Tolerating Faults While Maximizing Reward *

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

Abstract

The imprecise computation (IC) model is a general scheduling framework, capable of expressing the precision vs. timeliness trade-off involved in many current real-time applications. In that model, each task comprises mandatory and optional parts. While allowing greater scheduling flexibility, mandatory parts in the IC model have still hard deadlines and hence they must *be completed before the task's deadline even in the presence of faults. In this paper, we address fault tolerant (FT) scheduling issues for IC tasks. First, we propose two recovery schemes, namely* Immediate Recovery *and* Delayed Recovery. *These schemes can be readily applied to provide fault tolerance to mandatory parts by scheduling optional parts appropriately for recovery operations. After deriving the necessary and sufficient conditions for both schemes, we consider the* FT-Optimality *problem, that is, generating a schedule which is FT and whose reward is maximum among* all *possible FT schedules. For Immediate Recovery, we present and prove correctness of an efficient FT-Optimal scheduling algorithm. For Delayed Recovery, we show that the FT-Optimality problem is NP-Hard, thus is intractable.*

1 Introduction

In real-time systems, timeliness is as important as the correctness of the output. Traditionally, hard real-time scheduling theory has aimed at achieving predictability by assuming worst-case scenarios, such as worst-case execution time and interarrival rates. However, the advance of new technologies such as multimedia applications and Web-based information servers has introduced extra dimensions to the traditional framework. One major characteristic of the new era is the *progressive* nature of the task's execution: First a result or output of minimal quality is produced, then it is *refined* by additional computation(s). Yet, scheduling decisions must a priori assure an output of minimal quality for *every* task, even in the presence of unpredictable events. In addition, CPU allocation to refinement processes should maximize a performance metric.

The *Imprecise Computation (IC)* technique appears is an appropriate framework to formulate and address many aspects of the issues discussed above. It was originally proposed as an overload scheduling technique for applications with less stringent timing constraints than hard-real time systems [4]. In this model, each task is composed of a *mandatory* part and an *optional part*. The mandatory part runs first and its completion before the deadline assures a result of minimal quality. The optional part becomes ready only after the mandatory part completes, and its execution can be interrupted at any time. The quality of the final result is proportional to the service time received by the optional part. The applicability of the model has steadily expanded to several areas of Computer Science/Engineering, such as multimedia applications, image and speech processing, time-dependent planning, robot control/navigation systems, medical decision making, information gathering, real-time heuristic search and database query processing.

In the IC model, the timely completion of mandatory parts is still crucial, even in the presence of faults. A first study addressing Fault Tolerance (FT) issues in the context of IC framework appeared in [2]. An extension for on-line scheduling was considered in [3]. However, these works assume a priori the knowledge of the worst-case fault profile *per task*. Besides the considerable difficulty of obtaining this information for real applications (as opposed to the fault profile for the *system*), the solutions enforce provisioning for simultaneous occurrences of *all faults of all tasks*. This fact (as observed in [2]) drastically reduces the number of task sets which can pass FT-schedulability set, although in practice only one or a few tasks incur faults. Further, the scheme requires the update of the schedule when a task completes succesfully (without fault), which largely increases the run-time overhead.

Our work is based on the observation that the optional parts in IC schedules have the *potential* of providing the time redundancy for recovery of mandatory parts. Hence, one of the main issues in this study is the investigation of FT conditions and recovery techniques involved in IC environments. We fur-

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

ther enforce *FT-Optimality*: that is, to compute the schedule which provides the *largest reward (utility) among all possible FT schedules*. Finally, we require that the on-line updating of the schedule be avoided and the FT-Optimality of the schedule be *preserved*, as long as faults are not encountered as opposed to [2, 3]. This is crucial in preventing excessive run-time overhead, since the successful completion of tasks is much more common than the occurrence of faults in any realistic system.

In a recent work, we addressed the problem in the context of tasks with or without precedence constraints, but sharing a single, end-to-end deadline [1]. In practice, however, numerous applications mandate the use of different deadlines and our main purpose in this work is to extend the FT-Optimality approach to tasks with multiple deadlines. Moreover, unlike the single-deadline case, the new setting requires distinguishing among different recovery schemes, which lead to different FT approaches/schedules and as we show, even the tractability of the problem is affected by this choice.

2 System Model

2.1 Task Model

We consider a uniprocessor system with a task set $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$. All tasks are assumed to be independent. Further, each task T_i is composed of a mandatory part M_i and an optional part O_i. The worst-case execution times of M_i and O_i are denoted by m_i and o_i, respectively. M_i becomes ready at $t = r_i$ *(ready time)* and it must be completed by $t = d_i$ *(deadline)*. On the other hand, O_i can start to execute only when M_i completes, and further refines the latter's output while still before the deadline. Preemption is allowed and we ignore the costs of preempting the tasks throughout the paper. We say that a schedule is *feasible* if every mandatory part M_i receives at least m_i units of service time after its ready time r_i, but before its deadline d_i.

The quality of the final result produced by T_i is directly proportional to the amount of service time t_i assigned to the *optional part* O_i: it is still acceptable, yet minimal when $t_i = 0$, and maximal when $t_i = o_i$. Hence, the *reward* of task T_i is $R_i = t_i$ when $0 \leq t_i \leq o_i$, and $R_i = o_i$ when $t_i > o_i$ (that is, the execution beyond o_i does not improve further the quality). Our concept of reward is analogous to (dual of) the concept of *(precision) error* (the amount of optional part left unexecuted) in other IC studies [4]. A schedule S is *optimal* if it is feasible and maximizes the total reward $REW = \sum_{i=1}^{n} R_i$.

