Quality of Service Guarantee for Temporal Consistency of Real-Time Objects

Ming Xiong[1], Kam-Yiu Lam[2] and BiYu Liang[2,3]

Bell Laboratories[1]
600-700 Mountain Ave.
Murray Hill, NJ 07974, USA

Department of Computer Science[2]
City University of Hong Kong
83 Tat Chee Avenue, Kowloon
Hong Kong

Department of Mathematics[3]
Jinan University
Guangzhou 510632, P.R. China

Email:{cskylam@cityu.edu.hk; wleung@jnu.edu.cn; xiong@research.bell-labs.com}

Abstract

The More-Less (ML) scheme has been shown to be an efficient method for maintaining temporal consistency of real-time data objects. Although ML could provide a 100% guarantee in temporal consistency, the number of update transactions that can be supported in the system is severely limited due to its use of the worst-case computation time of the jobs from update transactions in scheduling and deadline assignment. In this paper, we extend ML for the soft/firm real-time database systems where the jobs from a transaction may have high variation in computation time and having a certain degree of temporal inconsistency is acceptable. We propose a new approach, called *Statistical More-Less (SML)*, to tradeoff between quality of service (QoS) of temporal consistency and the number of update transactions that can be scheduled in the system. To further improve the QoS, we extend the base algorithm of SML (SML-BA) by adding a slack reclaiming scheme (SML-SR) into it. The reclaimed slacks are reallocated for the admission of the jobs whose computation times are higher than the guaranteed value.

1 Introduction

In this paper, we study the problem on how to maintain temporal consistency of real-time data objects [3]. As explained in [2], an efficient method is the *More-less (ML)* scheme which is a *deterministic approach* as it can guarantee temporal consistency of real-time data objects. Although providing a 100 % guarantee in temporal consistency on real-time data objects is highly critical to hard real-time transactions, it is difficult to be achieved in soft/firm real-time database systems. Due to the less predictable behavior of soft real-time transactions, it is difficult to have the exact estimation on the computation time of soft real-time transactions. Since ML requires the use of worst-case computation time of transactions in determining the execution schedule, it may not be feasible to have a schedule to meet all the deadlines of the transactions. In this paper, we extend the More-Less scheme [2] with the objective to provide a *statistical* guarantee of real-time data objects.

2 Deterministic Approach: More-Less

$T = \{T_i\}_{i=1}^{m}$ refers to a set of periodic update transactions $\{T_1, T_2, ..., T_m\}$ and $X = \{X_i\}_{i=1}^{m}$ refers to a set of real-time data objects in a real-time database. Associated with X_i ($1 \le i \le m$) is the validity interval of length V_i. Update transaction T_i updates data object X_i. An update transaction T_i can be formally defined as: T_i: (C_i, D_i, P_i), where C_i, D_i and P_i denote the computation time, relative deadline, and period of T_i, respectively. D_i of T_i is defined as: deadline of T_i − sampling time of T_i.

The objectives of the deterministic approaches are to determine P_i and D_i such that the temporal consistency of the real-time objects can be ensured and at the same time all the update transactions are schedulable with minimum CPU workload. In ML, there are three constraints to follow to resolve the problems [2]:

1. *Deadline Constraint:* the period of an update transaction is assigned to be *more than half* of the validity interval of the data object to be updated by the transaction, while its corresponding relative deadline is assigned to be *less than half* of the validity interval of the same data object. For a transaction T_i to be schedulable, D_i must be greater than or equal to C_i, the worst-case execution time of T_i.
2. *Validity Constraint:* the sum of the period and relative deadline of T_i always equals V_i, the length of the validity interval of the data object updated.
3. *Schedulability Constraint:* deadline monotonic scheduling algorithm is used to schedule the set of update transactions.

The algorithm used by ML to obtain the deadlines and periods is shown in Appendix I.

3 Statistical Approach

For many soft/firm real-time database systems, it is usually acceptable to have certain percentages of transaction jobs to miss their deadlines. Therefore, a *statistical* approach may be adopted to control the quality of services (QoS) of freshness of data objects provided by update transactions, i.e., a parameter for adjusting the percentage of temporal consistency, as a tradeoff for the schedulability and total update workload in the system [1].

