly, the first condition is that the schedule should be k-FT at level n, that is, M_n should be scheduled early enough to leave time for recoveries of k faults of M_n. Similarly, k fault tolerance at level $n-1$ requires that M_{n-1} be scheduled early enough to let timely recoveries of *any k faults* which affect M_n and M_{n-1}, and so on. Note that while computing the upper bound on the scheduling time of M_i, we should consider the worst-case scenario which requires maximum recovery time after $End(M_i)$, which corresponds to a k-fault pattern affecting only $M_i, M_{i+1}, \ldots, M_n$. Let us define a function over individual tasks to resume our analysis.

Definition: *The function $LCT(M_i)$, denotes the latest completion time of M_i in any schedule for* T*, which allows the timely completions (and recoveries) of the mandatory parts $M_i, M_{i+1}, \ldots, M_n$ under any k-fault pattern which begins with a fault on M_j $(j \geq i)$, that is, $p_h = 0 \;\; h = 1, \ldots, i-1$.*

Informally, $LCT(M_i)$ is a measure of the maximum amount M_i can be pushed late in the schedule, without compromising level i k-Fault Tolerance. Using Propositions 2 and 3, we obtain:

Proposition 4 *A schedule S is k-FT at level i if and only if $End_S(M_j) \leq LCT(M_j) \;\; j = i, \ldots, n$.*

Corollary 1 *A task set is k-FT if and only if there exists a schedule S where $End_S(M_j) \leq LCT(M_j)$, $j = 1, \ldots, n$.*

Therefore, once $LCT(M_i)$ is determined for every task, the k-fault tolerance of a given schedule can be decided in $O(n)$ time. Hence, the problem is now reduced to the efficient computation of $LCT()$ values for each task.

Computation of $LCT()$ Values:

We will adopt the dynamic programming approach to compute the LCT values in a bottom-up manner. First, we define an auxiliary function.

Definition: *The function $lct(M_i, u, v)$ denotes the latest completion time of M_i in any schedule for* **T**, *which allows the timely completions (and recoveries) of the mandatory parts $M_i, M_{i+1}, \ldots, M_n$ under any fault scenario where there are exactly u fault(s) during the execution of $M_1, \ldots, M_{i-1}$ and exactly v fault(s) during the execution of $M_i, \ldots, M_n$, such that $u + v = k$.*

Again, the $lct()$ function puts an upper bound on the amount of shift a task M_i can tolerate in a schedule, in a fault pattern with exactly u faults before it, and exactly v faults on and after it. Note that if M_i itself actually incurs exactly y faults in that specific scenario, it should be scheduled to complete early enough to be able to execute y recovery blocks before d_i, as well as not to push the tasks which follow beyond what is mandated by level i k-Fault Tolerance. Notice that the value of the $LCT(M_i)$ function is (by definition) simply $lct(M_i, 0, k)$. Let us start by evaluating the lct value for the last task M_n. In case that exactly i faults occur in M_n, we should have time to execute all the first i recovery blocks before the deadline d_n, which gives:

$$lct(M_n, k-i, i) = d_n - \sum_{j=1}^{i} b_{n,j} \quad (2)$$

Clearly, $LCT(M_n)$ is the minimum among all $lct(M_n, k-y, y)$ values, which correspond to the case where all the faults occur in or after M_n (in this case, all faults will occur *in* M_n):

$$LCT(M_n) = lct(M_n, 0, k) = d_n - \sum_{j=1}^{k} b_{n,j} \quad (3)$$

For $i < n$, the lct function will be computed by a dynamic programming approach:

$$lct(M_i, k, 0) = min\{d_i, lct(M_{i+1}, k, 0) - m_{i+1}\} \quad (4)$$

The above formula expresses the fact that, assuming that all the faults occur before M_i, $lct(M_i, k, 0)$ is either d_i, or the latest *start* time of M_{i+1} under this scenario (which is $lct(M_{i+1}, k, 0) - m_{i+1}$), whichever is the smallest. Similarly:

$$lct(M_i, k-1, 1) = min\{X_0, X_1\}$$

$$X_0 = min\{d_i, lct(M_{i+1}, k-1, 1) - m_{i+1}\}$$

$$X_1 = min\{d_i, lct(M_{i+1}, k, 0) - m_{i+1}\} - b_{i,1}$$

Above, X_0 corresponds to the scenario where the single fault which is supposed to occur during the execution of $M_i, \ldots, M_n$, does *not* occur in M_i, but later; while X_1 captures the case where the fault occurs in fact in M_i in which case no fault can occur in $M_{i+1}, \ldots, M_n$. We can obtain:

$$lct(M_i, k-v, v) = \min_{j=0}^{v}\{X_j\} \; v = 0, \ldots, k \text{ where :}$$

$$X_j = min\{d_i, lct(M_{i+1}, k-(v-j), v-j) - m_{i+1}\} - \sum_{y=1}^{j} b_{i,y}$$

for the general case. Note that the set $\{X_j\}$ above expresses all *possible* k-fault scenarios where we have exactly $k - v$ faults before M_i and exactly v faults on or after M_i: M_i can incur exactly j faults and the tasks $M_{i+1}, \ldots, M_n$ can incur exactly $v - j$ faults. Once again, $LCT(M_i)$ can be computed by taking the minimum among all $lct(M_i, k-y, y)$ values, which always happens to be $lct(M_i, 0, k)$ — the scenario where all the faults occur during the execution of $M_i, \ldots, M_n$. Observe further that $lct(M_i, k-v, v)$ values for $v < k$ are not used for the computation of $LCT(M_i) = lct(M_i, 0, k)$, instead they contribute to the computation of $LCT(M_j)$ $(j < i)$.

What is the complexity of computing $lct()$ values for a given task set? First, the computation of $\sum_{y=1}^{j} b_{i,y}$ $i = 1, \ldots, n$; $j = 0, \ldots, k$ can be accomplished in $O(n \cdot k)$ time and we can store them in a look-up table. Further, there are only $n \cdot (k+1)$ function values to be computed. Each of them can be done in $O(k)$ time (we need to perform at most $O(k)$ comparisons and arithmetic operations), which suggests that the overall complexity is $O(n \cdot k^2)$.

Once we compute $LCT(M_i)$ values, we can quickly test the k-fault tolerance of a given schedule S, by checking whether $End_S(M_i) \leq LCT(M_i)$ $i = 1, \ldots, n$. We conclude the analysis by pointing out that, if during the checking procedure we observe the existence of an $LCT(M_i)$ value such that $LCT(m_i) - m_i < 0$, it immediately implies that there are **no k–FT schedules for the given task set**: there can be no $End_S(M_i)$ values which can satisfy the requirement of Corollary 1, if this is the case. In other words, the test has the additional property of being able to detect task sets for which no k-FT schedules exist.

Example

Let us illustrate the application of the technique on the example task set of Section 3. We start by the last mandatory task, which is M_3. By applying Equation (2), we find $lct(M_3, 2, 0) = 35$, $lct(M_3, 1, 1) = 30$ and $lct(M_3, 0, 2) = 29$. Hence, the worst-case scenario for level 3 fault tolerance is the occurrence of two faults in M_3; and $LCT(M_3) = lct(M_3, 0, 2) = 29$. For M_2, we compute:
$lct(M_2, 2, 0) = min\{d_2, lct(M_3, 2, 0) - m_3\} = 30$,
$lct(M_2, 1, 1) = min\{X_0, X_1\} = 25$; where
$X_0 = min\{30, lct(M_3, 1, 1) - m_3\} = 25$ and
$X_1 = min\{30, lct(M_3, 2, 0) - m_3\} - b_{2,1} = 25$.

Note that X_0 above corresponds to the fault pattern in which the single fault supposed to occur on or after M_2, actually occurs later, not in M_2. In contrast X_1 captures the case where M_2 actually suffers a fault but M_3 does not; we should incorporate the execution of the first recovery block $B_{2,1}$ in the computation. As it turns out, the two scenarios impose the same upper bound on the $lct()$. Lastly, $lct(M_2, 0, 2)$ involves the consideration of three different scenarios:
$lct(M_2, 0, 2) = min\{X_0, X_1, X_2\} = 20$; where
$X_0 = min\{30, lct(M_3, 0, 2) - m_3\} = 24$ (both faults occur in M_3);
$X_1 = min\{30, lct(M_3, 1, 1) - m_3\} - b_{2,1} = 20$ (one fault occurs in M_2 and the other in M_3.)
$X_2 = min\{30, lct(M_3, 2, 0) - m_3\} - b_{2,1} - b_{2,2} = 24$ (both

faults occur in M_2).