2.2 Fault Model

We assume that at most k faults may occur during the execution of the task set. However, we develop and present our methodology first in the context of a *single* fault model (that is, $k = 1$) for the sake of simplicity. An extension of this framework to the k-faults case is presented at the end of Section 4. We assume that whenever a mandatory part M_i completes, consistency or acceptance checks are performed on its output. If the outcome is positive, then the output of M_i is committed, and the optional part O_i becomes ready. However, in case that the checks reveal an error, a recovery mechanism is invoked, either to re-execute the mandatory part, or to execute a recovery block. The recovery block associated with M_i and its worst-case execution time are denoted by B_i and b_i, respectively. If an error is detected at the end of the optional part, then its result is not committed.

In general, a feasible schedule is said to be *(single) Fault Tolerant (FT)* if it allows the recovery of a single fault detected in any mandatory part M_i to be completed before the deadline d_i, *while not compromising the timely completion of other mandatory parts*. Finally, a schedule is *FT-Optimal* if
i. it is FT and,
ii. its total reward is maximum among all possible FT schedules. We require that the FT-Optimality of the schedule be preserved as long as no faults are encountered.

3 Recovery Models for Imprecise Computation

We should underline that the definition of FT(-Optimal) schedule above is incomplete, in the sense that one needs to specify *how and when the recovery block is executed*. Since an additional workload is introduced to the system because of the recovery block, *all the subsequent optional parts may not have a chance to execute*. We can distinguish two recovery schemes that can be readily applied in practical real-time applications without considerable overhead:

- **Immediate Recovery (IR):** The recovery block is executed immediately when the error is detected at the end of the faulty mandatory task. The mandatory parts yet to execute may have to be delayed/postponed due to the recovery operation.

- **Delayed Recovery (DR):** The recovery block is allowed to execute *only* in time slots which are not dedicated to any mandatory task (i.e., those which are empty or assigned to an optional part), but still before the deadline. Thus, mandatory parts are not shifted in the timeline.

As an example, consider the schedule in Fig. 1a. We assume that $b_1 = m_1$. If a fault is detected at the end of M_1 at $t = 4$, the recovery operation is initiated. If the mechanism chosen is the immediate recovery then the recovery proceeds without preemption and until full completion, resulting in a 'shift' of M_2 (Fig. 1.b). On the other hand, if the delayed recovery is applied then B_1 runs only during the optional slots while before d_1. Clearly, M_2 does not need to be shifted in this case (see Fig. 1.c). Note that, in both cases M_2 and the recovery block B_1 complete successfully before their deadlines.

It should be emphasized that the two schemes above do *not* lead to identical FT schedule spaces. To see this, consider the schedule in Figure 2.a. Assuming that $m_1 = m_2 = b_1 = b_2 = 4$, the schedule above is clearly FT according to the immediate recovery scheme, but not according to the delayed recovery:

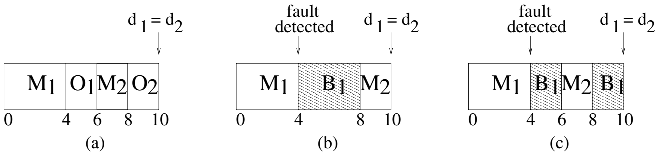

Figure 1. (a) A feasible schedule (b) Recovery of M_1 with IR (c)Recovery of M_1 with DR

There is simply no optional part scheduled before d_1, during which one would run the recovery block B_1. Similarly, the schedule shown in Figure 2.b is FT according to the delayed recovery scheme, while the immediate recovery of a fault in M_3 would cause M_2 to miss its deadline (assume again that $b_i = m_i$ for all the tasks and recovery by re-execution).

Figure 2. (a) An FT schedule according to IR but not DR (b) An FT schedule according to DR but not IR

Given a schedule and a recovery mechanism, checking its tolerance to fault(s) is not a difficult process and can be accomplished in polynomial-time. On the other hand, as indicated in Section 1, our aim is to construct FT-Optimal schedules in a computationally efficient manner, which turns out to be a non-trivial task. Given a feasible schedule S, we define:

$End_S(M_i)$: The time by which M_i completes in S.

$Opt_S(t_1, t_2)$:The sum of optional parts and empty slots in time interval $[t_1, t_2]$ in S.

We will drop the subscript S, whenever the schedule involved is clear from the context. As an example, in the schedule of Figure 2.b, $End(M_2) = 26$, while $Opt(16, 32) = 10$. Clearly, in case of preemptions, the End function should consider the completion of the last portion of the task.

Observe that in any of the schemes, we can not impose an *a priori* order among the different tasks, as long as we are not ready to compromise FT-optimality.