Definition: A *x%* statistical guarantee in temporal consistency for a real-time data object X_i is provided if at least *x%* jobs of a periodic transaction that updates X_i commit by their deadlines over an arbitrarily long period of time.

3.1 Principles and Base Algorithm of SML

Unlike ML, the *Statistical More-Less scheduling (SML)* schemes proposed in this paper are for the soft/firm real-time database systems where the computation time of the jobs from an update transaction T_i varies but follows certain distribution. The SML schemes are developed based on the assumptions: (1) The distribution of the computation time of an update transaction can be pre-analyzed offline; and (2) once an update transaction is released, its computation time is known.

We first introduce the base algorithm of SML (SML-BA) which has an *admission test* that determines when a job could be admitted, and when to reject a job to guarantee the promised QoS of real-time objects. We use the probability density function $f_i(t)$ to represent the distribution of c_i. Denote the QoS requirement of temporal consistency of the real-time data object being updated by T_i be $Q_i = x_i\%$, and let $F_i(x) = \int_0^x f_i(t)dt$ be the cumulative distribution function of the computation time of T_i. Solving the equation for x: $F_i(x) = Q_i$ to get the solution $x = \tilde{c}_i$. The probability that the computation time of the jth job from T_i, c_{ij} is less than $\tilde{c}_i$ is Q_i (as shown in Figure 1). We then treat T_i as a transaction with fixed computation time $\tilde{c}_i$. Then, we use $\tilde{c}_i$ instead of the worst-case computation time to derive deadlines and periods (D_i and P_i) following the approach proposed in ML. If the transaction set is schedulable using ML with deadlines and periods derived from $\tilde{c}_i$, T_i is guaranteed at least $\tilde{c}_i$ computation time for each period. Thus the requested QoS is guaranteed and the utilization cost for doing this is minimized.

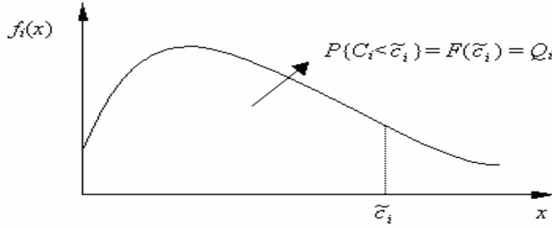

Figure 1: $\tilde{c}_i$ for guaranteeing statistical temporal consistency.

The following algorithm is the admission test in SML-BA.

```
If the jth job of Ti with computation
time requirement cij < c̃i then
       ADMIT the jth job of Ti
else
       REJECT the jth job of Ti
endif
```

Algorithm 1: Job Admission Algorithm in SML-BA.

If the system has spare capacity after guaranteeing the required QoS of each transaction, it may further improve the total QoS provided in the system by admitting more update transactions. This is an optimization problem. We need to find out a set of Δc_i to enlarge c_i to maximize $\tilde{Q}(c) = \sum_{i=1}^{m} \tilde{Q}_i(c_i)$. Let c_i, D_i, P_i be the guaranteed computation time, deadline and period derived from SML-BA after guaranteed computation time is incremented by Δc_i (i.e., $c_i = \tilde{c}_i + \Delta c_i$). T_i ($1 \le i \le m$) needs to satisfy the following four constraints.

1. ***Validity Constraint:*** $P_i + D_i = V_i$
2. ***Deadline Constraint:*** $c_i \le D_i \le P_i$
3. ***Schedulability Constraint:*** $\sum_{j=1}^{i} \lceil D_i / P_j \rceil \cdot c_j \le D_i$
4. ***QoS Constraint:*** $\tilde{Q}_i(c_i) \ge Q_i^*$ ($1 \le i \le m$)

In Appendix III, we provide an approximate algorithm for resolving the above optimization problem. The schedulability of the solution is ensured by the algorithm in Appendix II.

3.3 SML with Slack Reclaiming

Now, we discuss how to extend SML-BA by adding a reclaim policy on unused slacks of the jobs of transaction T_i ($1 \le i \le m$) whose computation times are smaller than the guaranteed computation time c_i, and the unused slacks from those rejected jobs of T_i. The extended scheme is called *SML with Slack Reclaiming (SML-SR)*. From here on, we refer to the unused computation times originally assigned to a job as slack.