From these results, $LCT(M_2)$ is evaluated to be equal to X_1, which is 20. Observe that, for level 2 fault tolerance, the worst-case scenario corresponds to the one where both M_2 and M_3 fail once. Finally, we focus on M_1:
$lct(M_1, 2, 0) = min\{d_1, lct(M_2, 2, 0) - m_2\} = 25$,
$lct(M_1, 1, 1) = min\{X_0, X_1\} = 20$; where
$X_0 = min\{25, lct(M_2, 1, 1) - m_2\} = 25$ and
$X_1 = min\{25, lct(M_2, 2, 0) - m_2\} - b_{2,1} = 20$.

Computing $lct(M_1, 0, 2)$ also requires a comparison of three values:
$lct(M_1, 0, 2) = min\{X_0, X_1, X_2\} = 15$; where
$X_0 = min\{25, lct(M_2, 0, 2) - m_2\} = 15$ (both faults occur after M_1);
$X_1 = min\{25, lct(M_2, 1, 1) - m_2\} - b_{1,1} = 15$ (one fault occurs in M_1 and the other one after it.)
$X_2 = min\{25, lct(M_2, 2, 0) - m_2\} - b_{1,1} - b_{1,2} = 17$ (both faults occur in M_1).

Level 1 Fault-tolerance requires that M_1 be scheduled no later than $t = 15$, otherwise a single fault of M_1 and a single fault of M_2 would result in a deadline miss. In other words, the $LCT()$ bounds are evaluated as:

$$LCT(M_1) = 15, \quad LCT(M_2) = 20, \quad LCT(M_3) = 29.$$

5 Generation of k-FT Optimal Schedules

In this section, we address the problem of generating a $k-$FT schedule with the maximum reward. Corollary 1 in Section 4 revealed the necessary and sufficient conditions for k-fault tolerance, which involves only pre-computation of $LCT(M_i)$ bounds and checking $End(M_i)$ values against these. Without loss of generality, and in accordance with [2, 3], we consider FT-Optimal schedules with no idle time[1].

To obtain a $k-$FT optimal schedule S^* for a given task set, we will start creating an initial (template) schedule S_t which contains *only* mandatory parts. Further, in S_t, each mandatory part M_i will be scheduled exactly in time interval $[LCT(M_i) - m_i, LCT(M_i)]$. Figure 6 shows the template schedule S_t for the example task set of Section 3.

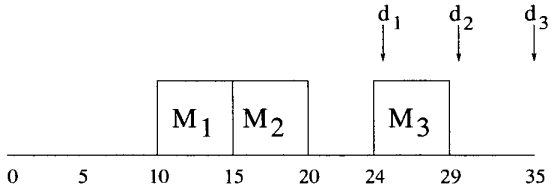

Figure 6. The template schedule S_t

[1] If there is an idle time unit in the middle of an optional execution, one can execute that optional task for one more time unit. Idle time which occur during mandatory parts can be removed, effectively decreasing the completion times and not hurting k-fault tolerance. Clearly, the total reward never decreases after such schedule modifications.

Note that mandatory tasks do not overlap in S_t since:

$$LCT(M_i) =$$
$$lct(M_i, 0, k) \leq min\{d_i, lct(M_{i+1}, 0, k) - m_{i+1}\}$$
$$\leq lct(M_{i+1}, 0, k) - m_{i+1} = LCT(M_{i+1}) - m_{i+1}$$

Thus, S_t will contain mandatory parts scheduled at their latest completion times, with (possibly) gaps among them. However, $Rew_{S_t} = 0$ since we did not schedule any optional part yet. Observe that, in any $k-$FT schedule, and incidentally in the $k-$FT Optimal schedule denoted by S^*, **any mandatory part can be moved earlier from its original allocation in S_t to create room for optional parts if needed, but never later**, since this would result in a non-FT schedule (Corollary 1). For example, Figure 7 illustrates three fully utilized schedules corresponding to the template schedule S_t. They all satisfy the $k-$fault tolerance requirement: yet, the total rewards are different, and any of them could be the k-FT optimal schedule, depending on the specific reward functions. To summarize, S_t represents the boundary conditions that any k-FT schedule must satisfy.