4 Immediate Recovery(IR) with identical ready times

We start by defining necessary and sufficient FT conditions according to the IR scheme. The execution of the recovery block B_i may affect *only* the mandatory parts which are scheduled *after* M_i. We say that a task T_j is *safe* in a given schedule S, if and only if M_j can be completed in a timely manner even in the presence of any faults in tasks preceding or occurring during M_j; or equivalently, if and only if:

$$End(M_j) + b_i - Opt(End(M_i), End(M_j)) \leq d_j \quad \forall\ i \ \text{ such that } \ End(M_i) \leq End(M_j) \tag{1}$$

For a given i, the left-hand side of the inequality (1) indicates the time by which M_j *would* complete, if a fault *were* detected in M_i; or equivalently the (new) $End(M_j)$ value after the fault and recovery of M_i. More precisely, it is derived by taking into account that the workload is increased by an amount of b_i, but decreased by (if necessary) removing all the subsequent optional parts, in case an error is detected at M_i. Now, a schedule S is said to be *Fault Tolerant* (according to IR scheme), if and only if all the tasks are safe; that is, if and only if the Equation (1) holds for all mandatory tasks $M_j,\ \ j = 1, \ldots, n$.

Given an FT schedule S, consider the effects of moving a portion τ_x of a task T_x to a later in point in the schedule as shown in Figure 3. We call such a move *the forward (FW) move* of τ_x —which may be mandatory or optional. Throughout this section, α denotes the schedule portion preceding τ_x prior to the FW-move and γ denotes the schedule portion following τ_x after the FW-move. Observe that both α and γ occupy the same portion of the shedule before and after the move. The portion initially located between τ_x and γ is denoted by β.

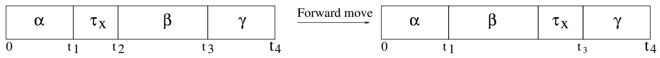

Figure 3. The forward move of the task portion τ_x

Theorem 1 : (Forward-moving theorem) *If a schedule S is FT, then the schedule S' resulting from the forward-move of a task portion τ_x belonging to T_x is still FT if and only if, in S':*
i. T_x is safe, and
ii. A fault of T_x does not cause a deadline miss in any mandatory part located in γ.

Proof: We will prove the theorem by showing that in the transformed schedule S',
(i.) Every task T_j such that $End_S(M_j)$ is in α are still safe,
(ii.) Every task T_j such that $End_S(M_j)$ is in β are still safe,
(iii.) No fault detected in α or β can cause a deadline miss (of mandatory parts) in γ.

To establish (i.), it is sufficient to observe that the segment α remains unchanged in S', hence any task T_j such that $End(M_j) \leq t_1$ is still safe in the transformed schedule.

To see that (ii.) holds, consider any task T_j such that $t_2 < End_S(M_j) \leq t_3$ (M_j in β). Since T_j was safe in S by definition, the equation (1) gives:

$$End(M_j) + b_i - Opt(End(M_i), End(M_j)) \leq d_j \quad \forall\ i \ \text{ such that } \ End(M_i) \leq End(M_j) \tag{2}$$

Consider the fault of a task M_i preceding M_j in S ($End_S(M_i) \leq End_S(M_j)$). If $End_S(M_i) = t_2$, then the FW-moved portion τ_x belongs to M_x, $M_i = M_x$ and the fault of M_x can no longer cause a problem for M_j in S' where now it succeeds M_j. Otherwise, it is either in α or β (preceding M_j). Note that $End_S(M_j) - End_{S'}(M_j) = t_2 - t_1 > 0$ (M_j has been moved backward in the schedule by an amount of $t_2 - t_1$). Also $Opt(End_S(M_i), End_S(M_j)) - Opt(End'_S(M_i), End'_S(M_j))$ is either $t_2 - t_1$ (in case that M_i is in α and τ_x is a portion of the optional part O_x) or 0 (the remaining cases). Hence, the sum of optional parts between M_i and M_j may have decreased, but *at most* by an amount of $t_2 - t_1$. However, this shows that M_j is still safe in S' since: $End_{S'}(M_j) + b_i - Opt(End_{S'}(M_i), End_{S'}(M_j)) \leq End_S(M_j) + b_i - Opt(End_S(M_i), End_S(M_j)) \leq d_j \quad \forall\, i \text{ such that } \quad End_{S'}(M_i) \leq End_{S'}(M_j)$

Finally, to show (iii.), let $End_S(M_j)$ be in γ. Clearly, $End_S(M_j) = End_{S'}(M_j)$. Consider a fault of any task M_i preceding M_j in S', such that $End_S(M_i)$ in α or β. It is not difficult to see that $Opt(End_S(M_i), End_S(M_j)) \leq Opt(End_{S'}(M_i), End_{S'}(M_j))$, thus the sum of optional parts between M_i and M_j may have never decreased in S'. But this gives: $End_{S'}(M_j) + b_i - Opt(End_{S'}(M_i), End_{S'}(M_j)) \leq End_S(M_j) + b_i - Opt(End_S(M_i), End_S(M_j)) \leq d_j \quad \forall\, i \quad \text{such that} \quad End_{S'}(M_i)\,\text{is in}\ \alpha$ or β, completing the proof. □

With the help of the Forward-moving theorem, we can further obtain important properties of FT schedules, as the next theorem demonstrates.

Theorem 2 *It is always possible to transform an FT schedule S to another FT schedule with the same total reward, where:*
i. Mandatory parts are not preempted,
ii. Optional parts are not preempted,
iii. Optional parts follow EDF order.