3.3.1 Principles of SML-SR

The principle of SML-SR is based on the variation in computation times of the jobs from a transaction. We classify the jobs from a transaction into three categories based on their computation times: *short jobs*, *long jobs* and *over-long jobs*. Short jobs are those with computation time $c_{ij} \le c_i$. Following the admission test in SML-BA, all the short jobs of a transaction will be admitted. Since the computation time of a short job could be significantly smaller than c_i, the remaining amount of slack ($c_i - c_{ij}$) could be reclaimed for other jobs whose computation time is lager than c_i. Over-long jobs are those jobs with very large computation times. They have a higher probability to miss their deadlines even if they are admitted into the system. To determine what is an over-long job, we introduce a computation time value $\hat{c}_i$. If $c_{ij} > \hat{c}_i$, the job is an over-long job. In order not to waste computation resource on over-long jobs, they are rejected in the admission test. The computation times originally assigned to over-long jobs could be reallocated to long jobs whose computation times are larger than c_i, i.e. $c_{ij} > c_i$, but is smaller than $\hat{c}_i$. The admission of long jobs increases the QoS of temporal consistency provided to be higher than the initial guaranteed value. The main design problem in SML-SR is how to reclaim unused slacks from short jobs and over-long jobs for the admission of long-jobs of the same transaction.

In estimating the amount of reclaimed slack for a transaction, we need to determine the value for $\hat{c}_i$. One way to define $\hat{c}_i$ is to use the *least upper bound of the computation time* of the transaction. It is the maximum computation time that a job from a transaction may use before its deadline expired in the worst-case. Since a job of T_i may be preempted by the jobs from higher-priority transactions, it may be impossible for it to use up to D_i of computation time. Note that D_i of a transaction is pre-computed following the method proposed in ML.

Suppose the first jobs of all transactions are released at the same time. The least upper bound of computation time $\hat{c}_i$ that a job of T_i may use is: $\hat{c}_i = \hat{D}_i - \sum_{j=1}^{i-1} \lceil \hat{D}_i / P_j \rceil \cdot c_j$, where $\hat{D}_i$ is the relative deadline of T_i in the derivation of $\hat{c}_i$ for admitting long jobs. It is larger than the original relative deadline D_i (i.e., $\hat{D}_i > D_i$) as D_i is a tight deadline with the purpose to minimize the update workload. If $\hat{D}_i = D_i$, it makes $\hat{c}_i = c_i$, and only short jobs with computation times less than or equal to c_i can be guaranteed to meet their deadlines. A job of T_i that uses more

than c_i (the long jobs) will be rejected in the admission test. To increase the probability of long-jobs to be admitted (this is important to improve the QoS higher than the requested QoS value), we propose to enlarge the deadline $\hat{D}_i$ to be greater than D_i of transaction T_i, which is originally obtained from the deadline assignment scheme in ML, by using the concept of *equilibrium state*.

3.3.2 Calculation of Unused Slacks

If J_{ij} is a short job, i.e. $c_{ij} \le c_i$, then the job will be admitted. The average value of this kind of slack to be reclaimed within one period of T_i is: $SLK_i^- = \int_0^{c_i}(c_i - s)f_i(s)ds$. If J_{ij} is an over-long job, i.e., $c_{ij} > \hat{c}_i$, it will be rejected. A large amount of slack c_i, which is previously guaranteed to the rejected over-long job, will be reclaimed to T_i. The average value of slack that will be reclaimed during one period of T_i is: $SLK_i^+ = \int_{\hat{c}_i}^{+\infty} c_i f_i(s)ds$. So the average slack to be reclaimed from short jobs and over-long jobs in one period of T_i is: $SLK_i = SLK_i^- + SLK_i^+$. We call SLK_i as the *predicted slack* in one period of T_i. If J_{ij} is a long job, i.e. $\hat{c}_i \ge c_{ij} > c_i$, it may be admitted using guaranteed computation time together with the reclaimed slack of T_i to supplement the exceeded computation time ($c_{ij} - c_i$). If the total slack available is not sufficient for the exceeded computation time of the job, it will be rejected, and the previous guaranteed computation time c_i for it will be reclaimed as slack of T_i. We call the slack reclaimed from rejected long jobs as *run-time slack*.