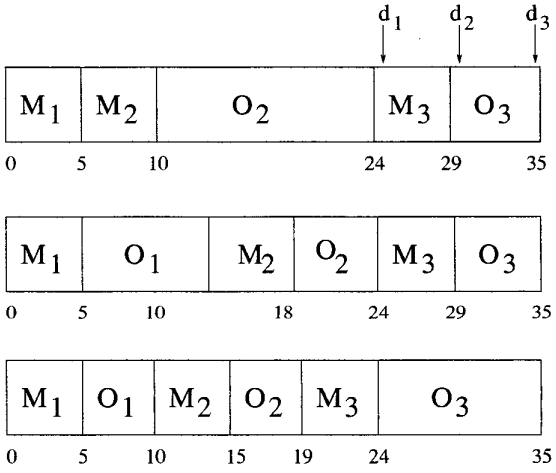

Figure 7. Three k-FT schedules

To maximize the total reward, we schedule optional parts into the 'gaps'. Yet, there are several necessary constraints that we should obey. Namely, no optional part O_i can be scheduled beyond deadline d_i. Similarly, precedence constraints and fault tolerance requirements enforce the following:

Proposition 5 *In the schedule segment $[LCT(M_i), d_i]$ of any k-FT schedule S, no optional part O_j, $j < i$, can be scheduled.*

Proof: Suppose there is an optional part O_j scheduled after $LCT(M_i)$ $(i > j)$ in S. Since M_i should be scheduled later than O_j, $End_S(M_i) > LCT(M_i)$, which suggests that S is not $k-$FT, leading to a contradiction. □

Hence, we obtain the following **necessary** conditions for k-fault tolerance:

N1: In interval $[d_i, d_{i+1}]$ the only optional parts that can be scheduled are $O_{i+1}, \ldots, O_n$.

N2: In interval $[LCT(M_i), d_n]$ the only optional parts that can be scheduled are $O_i, \ldots, O_n$.

We can reflect the constraints N1 and N2 in the optional assignments as follows. Let us start by noting that $Start_{S_t}(M_i) = End_{S_t}(M_i) - m_i = LCT(M_i) - m_i$ is the latest start time of M_i in any k−FT schedule. Clearly, the total CPU allotment for the first task, namely $m_1 + t_1$, cannot exceed d_1 (constraint N1), or $Start_{S_t}(M_2)$ (constraint N2): $m_1 + t_1 \leq min\{d_1, Start_{S_t}(M_2)\}$. Similarly, the total assignment $m_1 + t_1 + m_2 + t_2$ should satisfy the deadline constraint for T_2 and LCT constraint for M_3: $m_1 + t_1 + m_2 + t_2 \leq min\{d_2, Start_{S_t}(M_3)\}$. And in general:

$$\sum_{i=1}^{j}(m_i + t_i) \leq min\{d_j, Start_{S_t}(M_{j+1})\} \quad j < n$$

Finally, the schedule will contain no idle time:

$$\sum_{i=1}^{n}(m_i + t_i) = d_n$$

We continue with the following definitions:

$$\tilde{d}_i = \begin{cases} min\{d_i, Start_{S_t}(M_{i+1})\} & \text{if} \quad i < n \\ d_n & \text{if} \quad i = n \end{cases}$$

$$C_i : \text{the constraint} \begin{cases} \sum_{j=1}^{i}(m_j + t_j) \leq \tilde{d}_i & \text{if} \quad i < n \\ \sum_{j=1}^{i}(m_j + t_j) = \tilde{d}_i & \text{if} \quad i = n \end{cases}$$

Note that $\tilde{d}_i$ represents the *effective* deadline of T_i, that is, the execution of T_i beyond this limit violates either its deadline or k-fault tolerance (for convenience, we define $\tilde{d}_0 = 0$).

The theorem below proves that the constraint set $\{C_i\}$ derived from conditions **N1** and **N2** is **necessary and sufficient** for k-fault tolerance:

Theorem 1 *A schedule S is k-FT if and only if it satisfies the constraints $C_1, \ldots, C_n$.*

Proof: Suppose that each constraint $C_i \quad i = 1, \ldots, n$ is satisfied. The schedule is feasible since $\sum_{j=1}^{i}(m_j + t_j) \leq \tilde{d}_i \leq d_i \quad 1 \leq i \leq n$. But also, $\sum_{j=1}^{i-1}(m_j + t_j) \leq \tilde{d}_{i-1} \leq Start_{S_t}(M_i) \quad 1 \leq i \leq n$. Hence, S is also k-FT since:

$$\sum_{j=1}^{i-1}(m_j + t_j) + m_i = End_S(M_i) \quad \leq$$

$$Start_{S_t}(M_i) + m_i = End_{S_t}(M_i) = LCT(M_i) \quad i = 1, \ldots, n$$

Conversely, suppose that the schedule S is k-FT. This implies: $End_S(M_{i+1}) = \sum_{j=1}^{i}(m_j + t_j) + m_{i+1} \leq LCT(M_{i+1}) = End_{S_t}(M_{i+1}) \quad i = 0, \ldots, n-1$. By subtracting m_{i+1} from both sides (note that $End_{S_t}(M_{i+1}) - m_{i+1} = Start_{S_t}(M_{i+1})$), we obtain:

$$\sum_{j=1}^{i}(m_j + t_j) \leq Start_{S_t}(M_{i+1}) \quad 0 \leq i \leq n-1$$

But the schedule is also feasible and fully utilized, hence:

$$\sum_{j=1}^{i}(m_j + t_j) \leq d_i \quad i = 1, \ldots, n-1 \quad \text{and} \quad \sum_{j=1}^{n}(m_j + t_j) = d_n$$

Combining the last results, we obtain: $\sum_{j=1}^{i}(m_j + t_j) \leq min\{d_i, Start_{S_t}(M_{i+1})\} \quad i = 1, \ldots, n-1$ and $\sum_{j=1}^{n}(m_j + t_j) = d_n$, yielding:

$$\sum_{j=1}^{i}(m_j + t_j) \leq \tilde{d}_i \quad i = 1, \ldots, n-1$$

$$\sum_{j=1}^{n}(m_j + t_j) = \tilde{d}_n$$

Hence, all the C_i constraints are satisfied. □

Having expressed the FT conditions in terms of optional service times $\{t_i\}$, we can now formulate the non-linear optimization problem to maximize the total reward. Note that every t_i assignment should be non-negative, in order to have a physical interpretation. Hence, we obtain our final theorem:

Theorem 2 *k-FT optimal optional service assignments $\{t_i\}$ are given by the following optimization problem:*

$$\text{maximize} \quad \sum_{i=1}^{n} R_i(t_i) \tag{5}$$

$$\text{subject to} \quad \sum_{i=1}^{n} t_i = d_n - \sum_{i=1}^{n} m_i \tag{6}$$

$$0 \leq t_i \quad i = 1, \ldots, n \tag{7}$$

$$\tilde{d}_n - \tilde{d}_{i-1} - \sum_{j=i}^{n} m_j \leq \sum_{j=i}^{n} t_j \quad i = 2, \ldots, n \tag{8}$$

Proof: The constraint (6) corresponds to C_n. The constraint (7) is self-explanatory. The constraint set (8) is equivalent to $\{C_i\}$, as proven below. Let us define:

$$\overline{C_i} : \text{the constraint} \begin{cases} \sum_{j=n-i+1}^{n}(m_j + t_j) \geq \tilde{d}_n - \tilde{d}_{n-i} & \text{if } i < n \\ \sum_{j=1}^{n}(m_j + t_j) = \tilde{d}_i - \tilde{d}_0 & \text{if } i = n \end{cases}$$

We claim that the constraint set $\{C_i\}$ is satisfied for a given schedule S if and only if the constraint set $\{\overline{C_i}\}$ is satisfied.

In fact, suppose that the set $\{C_i\}$ is satisfied. Then by subtracting C_{n-i} from C_n, one can obtain $\{\overline{C_i}\}$ (for $i = 1, \ldots, n-1$). C_n is identical to $\overline{C_n}$ ($d_n = \tilde{d}_n$ and $\tilde{d}_0 = 0$).

Conversely, suppose that the set $\{\overline{C_i}\}$ is satisfied. One can subtract $\overline{C_{n-i}}$ from $\overline{C_n}$ to get C_i (for $i = 1, \ldots, n-1$), and so on. By re-arranging the new constraint set $\{\overline{C_i}\}$, we obtain the formulation used in the optimization problem above. □

The above problem is a non-linear (concave) optimization problem with equality and inequality constraints. It can be solved by using the algorithm developed for the solution of problem CHAIN, in [2]. The complexity of the solution is $O(n^2 \log n)$ for linear, logarithmic or identical concave reward functions; in the general concave case, the complexity is $O(n^3 \log n)$.