Proof: Suppose that M_i is preempted once or more in S. Let M_{ia} be the first 'segment' of M_i in S, M_{ib} the second and so on. Consider forward-moving of M_{ia} close to M_{ib} as illustrated in Fig 4. We call the resulting schedule S'. According to Theorem 1, we need to check only the safety of M_i and the effect of M_i's fault in γ. It is easy to see that $Opt(End_S(M_j), End_S(M_k)) = Opt(End_{S'}(M_j), End_{S'}(M_k)) \quad \forall j, k$ and further, there is no task M_j such that $End(M_j)$ has increased in S'. Hence the condition (1) implies that M_i and all tasks in γ are still safe. One can apply this transformation repeatedly to obtain a schedule where the mandatory parts are never preempted.

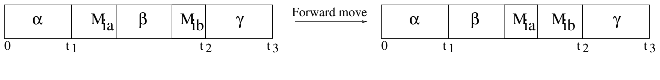

Figure 4. Forward move of the preempted mandatory portion

Similarly, if O_i is preempted, one can move it forward to obtain the schedule S' (see Fig 5). Clearly, $0 \leq End_S(M_i) = End_{S'}(M_i) \leq t_1$ and M_i is in α, T_i definitely is still safe in S' (where α did not change at all). Also, $Opt(End_S(M_i), End_S(M_j)) \leq Opt(End_{S'}(M_i), End_{S'}(M_j)) \quad \forall\, M_j$ in γ and the condition (1) ensures that all tasks in γ are safe, thus S' is still FT. Hence, it is possible to apply the transformation repeatedly to remove all the preemptions of optional parts.

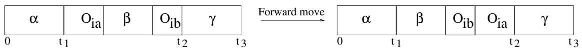

Figure 5. Forward move of the preempted optional portion

Finally, consider a pair of optional parts which do not follow EDF order in S. We can consider that all the preemptions are already removed via the transformations described above. Again, consider moving O_i after O_j (Fig. 6). Following a reasoning completely analogous to the preceding paragraph, we can establish that M_i and all the tasks in γ are still safe. A finite number of transformations would definitely yield a schedule where optional parts follow EDF order. Note also that, the total reward of the FT schedule remains the same after any forward-moves just described. □

Figure 6. Forward move of the optional task with later deadline

Corollary 1 *Provided that a task set* **T** *is FT, there exists an FT-optimal schedule for* **T***, where no mandatory or optional part is preempted and optional parts follow EDF order.*

Unfortunately, it may not be always possible to find an FT-optimal schedule where *mandatory* parts follow also EDF order. To see this, consider the tasks T_1, T_2, T_3 with parameters $m_1 = 4, o_1 = 8, d_1 = 20; m_2 = 2, o_2 = 2, d_2 = 24; m_3 = 8, o_3 = 2, d_3 = 26$. We assume that $b_i = m_i \quad i = 1, 2, 3$. In fact, one can schedule the task set in an FT manner without discarding any optional part as shown in Figure 7.a, which is consequently an FT-Optimal schedule. On the other hand, if one tries to enforce the EDF rule for mandatory parts also, the best schedule that can be obtained is the one shown in Fig. 7.b. Observe that O_1 can not be scheduled beyond d_1 and the large recovery block of M_3 contributes to a lower reward.

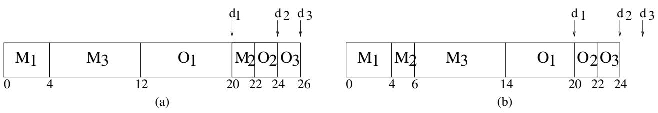

Figure 7. (a) The FT-Optimal schedule (b) The suboptimal schedule with EDF order among mandatory parts

Before presenting our final algorithm, we obtain another useful property of the FT-optimal schedules.

Theorem 3 *If a task set is FT, then there exists always an FT-optimal schedule, where, for each task T_i, $t_i > 0$ only if $t_j = o_j \quad j = i + 1, \ldots, n$.*

Informally, the theorem states that one can always find an FT-Optimal schedule where an optional part with later deadline can always be used instead of an optional part with earlier deadline, as long as the upperbounds on the optional execution times are not violated.

Proof: Without loss of generality, assume that the FT-Optimal schedule is in the 'canonical' form stated in Corollary 1: that is, no preemption exists and optional parts follow EDF order. Let O_p be the optional part with the *latest* deadline such that $t_p > 0$ and $t_{p+1} < o_{p+1}$ (such a task should definitely exist if the proposition does not hold). Since the schedule is in the canonical form, we can move $\Delta = min(o_{p+1} - t_{p+1}, t_p)$ units of O_p forward and consider it as part of O_{p+1}. Figure 8 illustrates the FW-move.