3.3.3 Enlarging Deadlines for Admission of Long Jobs

To increase the number of long jobs and their probability to be admitted, we propose to enlarge the relative deadline D_i to $\hat{D}_i$ based on the concepts of *equilibrium state* and *balance equation* to make the least upper bound of computation time $\hat{c}_i$ significantly greater than c_i. The purpose of the *equilibrium state* is to find the value for $\hat{c}_i$ that will give a proper definition of long jobs such that the total demands of additional computation times from them is close to and not greater than the total slack reclaimed from short jobs and over-long jobs. Then, we enlarge D_i of T_i according to its share of unused computation time. Of course, increasing D_i reduces P_i as we need to keep $D_i + P_i = V_i$ to guarantee the validity of the data object. Once D_i is enlarged, we recalculate the least upper bound of computation time $\hat{c}_i$ based on the enlarged D_i.

Transaction T_i is under e*quilibrium state* if in the long run the total amount of reclaimed slacks of T_i equals to the total amount of excess demand of computation time from the long jobs of T_i. Suppose the least upper bound of computation time $\hat{c}_i'$ gives an e*quilibrium state*. Then, according to the definition of short jobs, long jobs and over-long jobs, we have:

$$\int_{c_i}^{\hat{c}_i'}(s - c_i)f_i(s)ds = \int_0^{c_i}(c_i - s)f_i(s)ds + \int_{\hat{c}_i'}^{+\infty} c_i f_i(s)ds .$$

We call the above equation as a *balance equation*. The term on the left hand side is the average of the excess demand of computation time by long jobs when the least upper bound of computation time is $\hat{c}_i'$ while the sum of the two terms on right hand side is the average slack from short jobs and over-long jobs when the least upper bound of computation time is $\hat{c}_i'$. We solve this balance equation to get $\hat{c}_i' = \hat{c}_{eq}$. It can be seen that if we set the least upper bound of computation time to be $\hat{c}_i = \hat{c}_i'$ the transaction is under e*quilibrium state*. If we set $\hat{c}_i < \hat{c}_i'$, more slack is reclaimed than the demanded in the long run. If we set $\hat{c}_i > \hat{c}_i'$, the total demand of slack from long jobs is larger than the total slack reclaimed in the long run. The consequence is that some admitted jobs may miss their deadlines and the requested computation resources may be wasted. Such a situation should be avoided. Of course $\hat{c}_i$ may not always be set as large as $\hat{c}_i'$. For example, when $\hat{c}_i' > V_i/2$, according to ML, $\hat{c}_i < D_i < V_i/2 < \hat{c}_i'$. However, the closer is $\hat{c}_i$ to $\hat{c}_i'$, the better will it be (i.e. the slack produced and consumed is closer to the state of balance). That is to say, we need to solve the following optimization problem:

$$\min \; F_D(\hat{D}) = \left\| \hat{c} - \hat{c}_{eq} \right\|$$

s.t. $\hat{c} \le \hat{c}_{eq}$;

$$\hat{c}_i = \hat{D}_i - \sum_{j=1}^{i-1} \left\lceil \hat{D}_i / P_j \right\rceil \cdot c_j \ge c_i ;$$

$\hat{D}_i \le V_i/2$; $\hat{P}_i = V_i - \hat{D}_i$.

Appendix V shows the algorithm to solve the above optimization problem.

3.2.2 Job Admission Algorithm in SML-SR

There are three main considerations in the design of the slack assignment policy and admission test in SML-SR:

(1) To maximize the number of admission of long jobs. In our slack assignment policy, we use total slack instead of just the actual amount of slack reclaimed. The total slack includes both predicted slack and runtime slack.

(2) To prevent the admission of a long job of a higher priority transaction from intruding the execution of jobs of lower priority transactions causing them to miss their deadlines. In allocating reclaimed slack to a job from transaction T_i, we consider the QoS of data objects to be updated by lower priority transactions (T_{i+1}, T_{i+2}, ...). It is important to ensure that the scheduling of lower priority jobs is not affected by the long job admitted so that QoS of the data objects to be updated by lower priority transactions can be guaranteed. Also, there exists a maximum amount of slack to be used by a job since we need to guarantee the QoS of temporal consistency of the data objects to be updated by lower priority transactions at the same time.