To summarize, the algorithm in [2], which assigns FT-Optimal t_i values, proceeds in two phases. In the first phase, we focus solely on satisfying the inequality constraints by processing the task set in a *bottom-up* manner. During this phase, we apply a **least commitment strategy** in that we do not assign to an optional task a service time which is greater than required by inequality constraints. During the second phase, we make optimal distribution of the total schedule segments available for all optional parts in the chain, *considering the output of the first phase as lower bounds on the execution times.*

To illustrate the derivation of exact constraint set for the CHAIN, we return to the example task set of Section 3, of which the template schedule S_t was shown in Figure 6. The 'effective' deadline for each task is the minimum of its deadline and the start time of the next task in S_t, thus:

$$\tilde{d}_1 = 15,\ \tilde{d}_2 = 24,\ \tilde{d}_3 = 35$$

Similarly, the inequality constraints are obtained as:

$$\begin{aligned} m_3 + t_3 &\geq \tilde{d}_3 - \tilde{d}_2 = 11 \\ m_2 + t_2 + m_3 + t_3 &\geq \tilde{d}_3 - \tilde{d}_1 = 20 \end{aligned}$$

By substituting the values of m_i $i = 1, 2, 3$, we obtain the instance of CHAIN:

$$\begin{aligned} \text{maximize} \sum_{i=1}^{n} & \ R_i(t_i) \\ \text{subject to} \sum_{i=1}^{n} t_i & = 20 \\ 0 & \leq t_i \quad i = 1, 2, 3 \\ 6 & \leq t_3 \\ 10 & \leq t_2 + t_3 \end{aligned}$$

The algorithm which solves the problem CHAIN [2] returns the output set $\{t_1 = 10, t_2 = 4, t_3 = 6\}$, which, indeed corresponds to the k-FT Optimal schedule which was previously shown in Figure 5.

6 Conclusion

In this paper, we addressed the problem of generating FT-Optimal schedules for imprecise computation tasks with multiple timing constraints, recovery blocks and linear precedence constraints. The approach can be also used for a set of independent tasks, in case that preemption is not allowed. After pointing out to the disadvantages of adopting trivial extensions of previous solutions, we first provided an exact characterization of Imprecise Computation schedules which allow timely completion of mandatory parts even in the presence of k transient faults. The test is efficient and further, can be applied to *any* IC schedule, with arbitrary or no precedence constraints, with preemption or no preemption, as long as the recovery blocks are to be executed immediately following error detection(s). This special type of recovery technique was called 'Immediate Recovery' in [3].

In the last part of the paper, we showed how to use the task and fault characterization information to generate the schedules which allow timely recoveries of mandatory parts while compromising the total reward as little as possible. Our future work in this area includes the investigation of efficient heuristics for intractable cases.

References

[1] H. Aydın, R. Melhem, D. Mossé and P.M. Alvarez. Optimal Reward-Based Scheduling of Periodic Real-Time Tasks. In *Proceedings of 20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, December 1999.

[2] H. Aydın, R. Melhem and D. Mossé. Incorporating Error Recovery into the Imprecise Computation Model. *The Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, December 1999.

[3] H. Aydın, R. Melhem and D. Mossé. Tolerating Faults while Maximizing Reward. *Proceedings of the Twelfth Euromicro Conference on Real-Time Systems (Euromicro'00)*, Stockholm, June 2000.

[4] R. Bettati, N.S. Bowen and J.Y. Chung. Checkpointing Imprecise Computation. *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, Dec. 1992.

[5] R. Bettati, N.S. Bowen and J.Y. Chung. On-Line Scheduling for Checkpointing Imprecise Computation. *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems*, June 1993.

[6] J. K. Dey, J. Kurose and D. Towsley. On-Line Scheduling Policies for a class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks. *IEEE Transactions on Computers* 45(7):802-813, July 1996.

[7] J. Grass and S. Zilberstein. Value-Driven Information Gathering. *AAAI Workshop on Building Resource-Bounded Reasoning Systems*, Rhode Island, 1997.

[8] X. Huang and A. M. K. Cheng, Applying Imprecise Computation Algorithms to Real-Time Image and Video Transmission. In *IEEE Real-Time Technology and Applications Symposium*, 1995.

[9] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5): 58-68, May 1991.

[10] R. Rajkumar, C. Lee, J. P. Lehozcky and D. P. Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of 18th IEEE Real-Time Systems Symposium*, December 1997.

[11] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[12] W.-K. Shih, J. W.-S. Liu, and J.-Y. Chung. Algorithms for scheduling imprecise computations to minimize total error. *SIAM Journal on Computing*, 20(3), July 1991.

[13] S. Zilberstein and S.J. Russell. Anytime Sensing, Planning and Action: A practical model for Robot Control. In *IJCAI 13*, France, 1993.