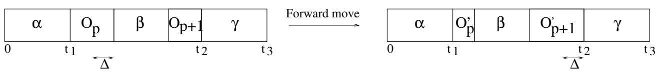

Figure 8. Forward move of an optional part's portion (Δ) and its renaming

Note that from the definition of Δ, $t_{p+1} + t_p = t'_{p+1} + t'_p$, hence the total reward remains the same. Moreover, one can easily verify that the FW-move of Δ units of O_p does not affect the safety of any task. Note that, $t_j = o_j$ for $j = p+2, \ldots, n$ from the assumptions. If the statement of the theorem does not hold after this transformation, then we can apply the technique repeatedly to favor later optional parts without harming the fault tolerance requirement or the optimal reward. □

Theorem 3 implies that one does not need to schedule an optional part unless *all* the optional parts with the later deadlines are *fully* utilized. Hence, given a set **O** of optional parts $\{O_1, \ldots, O_n\}$ and a positive real number X we can define the function $SUFFIX(O, X)$ which returns the (new) set of optional parts **O'**$= \{0, 0, \ldots, O'_m, O'_{m+1} \ldots, O'_n\}$ such that $o'_i = o_i$ for $i = m+1, \ldots, n$ and $o'_m = X - \sum_{i=m+1}^{n} o'_i$. Note that in case that $X \leq o_n$, **O'**$= \{0, 0, \ldots, 0, O'_n\}$ where $o'_n = X$; or if $X \geq \sum o_i$, then **O'**= **O**.

Corollary 2 *The existence of an FT schedule with total reward larger than or equal to X, implies that there is an FT-Optimal schedule where the optional parts in SUFFIX(O,X) appear fully and in EDF order.*

It may happen that in an FT-optimal schedule $\sum m_i + \sum t_i < d_n$; that is, there are 'empty' slots in the interval $[0, d_n]$. The following establishes that it is possible further to obtain a fully-utilized schedule or timeline without hurting FT, by augmenting t_n.

Theorem 4 *If* $\sum m_i + \sum t_i < d_n$ *in an FT-optimal schedule S, then we can increase* t_n *(beyond the upperbound* o_n*) to obtain an FT-optimal schedule S' where* $\sum m_i + \sum t_i = d_n$

Proof: Since $\sum m_i + \sum t_i < d_n$ in S, there should be empty slots within the timeline interval $[0, d_n]$. By a reasoning similar to the proof of Theorem 3, moving every empty slot to the end of the schedule and renaming it as part of the optional part O_n (beyond the upperbound o_n) keeps the total reward unchanged, the FT property is preserved, and the timeline is now fully utilized. □.

The results above allow us to devise a polynomial-time algorithm (Figure 9) to **decide** whether there exists an FT-Optimal schedule with a total reward of **exactly** X.

```
Algorithm DECIDE(M, O, X)
Set O'= SUFFIX(O, X)
Set diff = d_n - Σ_{i=1}^n m_i - Σ_{i=1}^n o'_i
If diff < 0 then return FAILURE
          else o'_n = o'_n + diff
Set pointer = shiftlimit = d_n
Set index = n
Set M_q = M
Repeat
  While pointer <= d_index do {
      /* schedule optional part there */
      Set pointer = pointer - o'_index
      Set Timeline[pointer, pointer + o'_index] = O_index
      Set index = index - 1
                                                }
  /* Find the FT-forward schedulable mandatory part with */
  /* the latest deadline and schedule it /*
  Set M_a = FT-forward-sch(pointer, shiftlimit)
  If M_a = nil then return FAILURE
  Set pointer = pointer - m_a
  Set Timeline[pointer, pointer + m_a] = M_a
  Set M_q = M_q - M_a
  If d_a >= shiftlimit then shiftlimit = shiftlimit - m_a
             else shiftlimit = d_a - m_a
Until M_q = ∅
Return(Timeline, SUCCESS)
```

Figure 9. Algorithm to solve FT-Optimality Decision Problem

```
Function FT-forward-sch(pointer, shiftlimit)
Repeat
  Set counter = n
  If M_counter in M_q {
    Set endpoint = pointer + b_counter
    if endpoint <= min(shiftlimit, d_counter) return M_counter
  }
  Set counter = counter - 1
Until (counter = 0)
Return nil
```

Figure 10. Function returning FT-Forward schedulable mandatory task

The algorithm is based on the fact that there exists a schedule S that *fully* utilizes the timeline between 0 and d_n, following Corollary 2 and Theorem 4 (in other words, S is an FT-schedule with total reward X, where t_n may be augmented beyond its upperbound). The algorithm:

- Proceeds backwards from d_n and tries to schedule one (mandatory or optional) task at a time, at the given point of the schedule (represented by the variable $pointer$).

- If unable to find such a task, it exits with a FAILURE message.
- If more than one task is eligible for a given time point, it always favors an optional part (in case of ties, the optional task with the latest deadline is chosen). The optional parts are only subject to the deadline constraints.
- If no optional part is eligible, then the mandatory part which is *FT-forward schedulable* at *pointer* is selected. We say that a mandatory part M_i is FT-forward schedulable at t_x, if scheduling M_i between $t_x - m_i$ and t_x does not cause M_i to miss its deadline (i.e., $t_x + b_i \leq d_i$) nor does it cause subsequent tasks to miss their deadlines, in case a fault occurs during the execution of M_i. In order to check how much the subsequent tasks can be pushed later in the timeline, the algorithm keeps a variable named $shiftlimit$. As with optional parts, ties are broken in favor of mandatory task with the later deadline.

Correctness: To begin with, the algoritm's commitment to non-preemptive tasks and to optional parts in SUFFIX(O,X) is immediately justified by Corollaries 1 and 2. Yet, we need to justify the selections of the algorithm whenever two (or more) tasks are eligible at a given point.