(3) To prevent a long job whose computation time is very large but it is still shorter than that of an over-long job to use up all the reclaimed slack. Rejecting such a job could reallocate the slack to several long jobs and is beneficial to the overall QoS of the data object.

For the first two considerations, we define two new terms: *k-super-interval* and *k-slack-budget.*

Definition: *The **k-super-interval** of T_i is a time interval of length D_k starting from the release of a job of T_k, where $k > i$.*

Corresponding to each lower priority transaction (T_{i+1}, T_{i+2} ...), T_i has a k-super-interval.

Example: Figure 2 gives an example of k-super-intervals of transaction T_i. In the figure, solid line ticks are periods, and long dotted line ticks are deadlines. In this example, there are 2, 5 and 9 periods of T_i in its (i+1)-super-intervals, (i+2)-super-intervals and k-super-intervals respectively. Each released job has a set of k-super-intervals for each of its lower priority transaction.

It can be seen that in the k-super-interval of T_i there are $n = \lfloor D_k/P_i \rfloor$ consecutive periods of T_i. The predicted slacks of T_i in the n consecutive periods contained in the same k-super-interval are aggregated to be $n \times SLK_i$. The actual amount of slack of T_i may be higher after including the run-time slack of T_i.

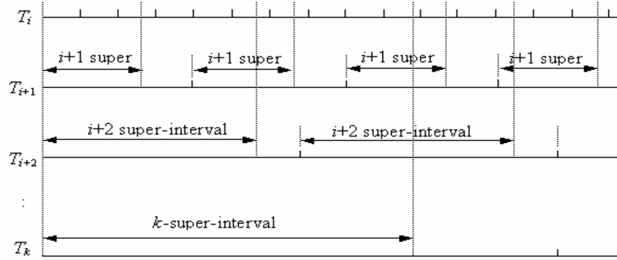

Figure 2: Illustration of the *k*-super-intervals of T_i.

Definition: *The **k-slack-budget** of T_i is the slack budget (the current amount of slack) of T_i in a k-super-interval of T_i.*

The k-super-intervals are important to limit the amount of slack that can reallocated for admission of long jobs of the same transaction and at the same time the QoS guaranteed to lower priority transactions can still be maintained. In our slack assignment policy, variation in computation times of the jobs from a transaction are smoothed through aggregation of reclaimed slacks along every of the k-super-intervals of T_i.

At the beginning of each k-super-interval of T_i, the corresponding k-slack-budget b_{ik} is replenished with the total aggregation of predicted slacks during that k-super-interval, i.e. $\lfloor D_k/P_i \rfloor \cdot SLK_i$, which indicates the amount slacks can be allocated to the long jobs of T_i within the k-super-interval. During a k-super-interval of T_i, the total computation time allocated to all the jobs of T_i (both short and long jobs) should not be larger than the total computation time guaranteed to T_i in a k-super-interval, i.e. $\lfloor D_k/P_i \rfloor \times c_i$. We maintain a run-time counter matrix $A_{m \times m}$. a_{ik} is the cumulative computation time used by T_i since the beginning of the current k-super-interval. a_{ik} should always be less than $\lfloor D_k/P_i \rfloor \times c_i$. If $a_{ik} + c_{ij} > \lfloor D_k/P_i \rfloor \times c_i$ for some $k > i$, i.e. when $c_{ij} > \min_{k>i, n_{ik} \neq 0} \{\lfloor D_k/P_i \rfloor \times c_i - a_{ik}\}$, the job must be rejected. Otherwise, the execution of the job will infringe on the time previously guaranteed to current job of some T_k $(k > i)$. When a long job of T_i with computation time c_{ij} is admitted, each of its k-slack-budget for T_k (k>i) is reduced by the amount of the slack time requested by the long job, i.e., $(c_{ij} - c_i)$. Using this technique, long jobs can use the slacks before they are reclaimed later in the same k-super-interval. On the other hand, if a long job is rejected due to insufficient slack, c_i of run-time slack is reclaimed and added to each of T_i's k-slack-budget.

For the final consideration, in the slack assignment policy in SML-SR, the reclaimed slack (k-slack-budget) of T_i is shared equally among the remaining periods of T_i in the current k-super-intervals. Thus, jobs are prevented from requesting a large amount of available slack, which can cause the rejection of the jobs with smaller slack requirements. In order to implement this slack assignment policy, we use a matrix $N_{m \times m}$ to maintain the remaining number of periods of each transaction during each of its current k-super-interval. n_{ik} is the number of remaining number of periods of T_i in the current k-super-interval. n_{ik} is initialized to be $\lfloor D_k/P_i \rfloor$ at the beginning of each k-super-interval of T_i (i.e. a job of T_k is admitted). It is reduced by 1 at the beginning of each period of T_i. If $n_{ik} = 0$, it means that there is currently no k-super-interval of T_i. Note that not all of these n_{ik} periods request extra computation time, and only long jobs will. The average number of long jobs in the n_{ik} periods is $n_{ik} \times m_i$, where $m_i = \int_{\bar{c}_i}^{\hat{c}_i} f_i(s)ds$ is the probability that a job is a long job. So in our slack assignment policy, we divided the k-slack-budget among the remaining long jobs and there are $\lceil n_{ik} \times m_i \rceil$ of them.

The algorithm of SML-SR is shown in Appendix VI. For $k > i$, if $n_{ik} \neq 0$, then there is a k-super-interval containing the arrived long job. The slack available for the long job must be no greater than $b_{ik} / \lceil n_{ik} \times m_i \rceil$, for all k satisfying $k > i$ and $n_{ik} \neq 0$, i.e., it must be no greater than $\min_{k>i, n_{ik} \neq 0} \{b_{ik} / \lceil n_{ik} \times m_i \rceil\}$. Considering the least upper bound of computation time $\hat{c}_i$, the amount of slack available for the long job of T_i upon its arrival is $\min\{\hat{c}_i - c_i, \min_{k>i, n_{ik} \neq 0} \{b_{ik} / \lceil n_{ik} \times m_i \rceil\}\}$. This is the maximum slack that can be used for its admission.

4 Conclusions and Future Work

In this paper, we present the design of the algorithms for quality of service (QoS) guarantee of real-time data temporal consistency. We propose a new approach, namely *Statistical More-Less (SML)*, to tradeoff between QoS of temporal consistency and the number of update transactions that can be scheduled in the system. To improve the QoS of baseline SML algorithm (SML-BA), we extend SML-BA by adding a slack reclaiming scheme (SML-SR) into it. The reclaimed slacks are reallocated for the admission of the jobs whose computation times are higher than the guaranteed value.

References

[1] A. Atlas and A. Bestavros, "Statistical Rate Monotonic Scheduling," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1998, Madrid, Spain.

[2] M. Xiong and K. Ramamritham, "Deriving Deadlines and Periods for Real-Time Update Transactions," in *Proceedings of 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

[3] M. Xiong, K. Ramamritham, J.A. Stankovic, Don Towsley and R. Sivasankaran, "Scheduling Transactions with Temporal Constraints: Expoiting Data Semantics ", *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no.5, pp 1155-1166.

Appendix I: Determining deadlines and periods in *More-Less* [3]

```
/* Compute the deadline and period of τ1*/
D1 = C1;  // Ci is the computation time of τi
P1 = α1 - D1; // αi is the validity length of τi

/* Compute Di and Pi for the rest of the tasks in the descending order of task priorities */

for i = 2 to m do
{
        Ri1 = Ci; /*Initialize Ri1, response time of Ji1*/
        do {/* Compute Ri1 iteratively*/
                Di = Ri1; /* Keep Ri1 for comparison*/
                Ri1 = Ci; /*Initialize Ri1 to recomputed it */
                /*Next, recompute Ri1 using Di*/
                for j =1 to i – 1 do
                /*Account for the interference of higher priority tasks*/
                { Ri1 = Ri1 + ⌈Di/Pj⌉ Cj ; }
        } while (Ri1 ≠ Di) and (Ri1 ≤ αi/2)

/*Computation of Ri1 stops if Ri1 does not change, or Ri1 exceeds */

if (Ri1 > αi/2)
then abort; /*Unscheduleable case*/
else Pi = αi - Di; /*Compute Pi */
}
```