Favoring optional parts does not compromise the correctness: Let O_x be the optional part (with the later deadline) such that $pointer \leq d_x$. Scheduling O_p at the $pointer$ clearly does not hurt FT-schedulability of the successive tasks. However, suppose that the algorithm's selection is incorrect, in the sense that it fails to find an FT schedule in the following steps, yet an FT-schedule S with the total reward X would have been obtained if a mandatory task M_i had been selected at this point. The hypothetical schedule S is clearly fully utilized and O_x precedes M_i in S (see Fig. 11). Consider forward-move of O_x after M_i to obtain the schedule S'. It should be straightforward to observe that the two conditions of Theorem 1 still hold in S': The schedule segment up to and including M_x (in α) not being changed, M_x and all the tasks in γ are still safe. Hence the algorithm's selection of O_x in the first place would have also yielded an FT-schedule, and we reach a contradiction.

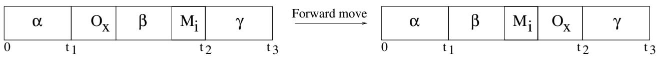

Figure 11. Forward move of the optional task

Favoring mandatory part (with the later deadline) does not compromise the correctness: Note that the algorithm schedules a mandatory part only when it is FT-forward schedulable; this is clearly a necessary condition in an FT-schedule. As to justify the selection of the mandatory task with the later deadline M_x: suppose again that the algorithm fails after choosing M_x in consecutive steps, while the scheduling of another FT-forward schedulable task M_p with an earlier deadline would have yielded an FT-schedule S.

Again consider forward-move of M_x after M_p to obtain the schedule S' (See Fig. 12). The second condition of Theorem 1 is immediately satisfied since M_x is FT-forward schedulable. M_x can not suffer from its own fault because of the same reason.

As the last possibility, suppose M_x misses its deadline in S', due to the fault of a task M_f preceding it. However, S was FT and M_p was safe in S, i.e. , $End_S(M_p) + b_f - Opt(End_S(M_f), End_S(M_p)) \leq d_p$. Clearly, $End_S(M_p) = End_{S'}(M_x)$ and $Opt(End_S(M_f), End_S(M_p)) = Opt(End_{S'}(M_f), End_{S'}(M_x))$, giving $End_{S'}(M_x) + b_f - Opt(End_{S'}(M_f), End_{S'}(M_x)) \leq d_p \leq d_x$. But this shows that M_x is also safe in S' and consequently S' is also FT, yielding a contradiction.

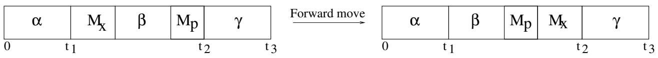

Figure 12. Forward move of the mandatory task with later deadline

Having an efficient decision procedure such as DECIDE above, we can easily develop an algorithm to compute the FT-optimal schedule, by adopting a binary-search like technique on the *maximum reward*. The lower and upper bounds for the binary search are initially set to 0 and $d_n - \sum_{i=1}^{n} m_i$, respectively (the latter being the maximum possible reward of the task set, FT or non-FT). The algorithm initially checks the existence of FT schedules with exactly these amount of rewards, and if needed, the 'middle' point as chosen as the average of these values. Lower and upper bounds are adjusted accordingly throughout the iterations.

Complexity: The time complexity of the algorithm DECIDE is clearly $O(n^2)$, since we schedule $2n$ tasks (n optional and n mandatory) and for each of the tasks we execute at most $O(n)$ operations (For mandatory tasks, the function FT-forward-sch and lines 16-20, for optional tasks lines 10-12). The complexity of the final algorithm is then $O(n^2 \log d_n)$.

Extension to k faults:

Now, we consider the problem of extending the FT-Optimality framework to the case of k faults, with IR scheme and tasks having identical ready times. We assume that all the recovery blocks for a given task have the same worst-case execution time, and that all k faults *may* also occur during the execution and recoveries of a single task. The following serves as the basis for the approach.

Theorem 5 *A schedule* S *is tolerant to* k *faults if and only if it is tolerant to a single fault for the 'transformed' task set where* $b'_i = k \cdot b_i,\ i = 1 \ldots n$.

The result underlines that the same FT-Optimality analysis and algorithm presented above can be applied to the k-fault case, just by scaling up the b_i values accordingly.

5 Delayed Recovery (DR) scheme

Recall that in the Delayed Recovery (DR) case, the optional parts scheduled before d_i are used as recovery slots to achieve Fault Tolerance. Given a schedule S, we define the function $Slack(M_i)$ to be equal to the sum of optional parts scheduled after M_i but before d_i. We say that a schedule S is fault tolerant according to the DR scheme if $Slack(M_i) = Opt(End(M_i), d_i) \geq b_i \ \forall i$.

Unfortunately, the FT-Optimality problem within the DR scheme turns out to be NP-Hard *even for identical ready times and the single fault case*.

To prove this, we will define a decision problem version of FT-Optimality, denoted by FT-DR-OPT, and show that it is NP-Hard, which implies the intractability of the FT-Optimality problem. To prove NP-hardness of FT-DR-OPT, we will transform the KNAPSACK problem (which is NP-Complete), to an instance of FT-DR-OPT.

FT-DR-OPT: Given a set of imprecise computation tasks $\{T_i\}$, deadlines $\{d_i\}$ and an integer R representing the total reward of the system, is there an FT-Optimal schedule (according to the DR scheme) whose total reward is $\geq R$?

KNAPSACK: Let there be a set of items $U = \{u_1, u_2, \ldots, u_n\}$, integers s_i (size) and v_i (value) for each $u_i \in U$, positive integers B and K denoting the size of the knapsack and the value of the items in the knapsack, respectively. Is there a subset $U' \subseteq U$ such that $\sum_{u_i \in U'} s_i \leq B$ and $\sum_{u_i \in U'} v_i \geq K$?

Transformation: Let $S = \sum_{i=1}^{n} s_i$ and $V = \sum_{i=1}^{n} v_i$. Note that the KNAPSACK problem is NP-Complete only if $K \leq V$ and $B < S$ (for if $K > V$ then the answer is definitely 'no'; similarly if $B \geq S$ then the answer is quickly found by checking whether $V \geq K$). Hence, in the following discussion, we'll assume that $K + B < S + V$.

If an instance of KNAPSACK is given, we construct a corresponding instance of FT-DR-OPT with $n+1$ *independent* imprecise computation tasks. The first n tasks have the following characteristics:

$$m_i = s_i \quad o_i = v_i \quad b_i = 0 \quad d_i = K + S + V \quad i = 1, \ldots, n$$

We also introduce an extra task T_0 with the parameters:

$$m_0 = K \quad o_0 = 0 \quad b_0 = K \quad d_0 = 2K + B$$

For convenience, the schedule to fill out is shown in Figure 13. Now, consider the question: Is there an FT-Optimal schedule of $T_0, T_1, \ldots, T_n$ with the total reward $\geq V = \sum_{i=1}^{n} v_i$? We claim that this instance of FT-DR-OPT is equivalent to that of KNAPSACK.

First, note that $\sum_{i=0}^{n} o_i = V$, hence the inequality above can be readily considered as equality. The total reward is V, implying $t_i \geq o_i = v_i, \quad \forall i$. Moreover, the length of the schedule is $L = d_1 = \ldots = d_n = K + S + V = \sum_{i=0}^{n} m_i +$

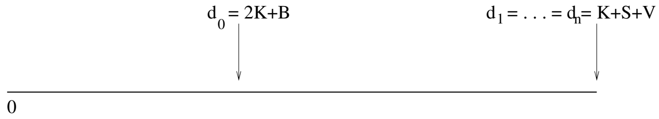

Figure 13. The schedule timeline

$\sum_{i=0}^{n} o_i$ which can be true only when $t_i = o_i \quad \forall i$ and when there exists no idle slots in the schedule (otherwise the total reward of $V = \sum_{i=0}^{n} o_i$ can not be reached). Thus the question above can be re-formulated as: "Is there an FT schedule with no idle slots and all optional parts are scheduled entirely?"

We note that $b_0 = K$ is the only recovery block whose execution time is larger than 0, that is, we require fault tolerance only for M_0. Finally, it is easy to see that M_0 and (some) optional parts of total length at least K should be scheduled before d_0 in any FT schedule. We prove the equivalence of two decision problem instances in two parts:

1. If there is a solution to the KNAPSACK instance, then there is also a solution to the FT-DR-OPT instance.

Proof: If U' is a solution set to the KNAPSACK instance, then $\sum_{u_i \in U'} s_i \leq B$ and $\sum_{u_i \in U'} v_i \geq K$.

Let $\alpha = \{M_i | u_i \in U'\}$, $\beta = \{O_i | u_i \in U'\}$, $\alpha' = \{M_i | u_i \in U - U'\}$ and $\beta' = \{O_i | u_i \in U - U'\}$. Consider the schedule given in Figure 14.

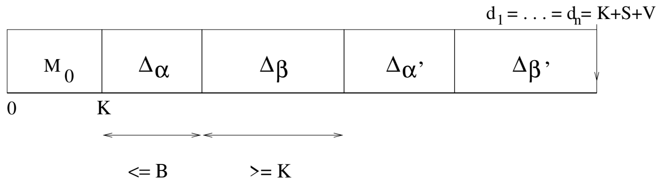

Figure 14. The FT-Optimal schedule

In this schedule Δ_X denotes the schedule segment where *only* tasks in the set X are scheduled. In other words, we schedule the tasks in the following order: M_0, the mandatory parts in α, the optional parts in β, the mandatory parts in α', the optional parts in β'. It should be clear that the schedule is feasible, contains no idle slots and $t_i = o_i \quad \forall i$.

However, in order to show that it is also Fault-Tolerant, we need to prove that $Slack(M_0) \geq b_0$, that is, the sum of optional parts scheduled before $d_0 = 2K + B$ is at least K. Recall that $m_0 = K$. Also, from the solution of the KNAPSACK: $|\Delta_\alpha| = \sum_{M_i \in \alpha} m_i \leq B$ and $|\Delta_\beta| = \sum_{O_i \in \beta} o_i \geq K$. Thus, d_0 is definitely after Δ_α and, $Slack(M_0) \geq d_0 - B - m_0 = d_0 - B - K$, which implies $Slack(M_0) \geq 2K + B - B - K$, or $Slack(M_0) \geq K$, completing the proof. □