Appendix II: Schedulability test in ML

```
/* Test if the computation time vector c = (c1, c2,…, cm)^T is schedulable with ML */
/* Use by Appendix III */
/* Compute the deadline and period of T1*/
D1 = c1;  // ci is the computation time of Ti
P1 = V1 - D1; // Vi is the validity length of Ti
/* Compute Di and Pi for the rest of the tasks in the descending order of task priorities */
for i = 2 to m do
{
    Ri1 = ci; /*Initialize Ri1, response time of Ji1*/
    do {/* Compute Ri1 iteratively*/
        Di = Ri1; /* Keep Ri1 for comparison*/
        Ri1 = ci; /*Initialize Ri1 to recomputed it */
        /*Next, recompute Ri1 using Di*/
        for j =1 to i – 1 do
      /*Account for the interference of higher priority tasks*/
            { Ri1 = Ri1 + ⌈Di/Pi⌉ cj ; }
      } while (Ri1 ≠ Di) and (Ri1 ≤ Vi/2 )
    /*Computation of Ri1 stops if Ri1 does not change, or Ri1 exceeds Vi/2 */

    if (Ri1 > Vi/2 ) then  return NON-SCHEDULABLE ;

    /*Unscheduleable case*/

    else Pi = Vi - Di; /*Compute Pi */
}

return SCHEDULABLE;
```

Appendix III: Finding the optimal guaranteed computation time that maximize the guaranteed QoS

```
/* Part 1: search the optimum up to the More-Less boundary */
/* initialized with the computation time for guaranteeing Q* */
c = (c̃1, c̃2,..., c̃m)^T;
h = 0.1 * max_{1≤i≤m} |Vi / 2 − ci| ; /* initialized step length */
c′ = c;
IS_SCHEDULABLE = True;
/* ε is a very small number, e.g. 10^-8 */
while (IS_SCHEDULABLE == True Or |h| > ε ) {
    /* use the schedulable point C’ as the new start point */
    c = c′;
    /* the steepest direction */
    p = (∂Q̃(c)/∂c1, ∂Q̃(c)/∂c2, ..., ∂Q̃(c)/∂cm)^T ;
    IS_SCHEDULABLE = False;
    do {
        /* move forward one step according to the gradient */
        c′ = c + h · p;
        if ( c′ not schedulable with More-Less)
            /* using Appendix II */
            h = h/2;  /* reduce step length */
        else
            IS_SCHEDULABLE = True;
      } while (IS_SCHEDULABLE == False Or |h| > ε);
}
/* Part 2: further optimization when it not schedulable on the gradient direction */
/* smaller step length near the boundary */
h = 10^−3 * max_{1≤i≤m} |Vi / 2 − ci| ;
while ( p > 0 ) {
    while ( c + h · p is not schedulable with More-Less ) {
        /* find the component with min QoS contribution */
        find k : st.  pk == min_{1≤i≤m, pi≠0} pi ;
        /* make the k^th component of the direction vector 0 */
        p = (p1,…, pk = 0,…, pm)^T;
    }
    if ( p == 0 ) then break; // no improvement in any direction
    c = c + h · p;
    p = (∂Q̃(c)/∂c1, ∂Q̃(c)/∂c2, ..., ∂Q̃(c)/∂cm)^T ; // update search direction */
        }
```

Appendix IV: Feasibility test on $\hat{D}$

```
/* Test if the deadline vector D̂ = (D̂1, D̂2,..., D̂m)^T satisfies the constraints of the optimization problem */
/* V = (V1,V2,...,Vm)^T , P̂ = (P̂1, P̂2,..., P̂m)^T */
if( D̂ > V / 2 ) then return UNFEASIBLE;
P̂ = V − D̂ ;
ĉ1 = D̂1 ;
for i = 2 to m do
```

{

$\hat{c}_i = \hat{D}_i - \sum_{j=1}^{i-1} \lceil \hat{D}_i / \hat{P}_j \rceil \cdot c_j$;

if($\hat{c}_i < c_i$) **then return** UNFEASIBLE;

}

if($\hat{c} > \hat{c}_{eq}$) **then return** UNFEASIBLE;

return FEASIBLE;

Appendix V: Finding the optimal $\hat{D}$ that leads to a state closest to equilibrium state

/* Part 1: search the optimum up to the constraint boundary */

// initialized with the deadlines derived for guaranteeing $\widetilde{Q}(c)$

$\hat{D} = (D_1, D_2, ..., D_m)^T$;