2. If there is a solution to the FT-DR-OPT instance, then there is also a solution to the KNAPSACK instance.

Proof: Suppose that there exists a solution to the FT-DR-OPT instance, hence an FT schedule with total reward V (with no idle slots and $t_i = o_i \quad \forall i$). Clearly, $Slack(M_0) \geq K$ in this schedule. Let t_i^o be the optional service time of T_i in the interval $[0, d_0]$. We remark that $\sum_{i=1}^{n} t_i^o \geq K$, otherwise the slack constraint of M_0 can not be satisfied. Also, observe that if $t_i^o > 0$, then M_i definitely starts *and* completes in the interval $[0, d_0]$, since otherwise O_i can not start executing at all. Let $\gamma = \{i \,|t_i^o > 0 \text{ and } i > 0\}$. It is easy to see that the following relations hold:

$$\sum_{i \in \gamma} o_i \geq \sum_{i \in \gamma} t_i^o \geq K \tag{3}$$

$$\sum_{i \in \gamma} m_i \leq d_o - m_o - \sum_{i \in \gamma} t_i^o \leq 2K + B - K - K = B \tag{4}$$

Recalling that $m_i = s_i$ and $o_i = v_i$ for $i = 1, \ldots, n$, we conclude from (3) and (4) that $\sum_{i \in \gamma} m_i = \sum_{i \in \gamma} s_i \leq B$ and $\sum_{i \in \gamma} o_i = \sum_{i \in \gamma} v_i \geq K$, or equivalently $U' = \{u_i \,|\, i \in \gamma\}$ is the solution to KNAPSACK, completing the proof. □

6 FT-Optimality with IR scheme and non-identical ready times

Finally we prove that, once we allow non-identical ready times, the FT-Optimality problem with IR also becomes intractable. The reduction is again from the KNAPSACK instance mentioned in Section 5. The decision problem version we consider is FT-IR-NONID: Given a set of imprecise computation tasks $\{T_i\}$, deadlines $\{d_i\}$, ready times $\{r_i\}$ and an integer R, is there an FT-Optimal schedule with IR whose total reward is larger than or equal to R?

Given an instance of the KNAPSACK (s_i, v_i, B and K values), we construct an instance of FT-IR-NONID this time with $n + 2$ imprecise computation tasks. The first n tasks have the following characteristics:

$$m_i = s_i \;\; o_i = v_i \;\; b_i = 0 \;\; r_i = 0$$

$$d_i = K + S + V + 1 \;\; i = 1, \ldots, n$$

We introduce two additional tasks T_0 and T_{n+1} with the following parameters:

$$m_0 = K \;\; o_0 = 0 \;\; b_0 = K \;\; r_0 = 0 \;\; d_0 = 2K + B$$

$$m_{n+1} = 1 \;\; o_{n+1} = 0 \;\; b_{n+1} = 0$$

$$r_{n+1} = 2K + B \;\; d_{n+1} = 2K + B + 1$$

where r_{n+1} is the ready time of the task T_{n+1}. Note that again the only task which requires FT is T_0 and the task T_{n+1} is only ready to execute at the deadline of task T_0. Considering the ready time and the deadline of M_{n+1}, we obtain the timeline shown in Fig. 15.

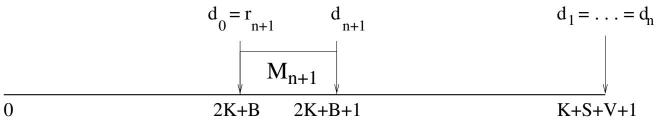

Figure 15. The schedule timeline

Now, consider the question: Is there an FT-Optimal schedule of $T_0, T_1, \ldots, T_n, T_{n+1}$ with the total reward larger than or equal to $V = \sum_{i=1}^{n} v_i$?

The need to provide FT for M_0 before $d_0 = r_{n+1} = 2K + B$ and the strict constraint that M_{n+1} can not be moved, imply that we must have at least K units of optional parts scheduled before d_0. Combining this with the requirement that every optional part should be used entirely to achieve the total reward V and hence, to fully utilize the timeline, we can infer that the interval $[0, 2K + B]$ may be used as a 'template' for the KNAPSACK problem. The details of the proof are analogous to those in Section 5 and omitted here for lack of space.

7 Conclusion

In this paper, we addressed the problem of incorporating fault tolerance to the IC model. By exploiting the (potential) time redundancy due to the existence of optional parts, we proposed and formalized two recovery schemes for mandatory parts whose timely completions are essential for the system.

In the immediate recovery, the recovery block is executed as soon as the fault is detected without preemption, while in the delayed recovery, the recovery block is to run only during slots originally reserved for optional slots. We showed that the two schemes yield non-identical FT schedules and then, we provided the FT conditions separately for each approach. Next, we considered the problem of generating FT-Optimal schedules in the context of each approach. In the case of the immediate recovery with identical ready times, we derived several properties of the FT-optimal schedules and based on these, we presented an efficient algorithm. Finally, we showed that the problem becomes NP-Hard if one allows non-identical ready times or commits to the delayed recovery scheme.

We consider the investigation of FT-optimality for tasks with precedence constraints, and the design of approximation algorithms for provably intractable cases as major avenues for the future work.

References

[1] H. Aydın, R. Melhem and D. Mossé. Incorporating Error Recovery in the Imprecise Computation Model. *Proceedings of the Sixth International Conference on Real-Time Systems and Applications (RTCSA'99)*, December 1999.

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

[3] 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.

[4] Jane 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.