$h = 0.1 * \max_{1 \le i \le m} |V_i / 2 - D_i|$; /* initialized step length */

$\hat{D}' = \hat{D}$;

IS_FEASIBLE = **True**;

/* ε is a very small number, e.g. 10^{-8} */

while (IS_FEASIBLE == **True Or** $|h| > \varepsilon$) {

$\hat{D} = \hat{D}'$; // use the feasible point $\hat{D}'$ as the new start point

/* the steepest descending direction */

$p = -(\frac{\partial F_D(\hat{D})}{\partial \hat{D}_1}, \frac{\partial F_D(\hat{D})}{\partial \hat{D}_2}, ..., \frac{\partial F_D(\hat{D})}{\partial \hat{D}_m})^T$;

IS_FEASIBLE = **False**;

do {

/* move forward one step according to the gradient */

$\hat{D}' = \hat{D} + h \cdot p$;

if ($\hat{D}'$ does not satisfy the constraints)

/* using Appendix IV */

$h = h/2$; /* reduce step length */

else

IS_FEASIBLE = **True**;

} **while** (IS_FEASIBLE == **False Or** $|h| > \varepsilon$);

}

/* Part 2: further optimization when it not schedulable on the gradient direction */

/* smaller step length near the boundary */

$h = 10^{-3} * \max_{1 \le i \le m} |V_i / 2 - D_i|$;

while ($p > 0$) {

while ($\hat{D} + h \cdot p$ does not satisfy the constraints) {

/* find the component with minimum decrease */

find k : st. $p_k == \max_{1 \le i \le m, p_i \ne 0} p_i$;

/* make the kth component of the direction vector 0 */

$p = (p_1, \ldots, p_k = 0, \ldots, p_m)$;

}

if ($p == 0$) **then** /* no improvement in any direction */

break;

$\hat{D} = \hat{D} + h \cdot p$;

/* update search direction */

$p = -(\frac{\partial F_D(\hat{D})}{\partial \hat{D}_1}, \frac{\partial F_D(\hat{D})}{\partial \hat{D}_2}, ..., \frac{\partial F_D(\hat{D})}{\partial \hat{D}_m})^T$;

}

Appendix VI: Algorithm of SML-SR for job *j* of T_i. (It is performed upon the arrival of a job of T_i.)

if $c_{ij} \le c_i$ **then** { /* short job */

Admit the job;

for $k < i$ **do** { /* beginning of *i*-super-interval of T_k */

$n_{ki} := \lfloor D_i / P_k \rfloor$;

$b_{ki} := n_{ki} \cdot SLK_k$;

$a_{ki} := 0$;

}

for $k > i$ **do** {

if $n_{ik} > 0$ **then** { /* there is a *k*-super-interval of T_i */

$n_{ik} := n_{ik} - 1$;

$a_{ik} := a_{ik} + c_{ij}$;

}

}

}

else if $c_{ij} - c_i \le \min\{ \hat{c}_i - c_i , \min_{k > i, n_{ik} \ne 0} \{b_{ik} / \lceil n_{ik} \cdot m_i \rceil\}\}$ **and** $c_{ij} \le \min_{k > i, n_{ik} \ne 0} \{ \lfloor D_k / P_i \rfloor \cdot c_i - a_{ik}\}$ **then** {

/* long job and the slack available is enough */

Admit the job;

for $k < i$ **do** { /* beginning of *i*-super-interval of T_k */

$n_{ki} := \lfloor D_i / P_k \rfloor$;

$b_{ki} := n_{ki} \cdot SLK_k$;

$a_{ki} := 0$;

}

for $k > i$ **do** {

if $n_{ik} > 0$ **then** { /* there is a *k*-super-interval of T_i */

$n_{ik} := n_{ik} - 1$;

$a_{ik} := a_{ik} + c_{ij}$;

$b_{ik} := b_{ik} - (c_{ij} - c_i)$; // reduce *k*-slack budget of T_i

}

}

}

else { /* over-long job */

Reject the job;

for $k > i$ **do** {

if $n_{ik} > 0$ **then** {

$n_{ik} := n_{ik} - 1$;

/* add run-time slack to the slack budget */

$b_{ik} := b_{ik} + c_i$;

}

}

}